Digital M^{c}Logic Design

Bryan J. Mealy & James T. Mealy

© Bryan J. Mealy 2012

Table of Croutons

Acknowledgements

Thanks to the few people who encouraged me on this project. There are too few people to list here. A special thanks to my amazing father, James T. Mealy, who agreed to help me after I twisted his arm. More than anything, I appreciate that he shared the excitement I felt about doing something meaningful.

A few people provided me with encouragement for this book. I've thanked these people in person and will not thank them again here. One person provided me with this quote, which I feel is 100% appropriate in an academic setting:

> If you do not wish a man to do a thing, you had better get him to talk about it; for the more men talk, the more likely they are to do nothing else.
>
> -Carlyle-

> Hey Dickson… Someday we'll work together on all the things we've yet completed. I'm looking forward to that day.

Rambling Commentary

My inspiration for this project came from my own personal notion that knowledge, particularly technical knowledge, should not be held ransom by publishing companies, bookstores, and book authors. Students seeking knowledge are particularly vulnerable when it comes to the notion of structured learning situations such as colleges. Being that students are the lowest hanging fruit, they always are the first to have their wallets lightened. I hope this book serves as an alternative to shelling out money for overpriced textbooks.

This book is going to have errors. Please accept my sincerest apologies for the errors you will come across. I did my best to remove errors, but, there are two main factors that mitigated my error removing initiative.

1. Writing and proofreading is somewhat timing consuming.

2. The Fall 2012 quarter is almost here (as of this writing); I need to output a version of this book for the courses I'll be teaching.

I could spend the remainder of my life tweaking this text, but I need to move onto other things. By all means, feel free to contact me with corrections and comments. Please don't write me at my ".edu" address as that account is generally filled with pointless academic drivel (which I tend to ignore) or email from my students (which I attend to with a high priority). Please use this address to write me: digitalmclogic@gmail.com.

There were two primary negative comments I received when I mentioned I was writing a textbook and was planning to give it away at no cost.

- "If you don't charge something, people will not value it". I don't understand this statement. The things I value most in my life were given to me. Maybe I'm missing something here.

- "You need to have experts in your field review your text". As a college teacher, I constantly receive requests from book companies to "review" one of their texts. They always sweeten the deal with an offer of cash. I know of no one who is going to dedicate any significant amount of their time to reading a text they care nothing about, but I know of people who pretend to review books, write down some drivel, and receive their cash. Wow! Great review! A book is a mechanism to transfer knowledge; it's not a popularity contest, as are most things in academia.

Finally, as you read this book, you may get the impression that I don't like academic administrators. I believe that all employees of a school should be serving students and that schools exist to help students learn.

Overview of Chapter Overviews

This text presents introductory digital logic design concepts and introductory VHDL modeling concepts. This book focuses on block-level design and has removed many of the unimportant low-level digital design details typically found in introductory digital design texts. This book results from my experiences from working with students and teaching digital logic design.

Chapter 1:
This chapter presents a relatively quick overview of both the digital design and the approach this textbook takes to introducing digital design. The digital design overview includes a brief history of digital design and commentary regarding current digital design courses and textbooks. This chapter also includes a general outline describing the nominal structure of all the chapters in this text. *This chapter is important because it provides a context for this text by describing the structure, style, and text content.*

Chapter 2:
This chapter introduces the notion of analog vs. digital and introduces the concept of modeling. This textbook is about digital design so this chapter provides an intuitive approach to the differences between analog and digital. Additionally, modern digital design is primarily concerned with various forms of modeling; this chapter also introduces an intuitive introduction to black box modeling in a generic sense (nothing associated with digital design however). *This chapter is important because describes the notion "digital" and provides an overview of basic modeling concepts.*

Chapter 3:
This chapter provides an introduction to the important issues involved in digital design. Some of these important issues include a hierarchical design overview, a introduction to various approached to design, a quick overview of the digital devices covered in this text, and issues regarding their seeming overlap between computer science and digital design. *This chapter is important because it provides basic information regarding digital design including the basic digital design paradigm and levels of abstraction. These concepts help this text's approach to digital design in an appropriate context.*

Chapter 4:
This chapter starts with a description and justification for using engineering notation for here and evermore. The chapter continues with a brief introduction to numbers as we know and love them. Included in this introduction is the relation between numbers and digital design. Other fun stuff includes an overview of juxtapositional notation, and an introduction to binary numbers. The presentation of binary numbers in this chapter provides only the information required to get through some of the upcoming chapters; later chapters provide a more complete description of useful number forms used in digital design. *This chapter is important because it provides a description of engineering notation and the basic form of numbers. This chapter also introduces the basic concepts of working with binary number representations.*

Chapter 5:
This chapter provides the first digital design experience mixed with an introduction to Boolean algebra. The chapter uses the "Brute Force Design" approach as its digital design approach and includes the notion of truth tables, basic logic gates, and schematic diagrams. This chapter also covers the basic development of Boolean algebra and its associated axioms and theorems. Other topics included in this chapter are functional equivalency and an introduction to the "half adder" circuit. *This chapter is important because it provides a basic approach to solving digital design problems. This particular approach employs Boolean algebra, which represents the foundation of all digital design.*

Standard Digital Circuits: half adder (HA)

Chapter 6:
This chapter provides an overview of the usefulness of DeMorgan's Theorem. While this text does not cover many digital theorems in much depth, this chapter covers most aspects of DeMorgan's theorem with emphasis on its ability to generate new representations of functions including both SOP and POS forms. *This chapter is important because it describes how to use DeMorgan's theorem to change Boolean expressions into functionally equivalent forms.*

Chapter 7:
This chapter provides an introduction to timing diagrams. Timing diagrams are found in all aspects of digital design as they are extremely useful for both the modeling and testing of digital circuits. This chapter also introduces the notion of bundle notation in digital design regarding both timing diagrams and circuit diagrams. *This chapter is important because it describes the use of timing diagrams to model typical digital circuit operations.*

Chapter 8:
This chapter introduces the remainder to the standard digital logic gates including NAND, NOR, XOR, and XNOR gates. The chapter also presents the concepts of using standard gates as inverters, switches, and buffers. *This chapter is important because it describes three of the more common logic gates used in digital design.*

Standard Digital Circuits: full adder (FA)

Chapter 9:
This chapter provides an overview of a few of the more popular methods of used to represent functions. The notion of representing functions is typically paired with representing functions in some relatively minimal form. As such, this chapter also introduces the concepts of function reduction using Karnaugh mapping techniques including the use of stripes and diagonals associated with XOR & XNOR functions. *This chapter is important because it describes more methods of representing functions including reduced representations using Karnaugh maps.*

Chapter 10:
This chapter describes some of the most useful ways of representing circuits using Boolean equations. These representations are also useful representing circuits, particular the NAND/NAND and NOR/NOR forms. This chapter also introduces the basic theory and approach behind minimum cost concepts, which is a popular notion in digital design. *This chapter is important because the circuit forms provide a significant amount of flexibility when it comes to representing functions. This flexibility allows you to implement function using the minimum possible cost.*

Chapter 11:
This chapter provides an introduction to designing digital circuits using the Iterative Modular Design (IMD) approach. Up until this chapter, all circuit were designed with the Brute Force Design (BFD) approach. The chapter uses the IMD approach to design standard digital circuits including ripple carry adders, comparators, and parity generators/checkers. *This chapter is important because it introduces the concept of "iterative modular design" (IMD). This chapter uses the IMD approach to design four standard digital circuits:*

Standard Digital Circuits: ripple carry adder (RCA), comparator, parity generator, and parity checker.

Chapter 12:
This chapter provides somewhat of an overview of programmable logic devices (PLDs). The overall notion here is that the details of these devices are not included in this text in order that we can spend more time actually designing digital circuits. There are many great sources of information on PLDs at a low level; this text simply is not one of them. *This chapter is important because it provides an overview of programmable logic devices, including a brief history and brief architectural overview. This chapter provides a context for PLDs and their use in modern digital design.*

Chapter 13:
This chapter the background necessary to work with number systems typically associated with all things digital including binary, octal, and hexadecimal representations. This chapter places reasonable effort in describing conversion techniques between these representations as well as overview of binary coded decimals (BCDs) and unit distance codes (UDCs). *This chapter is important because a significant portion of digital design deals with numbers and their various representations. Understanding of these representations, including conversions between representations, will help all digital designers.*

Chapter 14:
This chapter provides the background and description of the most common signed and unsigned binary number representations including signed magnitude (SM), radix complement (RC), and diminished radix complement (DRC); most of the emphasis is on DRC (or 2's complement) due to its popularity. This chapter introduces the concept of number ranges associated with fixed-bit number representations and the arithmetic associated with these representations with emphasis on the validity of addition and subtraction operations. *This chapter is important because is describes the basic representations of signed and unsigned binary numbers. In addition, this chapter describes mathematical operations (addition and subtraction) on binary numbers, which form the basis of many digital circuits.*

Chapter 15:
This chapter provides on overview of the design methods discussed so far and then delves into block-level design. Block-level design, or modular design, is the most powerful method of designing digital circuits as it support the notion of module generation and usage in addition of basic hierarchical design techniques. *This chapter is important because the modular design approach is the most powerful of all digital design approaches. It is hierarchical in nature and thus works well with VHDL structural modeling techniques. Most importantly, modular design supports the understanding of complex digital designs by directly supporting the hierarchical designs.*

Chapter 16:
This chapter provides an introduction to the basic theory and approach to VHDL. The chapter introduces the basic notion of entity and architectures as well as the basic tenets of modeling digital circuits using VHDL. *This chapter is important because it describes the principles behind VHDL. Modern digital design uses VHDL extensively to design and test digital circuits.*

Chapter 17:
This chapter describes Structural Modeling in VHDL. Structural modeling is the method VHDL uses implement hierarchical designs, which is the preferred method of modeling any remotely complicated design in VHDL. *This chapter is important because it describes VHDL structural modeling, which is a modeling approach that supports hierarchical design with VHDL.*

Chapter 18:
This chapter provides more depth to the notion of VHDL modeling. While previous chapters discussed VHDL from a higher-level, this chapter presents some of the lower-level details in the context of typical digital circuits. In particular, this chapter presents an overview of the four types of concurrent statements in VHDL and the three common types of modeling typically associated with VHDL. *This chapter is important because it provides the post-intro basics of modeling digital circuits using VHDL including the various flavors of concurrent statements*.

Chapter 19:
This chapter provides an overview of decoders. This text considers decoder to be any type of combinatorial digital circuit that can be modeled using a look-up table. This text also considers there to be two types of decoders: standard decoders and generic decoders, where the standard decoder is a special case the generic decoder. This chapter also describes VHDL models for the various flavors of decoders. *This chapter is important because it describes both the generic and standard decoders. Generic decoders as extremely useful in digital design based on their ability to implement circuits modeled using a table format*

Standard Digital Circuits: Generic decoders and standard decoders

Chapter 20:
This chapter presents the multiplexor, or MUX. The MUX is another standard digital device that is used as a "selector" circuit in digital design. Selecting between one of more inputs is a useful and highly used operation in digital design. This chapter introduces the underlying structure of a simple MUX as well as introducing various approaches to modeling different flavors of MUXes. *This chapter is important because it describes the Multiplexor, a standard digital model that serves as the basic "selection" mechanism in digital design.*

Standard Digital Circuits: Multiplexors (MUXes)

Chapter 21:
This chapter introduces the notion that digital circuits are physical devices and must handle issues caused by propagation delays inherent to the devices. Until now, we've modeled devices as being idea; this chapter deals with some of the issues involved with models that better represent the actual operating characteristics of the devices. This chapter includes the notion of glitches caused by static logic hazards and suggests methods to correct them. This chapter also includes several more issues regarding timing diagrams and associated annotations. *This chapter is important because it introduces non-idealized circuit models. Real digital circuits contain propagation delays, which can cause unwanted characteristics such as glitches.*

Chapter 22:
This chapter covers the theory behind mixed logic. While previous chapters have dealt primarily with positive logic (while avoiding the issue of negative logic), this chapter presents issues regarding both positive and negative logic. This chapter discusses both direct polarity indicators and the positive logic conventions in the design and analysis of Boolean logic in mixed logic systems. *This chapter is important because it provides the theoretical foundation for designing and analyzing mixed logic circuits.*

Chapter 23:
This chapter presents the theory behind mapped entered variables (MEVs) and their usage in various aspects of modeling digital devices. This chapter also discusses the relation between MEVs and Karnaugh map compression. This chapter uses both of these techniques for the notion of implementing functions using MUXes. While this is an antiquated approach to implementing functions directly, it can be both instructive and interesting and can often be worth browsing through. *This chapter is important because it provides the theoretical foundation for using and understanding map entered variables.*

Chapter 24:
This chapter represents the first foray into the notion of sequential circuits. Up until now, the text had dealt with combinatorial circuits only. This chapter describes the notion of "state" in a digital circuit with the concepts of memory elements including NAND and NOR latches and gated latches. This chapter describes these new concepts using digital design models that include state diagrams and PS/NS tables. *This chapter is important because it provides a low-level description of the most basic sequential circuits.*

Chapter 25:
This chapter provides the basic derivations and descriptions of the three common types of flip-flops: the D, T, and JK flip-flops. This chapter also shows how simple storage devices, such as flip-flops are modeled using VHDL including both synchronous and asynchronous flip-flop inputs. *This chapter is important because it describes the methods used to generate sequential circuits using VHDL models*.

<u>Standard Digital Circuits:</u> D, T, and JK flip-flops

Chapter 26:
This chapter outlines the basic techniques associated with the design and analysis of finite state machines (FSM). This approach to design is centered around the notion of theoretical FSMs in an effort to build skills and understanding of FSMs and is not overly practical in the notion of designing FSMs as controller devices. This chapter also introduces the notion of illegal-state recovery as a way to give FSMs the self-correction characteristic. *This chapter is important because it describes the basic procedures and theories regarding the design and analysis of finite state machines.*

Chapter 27:
This chapter provides most of the lower level details regarding the relationship between FSMs and timing diagrams. In particular, Mealy and Moore-type FSMs have their own particular issues regarding FSM timing; this chapter describes those issues. *This chapter is important because it introduces some of the major timing aspects associated with FSMs, particularly the differences between Mealy and Moore output timing.*

Chapter 28:
This chapter describes several methods of using VHDL to model FSMs. There are many approaches available to modeling VHDL; this chapter presented one that is considered the most straightforward for the beginning FSM modeler. This chapter also covers the notion of using enumeration types available in VHDL to encode the state variables associated with FSMs. In particular, this chapter describes the notion of one-hot encoding from an aspect of VHDL modeling of FSMs. *This chapter is important because it describes a straightforward approach to modeling FSM using VHDL and describes some of the methods used to encode state variables.*

Chapter 29:
This chapter provides the first FSM design experience involving something other than theoretical problems presented in previous chapters. This chapter begins with an intuitive description of state diagrams including the associated symbology and terminology. This chapter then introduces the FSM design aspects of the various flavors of sequence detectors. The design of sequence detectors using FSMs represents an introductory but meaningful first experience designing FSMs that actually do something that can be argue as useful. *This chapter is important because it describes the low level details of representing state diagrams and also the differences in timing diagrams associated with Mealy and Moore-type FSM.*

Chapter 30:
This chapter is primarily comprised of solved example problems including in-depth descriptions of the solutions. This chapter also provides a basic set of guidelines that can be used as a starting point to solving FSM-type problems. *This chapter is important because it describes techniques for solving actual design problems using FSMs.*

Chapter 31:
This chapter describes some of the more common aspects of clocking sequential circuits such as those implementing FSMs. This chapter starts by describing common clocking signals and associated terminology. This chapter also describes issues involved with non-idealized attributes of sequential circuits including setup and hold time and their relation the maximum clocking frequency attainable for a given circuit. *This chapter is important because it describes some of the more important timing aspects associated with sequential circuits and FSMs.*

Chapter 32:
This chapter describes a different approach to generating descriptive equations for FSMs implementations using D, T, and/or JK flip-flops. We refer to these techniques as "new" techniques and refer to previous FSM implementation approaches as the "classical" techniques. *This chapter is not that important; it's sort of interesting because provides some interesting information regarding FSM implementations and associated low level details.*

Chapter 33:
This chapter provides a basic description of simple registers, which is one of most common devices in digital circuits. *This chapter is important because registers and their variations are extremely useful and thus often found in just about all meaningful digital designs.*

Standard Digital Circuits: Registers

Chapter 34:
This chapter extends the notion of a simple register by describing other common flavors of registers including shift registers and counters. The discussion of shift registers includes barrel shifting, rotation and arithmetic shifts. The discussion of counters is primarily from a VHDL modeling level and includes descriptions of common counters such as up/down counters and decade counter. *This chapter is important because shift registers and counters are extremely useful in many areas of digital design, particularly in applications requiring fast arithmetic operations. These devices are basically extended feature simple registers.*

Standard Digital Circuits: Shift registers, counters

Chapter 35:
This chapter provides an introduction to computer architecture concepts; this introduction provides some background to the main chapter topic of arithmetic logic unit (ALU) design. This chapter describes ALU design on both a low level and high-level using VHDL modeling. *This chapter is important because it describes several approaches to designing ALUs. This description includes an introduction to the use of variables in VHDL.*

Chapter 99:
This chapter provides an overview and introduction to writing testbenches in VHDL. This introduction includes the example testbench models and methods of accessing test vectors used by the testbench. This chapter also contains an overview of other VHDL topic that are helpful in testbench writing such as more information on process statements and information on wait statements. *This chapter is important because it provides an overview and introduction to writing testbenches in VHDL. The VHDL language uses testbenches as a mechanism for verifying the proper operation of VHDL models using none other than other VHDL models.*

Disclaimer: This chapter has the high chapter number because the topics are presentable anywhere in this text (or not at all). Some of the material is somewhat advanced, but the initial portion of the chapter is digestible early in the traversal through this text.

Appendix
This chapter provides some random stuff that may be useful. This chapter has references and brief descriptions of many VHDL topics not covered in the main text. The end of this chapter has full VHDL models for most standard digital circuits.

1 Chapter One

(Bryan Mealy 2012 ©)

1.1 Introduction

You're at the point where for some reason you've opted to delve into this subject matter using Digital McLogic as a guide. This being the case, you need some introductory words of wisdom (or lack there of) regarding the approach taken by this text[1].

This text divides topics into small subject modules, which are creatively referred to as chapters. The intention here is to keep the subject matter as short as possible and bundled into relatively small readable portions. Let's face it, no one want to read long pages of technical drivel; people are simply more likely to read short pages of technical drivel.

There are actually lots of good reasons to keep things in bite-sized chapters, but the most important one could be that it makes organizing the text and assigning reading and/or problems from the text easier and more maintainable. As you'll see, there are some topics that are hard to group with other topics out there in digital-land, so these topics get a package of their own. Finally, few people really care about some of these topic, so you can safely skip these topics by skipping the entire chapter.

1.2 More Introduction

Each chapter has as many useful features as possible in order to help the reader spend less time with the text and more time understanding the subject matter. Each chapter generally has the following features:

- **Introduction**: quick motivating prose overview including a list of the main topics and the chapter and why that chapter is important (it if actually is) in digital design.
- **The Body of the Chapter**: In case you want the whole story (with example problems)
- **Chapter Summary**: The quick overview of chapter
- **Practice Problems**: Including both exercises and design problems for the reader's entertainment.

Main Chapter Topics

- **OVERVIEW OF TEACHING MODERN DIGITAL DESIGN:** Digital design has evolved faster than digital design courses could keep up with; a quick overview of the issues is helpful.
- **OVERVIEW TEXT ORGANIZATION:** This is text has an interesting form; knowing the underlying justifications for this form may be useful to you.
- **OVERVIEW OF APPROACH AND MOTIVATION:** This text takes a different approach than the historical approach to teaching digital design. This chapter describes the major differences between the historical approach and the approach taken in this text.

[1] Maybe not the best four-letter word to describe this text, but strangely adequate…

Why This Chapter is Important

This chapter is important because it provides a context for this text by describing the structure, style, and text content.

1.3 Digital Design: What is it?

This is somewhat tough to define this early in the game. The best I can say without generating too much confusion is the following:

Digital Design (early definition): the act of creating digital circuits to solve problems.

What makes this definition strange is the fact that you don't really know the full story behind the word "digital". You're probably familiar with the term circuit; but in this case, it specifically means some type of electronic circuit (currents, voltages, and all that)[2]. You have probably faced "problems" before; though in this context the word can mean just about anything. We'll explain more and expand this definition of digital design as you read along in this text, but this is a good working start.

1.4 Historical Overview of Digital Design Course

It was truly a different world when I was first introduced to digital logic (sometime in the mid-1800s). At that time, my digital design world revolved around the knowledge and topics presented in the course text. Back in those days, there was no laboratory associated with the digital design course. Because of this lack of academic sponsored hands-on experience, and the fact that the digital hardware involved was somewhat crude and the test/development equipment too costly for the average student, I could only rely on the course text to gather my digital knowledge[3]. Computers were expensive and not practically available to students, there was no internet, and software used to aid the digital designer either did not exist or was once again too expensive to be practical for the average digital design student.

In other words, both the knowledge and equipment associated with actual digital design was well outside the realm of the average student. Because of all of these factors, the digital design courses were severely limited and centered primarily upon the information presented in the course text (which, even to this day, still sucks badly). As a further consequence of a dearth of weak digital design texts out there, the instructors associated with digital design courses simply presented whatever was in the text, regardless of quality. Worst of all, any "designs" that were actually done were primarily "paper designs[4]". You could argue that this approach worked fine back then; but it simply doesn't work today[5].

[2] Don't worry about voltages and currents; the level of circuitry used in this text rarely deals with those details. The circuits designed in this text are described at relatively high levels of abstraction such that 99.9% of readers won't need to deal with the lower-level physics of the digital devices.
[3] The digital design instructors typically didn't know much about digital design either.
[4] A paper design was something you tried hard to convince someone else that it actually would work if it were actually implemented. The person you were trying to convince was often your instructor.
[5] Though the illusion of this actually working still exists at fine institutions such as Stanford

1.5 The Approach We'll Be Taking

Despite the fact that digital technology has advanced significantly in the recent past, the course texts associated with introductory digital design have remained in the dark ages of both engineering and educational technology. Despite these drawbacks, we've seen nothing but a steady increase in the price of introductory digital design textbooks accompanied by a steady decline in their quality[6]. As digital technology progressed (more computers, more hardware, more software, and more knowledge sources), more resources became available to both digital design instructors and students. These technological advances made both the purpose and content of digital design textbooks less absolutely defined.

Although the ultimate goal of transferring knowledge from the text to the student has remained the same, it is not exactly clear what knowledge is important and thus should be included in the text. What I do know is that the average digital design textbook contains way too much detail and thus can do no better than present the material in a manner that is sure to put even a die-hard caffeine junky into immediate slumber.

The problem with the typical introductory digital design text is that they are written from a standpoint of presenting digital concepts in a manner that supports generating questions for exams. In reality, the typical digital design text has painfully little to do with actual design; there are lots of topics to test you on, but very little actual design going on. Let's face it folks: actual design problems are harder to generate, harder to grade, and are thus rarely found on exams[7]. Simply stated, the current approach to digital design is outdated (though it was OK when there were no computers or software available). With all this in mind, it's a real mystery why the price of digital design textbooks has been rising[8].

The goal of this text is to present digital design in such a way as to gradually take the reader to digital design-land and make the reader into an actual digital designer. The underlying theme of this textbook is to travel lightly so that you can travel farther and faster. Topics that don't represent major steps toward the ultimate goal of becoming a viable digital designer are not covered or are covered only briefly. Not covering many of the less useful digital-like topics saves valuable time; this time is then free to develope true digital design skills[9].

I freely and openly acknowledge that advanced digital designers or instructors who read this text may feel that this text omits some topics and/or standard approaches to known topics. Once again, the goal of this text was to eject some of the less useful topics in favor of learning true digital design. With the knowledge contained in this text, you can easily pick up a standard digital design textbook and gather in all the full details without (if you so choose).

One of the many nice things about digital design is that the basics do not change much. This means that textbooks from twenty years ago contain many of the fine details of digital design. In addition, because publishers typically generate new versions of these textbooks in an effort to keep instructors from using

[6] The digital design texts published by professors at Cal Poly are known for their innate crappiness.

[7] Due primarily to the innate laziness of instructors.

[8] The truth is that digital design texts written by Cal Poly instructors are so crappy, the authors should be paying you to pretend like you're using them.

[9] I have proven this many times by asking some of the brightest intermediate and advanced digital design students about some of the topics taught in typical digital design courses. Although these students were currently well on their way to becoming great digital designers, they had completely forgotten the extraneous material presented in the typical introductory digital design course. Why? *Because they never had a need to use these concepts anywhere other than a quiz or exam in their introductory course.* Bummer!

old textbooks (which are significantly less expensive than new textbooks[10]), there are plenty of excellent older textbooks available from used book websites[11]. The prices are generally excellent[12], especially based on the amount of valuable and interesting information they provide. You should strongly consider purchasing at least one of these textbooks and using them in conjunction with the one you're currently pretending to read. Be forewarned that if you actually peruse a standard digital design book, you'll find that most of the book is filled with interesting stuff though not too much of the stuff is going to help you become a viable digital designer.

1.6 One Final Comment

Throughout this text, you'll be learning many tools and techniques associated with digital design and engineering in general. Once you've mastered these tools and techniques, there is a tendency to approach digital design problems by rote. Learning to do problems by rote is somewhat OK in early stages of learning digital design because of the newness of much of this material limits the complexity of problems. Doing problems by rote could arguably be a good approach if digital design was one of those engineering topics that carry little importance in the big scheme of things (we all know of such topics). As you'll soon find out, digital design material is much more important than most other engineering topics.

The relation that digital design has to computers in general is equivalent to the relation that addition has to general mathematics: it's tough to do math without understanding the basics of addition. The more completely you learn and understand the material presented in this text, the better off you'll be when you eventually take on more complex digital designs or be required to apply basic digital design concepts in later engineering and/or computer coursework. If you memorize the material, you eventually forget it. If you truly understand the material, you'll never forget it. The better you understand the material, the less work and struggle you'll experience when you're faced with an actual digital design sometime in the future. This means you'll have more time to actually have a life as opposed to spending time in a cave or in front of a computer (that is, if having a life is something you aspire to).

[10] This is not true at the typical college bookstore; their monopolies are so strong that used books generally cost more than new books. I could not write this if it were not true.
[11] Check out your local library, www.half.com, or www.addall.com for availability and/or pricing of these books. Many websites also include reviews of these books in order to help you narrow your selection.
[12] Seriously, a two-inch thick textbook for under $5!

Chapter Summary

- This text takes a unique approach to introducing digital design. Most digital design textbooks are mired in low-level details that most people either don't use or quickly forget. Moreover, most digital design textbooks have painfully little to do with actual design. This textbook is refreshingly different.

- Digital design entails the knowledge, understanding, and ability to design digital circuits. Doing endless pointless exercises in the back of chapters doesn't make you a digital designer.

- Digital design has come a long way through the last 25 years, much farther than all other engineering disciplines. This is due primarily to the onslaught of computers that started happening in the late 1970's and continues today.

- When all is said and done, go buy a used digital design textbook from a website that sells used books. The price is extremely low (for older editions) considering that they provide an impressive amount of information.

Chapter Exercises

1) List a few of the drawbacks of most digital textbooks.

2) List a few devices that you feel would contain digital circuitry of some sort.

3) Briefly describe why you feel most instructors seem to be lazy and/or pre-occupied with things other than teaching.

2 Chapter Two

(Bryan Mealy 2012 ©)

2.1 Introduction

Generally speaking, the first step in doing anything you don't know anything about is to learn about some of the terminology that's going to be tossed at you. We'll take this approach with the digital design thang by first getting some lingo and basic design approaches out of way before we go on. As with everything, digital design is easier than it initially appears. This chapter starts with defining what is "digital" and what is "design". You won't be doing any digital design in this chapter, but you'll learn about "digital" and will be doing some real design on the same level that you'll eventually be designing digital circuits on. The approach we'' take in learning "design" is substantial in that it is applicable to designing just about anything, including bowling paraphernalia.

Main Chapter Topics

- **ANALOG AND DIGITAL:** Understanding digital design requires an understanding of the inherent differences between "analog" and "digital". This chapter outlines these differences.
- **"MODELING" AS A DESIGN TOOL:** This chapter introduces the concept of modeling as the most basic tool for understanding just about anything, particularly digital design.

Why This Chapter is Important

This chapter is important because describes the notion "digital" and provides an overview of basic modeling concepts.

2.2 Analog Things and Digital Things

Since we've referred to the term "digital" quite often already, it's time to provide a definition for it. Generally speaking, in the context of engineering, the concept of "digital" is best understood when it is presented along side the definition of "analog". The terms are somewhat hard to describe with words but a few quick examples describes them nicely.

> **Example 1:** In an effort to save the planet and do the sustainability thing, I've been installing compact fluorescent (CF) lights as well as dimmers for my incandescent lights in my house. However, I've arrived at a dilemma: while the CF lights generally use less power, the intensity of light they provide is not adjustable. In other words, the CF light is either all the way *on*, or all the way *off*. But then again, while incandescent lights require more energy to operate (that is, they provide both light and a significant amount of heat), I have the ability to save energy by using the dimmer to adjust the light's output intensity

level. In other words, I am hypothetically able to adjust the dimmer to provide an infinite number of light intensity levels (but only one intensity level at a time). The on/off nature of the CF bulb is a hallmark of "digital" while the infinite number of intensity levels associated with the incandescent bulb controlled by a dimmer is the hallmark of "analog".

Example 2: Many buildings around campus have both wheelchair ramps and stairs leading to the buildings. While both options lead you to your ultimate destination, they do so in distinctly different ways. The wheelchair ramp can be considered a *continuous* path to the building, which means that you can hypothetically stop at any one of an infinite number of levels along this path to the building. The stairs, on the other hand, only have a few "discrete" levels I can stop at; the individual stairs represent each of these levels. The big difference here is discrete (for the stairs) vs. continuous (for the ramp). This example differs from the previous example in the instead of having two discrete levels for the CF bulb (on and off), we now have many discrete levels (one level for each of the stairs). The *discreteness* of things is the hallmark of digital while the *continuousness* of things is the hallmark of analog. What's really bugging me, though, is how to characterize an escalator? The answer: ***Digalog***.

Example 3: This is an example for the musically inclined out there. Stringed instruments created sound by way of a string that vibrates between two fixed points. On instruments such as guitars (or mandolins, bass guitars, etc.) and violins (or violas, cellos, fretless bass guitars) you change the pitch of the vibrating string by placing your fingers at different positions on the fingerboard. The difference between these instruments is that guitars have frets on the neck while violins do not (see Figure 2.1). The frets only allow the string to vibrate at a set of discrete string lengths (generally 19-22 on a typical guitar)[13]. The violin, on the other hand, has no frets, so you can effectively play an infinite number of pitches on a given string. In other words, the guitar provides a discrete number of pitches it can generate while the violin provides a continuous number of pitches.

Figure 2.1: "Frets" on the bass guitar fingerboard and the "fretless" viola fingerboard on the right.

[13] We're of course not considering using your fingers to stretch the string (which changes the frequency of the note).

The concept of analog vs. digital described in the previous examples is generally thought of as the describing something in the *continuous domain* vs. the *discrete domain*, respectively. For the problem of the lights, the incandescent light that is controlled by a dimmer can effectively provide anyone of a number of a virtually endless set of light intensities dependent upon the position of the dimmer control. This range of intensities is considered *continuous* over the basic ON (full light) or OFF (no light) settings of the dimmer. On the other hand, the CF light is only capable of providing two light intensities: ON or OFF. Since there are no in-between options for the CF light, the set of options is represented by two *discrete* values: ON and OFF. While the incandescent lights effectively provide an infinite range of light intensities, the CF lights provide only a finite set of options. The continuous realm of the incandescent light is then considered *analog* because of its continuous nature while the CF light is considered *digital* because they only operate at two discrete levels.

The importance of the analog vs. digital concept relates directly to digital logic design. Although there are many ways to implement a functional electronic circuit, the approach taken in this text is to implement circuits using digital logic. A digital logic circuit is an electronic circuit that provides you with some desired result and is implemented using digital logic devices. As you'll see later, the basis of all digital logic is the use of circuit elements whose inputs and outputs can only be one of two values. These two values are typically described as OFF-ON, TRUE-FALSE, HIGH-LOW, GOOD-BAD, BLACK-WHITE, EE-CSC, CAT-DOG, TEACHER-ADMINISTATOR, DEMOCRAT-REPUBLICAN, etc. Either a "high voltage" or "low voltage" drives the actual circuit but we generally choose to describe circuits using more general terms. Note that the "high" vs. "low" forms the *digitalness* of this approach. As you'll see later, once we get closer to performing actual design, we primarily model the inputs and outputs of our digital circuit using 1's and 0's, which are nothing more than placeholders for the actual high and low voltage values[14].

So why do I want to design digital circuits in order to solve my problems? The last time I looked, we're still all living in an analog world. The problem we face is that the ubiquitous computer is only capable of operating in the discrete, or digital, realm. Since a computer is nothing more than a giant digital circuit[15], understanding digital design is the unstated first step in successfully communicating with computers. Understanding digital design is also the first step towards designing computers. With each passing day, the world we purportedly live in becomes more and more controlled by digital devices. So long as you choose to live in this world, it's best that you understand digital devices and work with these devices at something other than a user level. The starting point for this journey is learning digital design[16].

Now that read through a few examples, some viable definitions for "digital" and "analog" are in order. These definitions are still somewhat lacking in that some other background details need covering.

Digital: A description of a something (such as a signal or data) that is expressed by a finite number of discrete values (or states). These discrete values include the entire "range" of possibilities, but do not include any of the "in-between" values.

Analog: A description of something that (such as a signal or data) that is expressed by a continuous range of values. The continuousness of analog implies that there are an infinite number of possible values in the given range.

[14] Some budding digital designers may be scared off by the notion of "voltage" so we generally discuss digital design at a level of abstraction that enables us to ignore the reality that "voltage" is the lifeforce of digital circuits.
[15] The fun parts of a computer are digital; there are many analog portions of a computer such as the power supply, but these are really boring when compared to the digital parts.
[16] And protecting yourself from robots.

2.3 The "Modeling" Approach to Anything and Everything

The text up to this point has been careful not to use the words "model" or "modeling" in order not to create confusion. The truth is that everything we do in digital design (and engineering in general) is a matter of generating the correct "model" for a something, namely the digital circuit. If we model our digital circuits correctly using a useful description mechanism (such as VHDL), we can apply software that will use this model to automatically generate a working circuit on a programmable logic device (PLD).

So, what exactly is a model? I could look up the term in the dictionary, but that may not give me the digital design flavor of the word. So listed below are my two best definitions of the word model. These are the most useful definitions that I can think of; a list of the many subtleties of these definitions surely follows.

model (def. 1): a representation of something.

model (def. 2): a description of something in terms that highlights the relevant information while hiding some of less useful information.

1. I could not clearly define the word with one definition, so I used two. The two definitions contain a different amount of wording, and thus detail. In other words, these two definitions provide two different levels of detail regarding the definition of a model, as the idea of differing levels of detail in a model is extremely important to the notion of modeling.

2. Modes are used to represent or describe something. What the definitions do not state is the exact nature of how we represent the model. This implies that there is no one correct form of a model. Generally speaking, anything that presents information by representing or describing something can therefore be considered a model. The other implication here is that some models are more "useful" than others.

3. There is no one "correct" model for anything. This implies that there can be many different "valid" models of the same thing. The different models can provide varying amounts and levels of information, but in the absolute sense, no one model is better than another.

Now that we've defined the crap out of the poor word, you may be asking yourself something like: "Why should I care?" The reason you should care is that models in digital design have only one general purpose: ***models transfer information to the entity reading the model***. In this case, the entity in question could be a piece of software, your lab partner, your teacher, or your pet raccoon. In addition, if you have created a good model, then your model will quickly promote an understanding of the thing you're modeling. If you've created a bad model, no one will know what you're attempting to convey (but you'll generate a significant amount of job security).

The concept of model should be nothing new to you. If you think about, out there in the real world, there are endless things out there are representing something without really being something (and some of these things actually transfer information to you). The truth is that the use of models is so useful that we somewhat forget that we're actually using or relying on them. The following list provides a few examples that may give you an idea of what you're missing.

1. **Example 1:** Runway Models – We've all seen them: emaciated men and women wearing bizarre clothing and sporting unique hairstyles strutting down the runway. These people are some designer's representation of actual women and men. These are actually really bad models (literally) because they don't represent anything other than an attempt to make people feel inadequate about the bodies so they'll consume more crap.

2. **Example 2:** Role Models – These are the people that society expects up to look up to. While we do know some features about these models (probably the good features, which is why they are role models), we do not have the full description. Unfortunately, we sometimes are disappointed when a more accurate description of our role models appears in the news and/or the police blotter.

3. **Example 3:** The Weather Report (weather prediction) – So how is it we know that it's going rain next week? The satellite images indicate there is a storm somewhere; but the interesting fact is that it the images predict that rain will arrive a week in advance. The prediction is based on models of previous weather patterns. There is nothing to stop the storm from drastically changing its path and not having rain the next week, but probabilistically speaking, we'll probably get rain next week.

4. **Example 4:** Graphical User Interfaces (GUIs) – Practically every computer-type device uses some type of GUI. These GUIs generally contain graphical representations of items such as button, switches, sliders, elevator bars, etc. These items are not really what they're trying to mimic, they're nothing more than models of the items. Some pixels on a display are used to model a button; the device interacts with the model to make something meaningful happen when the button model is actuated. There's not really a button there though there is a healthy dose of button-on-button action.

5. **Example 5:** Video Games – Not that I play video games… but the entire genre is a model of real and/or imaginary life. Everything you see in the game is a model of something you can relate to in real life, but it's truly far from being real life. Guns in real life are actually much louder and smell funny when you fire them. Imagine for a second if your video game modeled nothing related to real life? Can your brain actually deal with something that is has no concept of?

Digital design uses modeling to aid in the understanding and the design of digital circuits. The concept of models in digital design comes in two different broad flavors:

> **Design a digital circuit:** Unless you start grabbing transistors in order to create some actual digital devices, you'll probably be using models while you're in the act of "digital designing". In this case, you must generate your own "model" of a digital circuit that performs some specific task. The model you generate can then be used to create an actual digital circuit.
>
> **Here is a model of something; use it in your digital design:** In this case, someone is going to give you some model(s) and expect you to understand them to the point that you can use them in your digital design. Remember, someone is handing you a description of something, so you need to understand how to use it, but not necessarily how it works, before you are able to successfully use it.

The good news is that digital design generally only uses a few types of models. The list bellows describes the main model types used in digital design[17]. Keep in mind as you read these model types that there is no one correct model of any given thing. Either the model is useful because it helps you design or understand

[17] Not listed here are Finite State Machine state diagrams. We'll start using these special digital models in a later chapter.

something, or it's not useful because its inherent description is provided in such a way as to effectively provide you with nothing useful.

- **The black box model:** This is probably the most used and useful model in digital design. The black box model is simply a box that graphically shows the inputs and outputs to the digital circuit. We'll cover black box designs as they relate to digital design later. Figure 2.2(a) shows an example of a black box model used in digital design (don't worry about the details now).

- **The digital circuit element model:** These are nothing more than special black box models. The models used in digital devices typically use man special symbols; when digital designer see these symbols, they know what they mean. Sometimes these special symbols are replaced with black boxes having appropriate description labels. Figure 2.2(b) shows an example of a digital circuit element model and its corresponding black box model.

- **The timing diagram:** Timing diagrams graphically describe the operational characteristics of a digital circuit based on the status of inputs and outputs plotted as a function of time. We'll deal with timing diagrams in a later chapter. Figure 2.3 shows an example of a timing diagram for some unspecified digital circuit.

- **The written description model:** A model of a digital design or component is something nothing more than a written description. In this case, if there is not an accompanying black box model with the written description, you should be able to generate one based on the written description. Figure 2.4(b) shows a written description of a digital circuit.

- **The VHDL model:** VHDL provides a language of its own to describe the operation of digital circuits. Although VHDL has special syntax and constructs typically associated with computer programming languages, VHDL is a hardware description language, not a programming language. We'll start dealing with this more in later chapters. Figure 2.4(a) shows a VHDL model of a circuit.

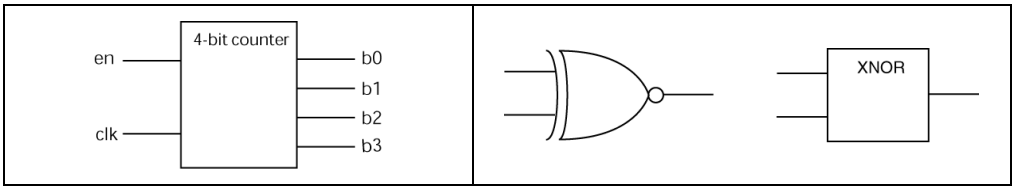


Figure 2.2: An example of a black box model (a), and a digital circuit element model with its corresponding black box model, (b).

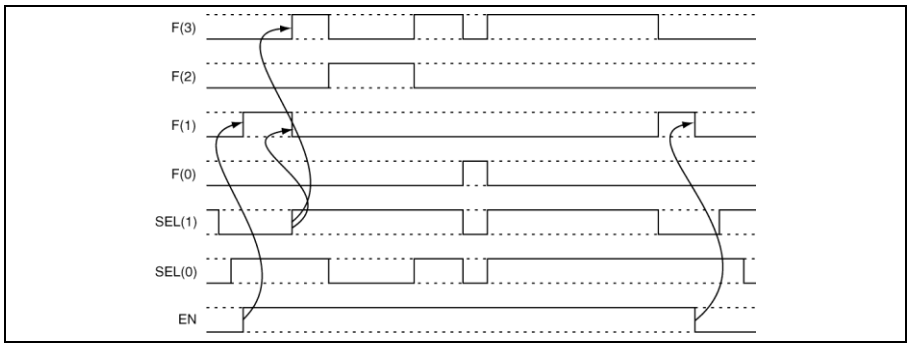


Figure 2.3: An example of a timing diagram (don't worry about the details).

```
entity dff is
   port (D,S,R : in std_logic;
           CLK : in std_logic;
         Q, nQ : out std_logic);
end dff;

architecture dff of dff is
begin
   process(D,S,R,CLK)
   begin
      if (R = '0') then
         Q <= '0';  nQ <= '1';
      elsif (S = '0') then
         Q <= '1';  nQ <= '0';
      elsif (rising_edge(CLK)) then
         Q <= D;
         nQ <= not D;
      end if;
   end process;
end dff;
```

The circuit has four inputs and two outputs. The outputs are always complements of each other. Two inputs, R and S, are asynchronous negative logic inputs. When R is asserted (negative logic), the Q output is '0'; when S is asserted, the output is '1'. The R input takes precedence over the S input. The Q output follows the D output on the active clock edge (rising-edge triggered).

(a) (b)

Figure 2.4: An example of a VHDL model (a), and a written description of a digital circuit (b). Incidentally, these two models described the same digital circuit.

The last general comment regarding these model types is that you should be able to generate all of these types of models for any circuit that you design. Your job as a digital designer is two-fold: 1) design a digital circuit, and 2) fully document your design. Using appropriate models help you both design and document your digital circuits. Models are your friends and they're going to help you in every aspect of your digital design career.

2.4 The Black Box Model in Digital Design

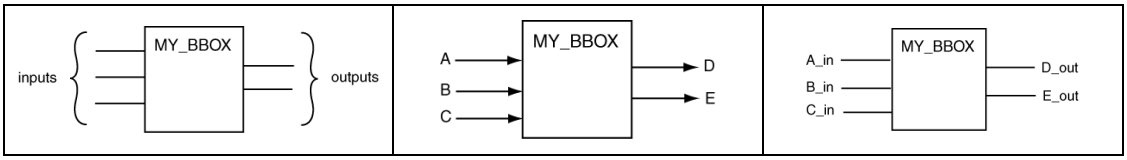
The black box model is extremely useful in designing anything, particularly digital circuits. There are approaches to designing digital circuits that don't use black box models, but these approaches can't compare to the efficacy of using black box modeling.

Unlike modeling techniques such as VHDL, black box modeling does not constrain you with special syntax or arcane rules. This being the case, you should not forget the overall purpose of using a model: models are quick ways to transfer information. The clearer the model represents its information, the more quickly you can glean information from it, and thus, the model is more effective. We'll soon demonstrate some of the basics of modeling as it relates to digital design.

In digital design, we're most concerned about the inputs to and the outputs from a digital circuit. Inputs and outputs represent digital signals going into and out of the circuit. The digital circuit itself is

represented by a box[18]; lines going into and out of the box represent the inputs and outputs, respectively. Figure 2.5 shows a few examples of a black box models.

Figure 2.5(a) shows the basic black box model with inputs and outputs listed on the left and right sides, respectively. Figure 2.5(b) shows an equivalent model with the inputs and outputs (I/O) indicated with the use of arrowheads on the signal lines. Figure 2.5(c) is another equivalent model that uses "self-commenting" signal names to differentiate the circuit's I/O. Note that in each of the models in Figure 2.5, the black box has a label. In addition, in case you have not figured it out yet, the word "black" in black box has a figurative meaning, not a literal one. In other words, the box is considered black because we don't know what's inside of it[19].



(a) (b) (c)

Figure 2.5: A few examples of basic black box models.

The models shown in Figure 2.5 are tough to write about because there are no hard rules for black box diagrams. However, here are a few strong guidelines you should always attempt to follow:

- The "flow" of digital models generally goes from left to right. Thus, inputs are on the left side of the box while outputs are placed on the right side of the box.
- Put arrowheads on you signals if it's not completely obvious what is an input and what is an output (and it usually isn't).
- All signals should generally be labeled unless there is some compelling reason not to (and there usually isn't).
- Place labels on boxes if the reason for the box's existence is not patently obvious (and it usually isn't).

The models shown in Figure 2.5 represent the first step in black box modeling. Let's take one more step and then you're on your own. One of the hallmarks of any type of design is the ability to abstract the design across many levels. For our particular purpose, a high-level model of something may not be that useful to use if we were hoping for a low-level model (and vice versa). These different levels of a model make up a hierarchy of a particular design with each level offering a different type and/or amount of information from other levels. Black box models are generally used to represent this hierarchy by essentially having a set of black box models for a given design. Keep in mind here that the goal is to transfer information: so strive to make your hierarchical models clear and concise[20]. What I'm trying to say here is that I have no hard rules for you to follow when you make your black box diagrams; therefore, I encourage you to make up your own rules. Whatever you do, your models will be judged by how well they transfer information between entities.

[18] Use any shape except circles; circles are used elsewhere in digital design.
[19] I'm thinking about using the term "dark box modeling" instead of black box modeling. Because the box is dark, there is no light shed on the interior of the box. Somewhat poetic, huh!
[20] This notion will be much clearer in later chapters when you're actually using more meaningful black box models.

Figure 2.6 shows an example containing two black box models. It just so happens that the larger model shown in Figure 2.7 uses these two models. Without being too judgmental, Figure 2.7 may not present the best models ever devised, but they are worth looking at. Figure 2.7(a) shows a black box diagram that sports a two-level hierarchy. The upper-level is the MY_BIG_BOX model; the lower level contains four previously defined models (these are the models shown in Figure 2.6). The model in Figure 2.7(b) is somewhat similar to the model shown in Figure 2.7(a), with a big difference as pointed out below.

- From Figure 2.7(a), we don't know precisely which signals are inputs and outputs by the way the model is drawn. Specifically, the higher-level model does not contain arrowheads on the signal nor do the signals contain self-commenting names.

- A black box named Z_BOX appears on the lower level and was not previously defined; what's in this box is therefore a total mystery and we'll hope it's defined elsewhere (which it's not).

- The interior black box diagram at the lower level is true to what Figure 2.6 shows. You can use this fact to extrapolate which signals are the inputs and outputs for most of the signals in the higher-level model. The I/O (input/output) characteristics of the Z_BOX remain a mystery.

- The two models in Figure 2.7 are almost identical. The model in Figure 2.7(b) seems to contain less information than the box in Figure 2.7(a) as it does not list the internal connections.

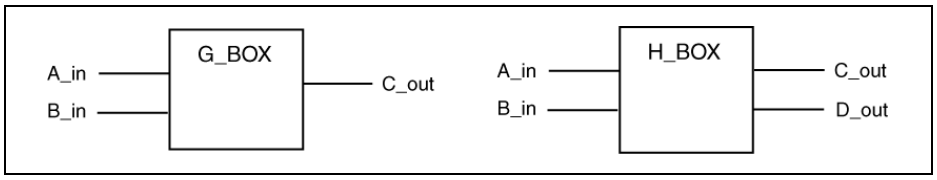
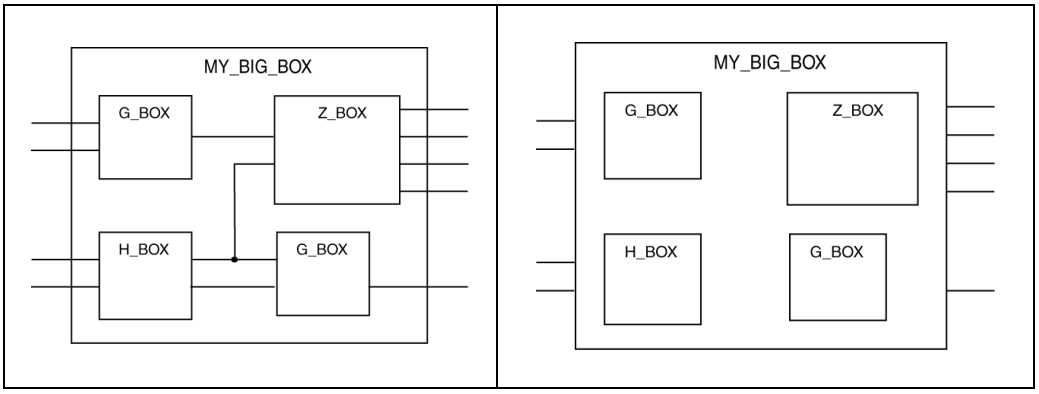


Figure 2.6: Two example black box models.



(a) (b)

Figure 2.7: Two examples of black box diagrams with similar features but varying levels of detail.

Here are a few examples to both drive home the black box modeling approach as a general approach to understanding things. This is actually the same example presented in three different flavors. Have fun.

Example 2-1

Provide a black box diagram showing a natural gas-powered storage-type water heater.

Solution: The first thing to notice about the problem is how vaguely it is stated. This is not necessarily a bad thing, particularly if you know nothing about hot water heaters. The problem is expecting you to simply do something; you probably won't be providing your solution to the Maytag company for immediate fabrication. In addition, you should definitely get used to the vagueness about how the problem was stated: bad descriptions are typical in most engineering pursuits[21].

The first step in all design problems should be to draw a box and place a somewhat meaningful label on it. Figure 2.8(a) shows the result of this complicated step. This step ain't much, but it is a great starting point, particularly if you have no idea of what you're doing. But then again, for anything you ever do in any engineering problem, drawing a block box should always be the first step.

The next step is to consider what little you know about the solution from the problem statement. You know the water heater heats water (duh!); therefore, there must be a cold water input as well as a hot water output. Once you include these in your model, your black box diagram should appear similar to Figure 2.8(b). Note that the arrowheads show the direction for the inputs and outputs.

For the last step, look again at the problem description. You know that the heater is a natural gas heater, so it must have an input for natural gas (so include that in your model). Figure 2.8(c) show the model when this input is included.

And that's about it. Are we done? Because the problem statement did not provide us with much direction as to the level of detail desired for the solution, we can thus declare ourselves done with this problem. Check it out: the model in Figure 2.8(c) is somewhat descriptive; especially considering you may know nothing about water heaters. The point here is that you started with nothing and you ended up with a reasonably helpful model. Tomorrow, the world.

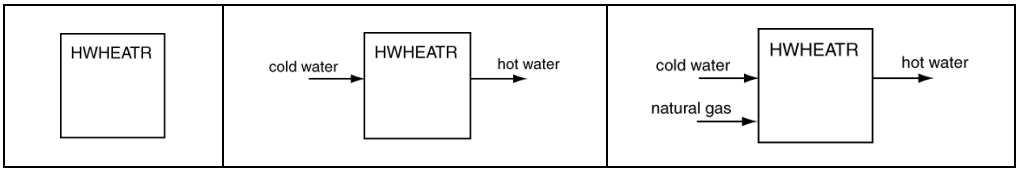


Figure 2.8: A possible thought process for this example.

Example 2-2

Provide a black box diagram showing a natural gas-powered storage-type water heater and some of its important subsystems.

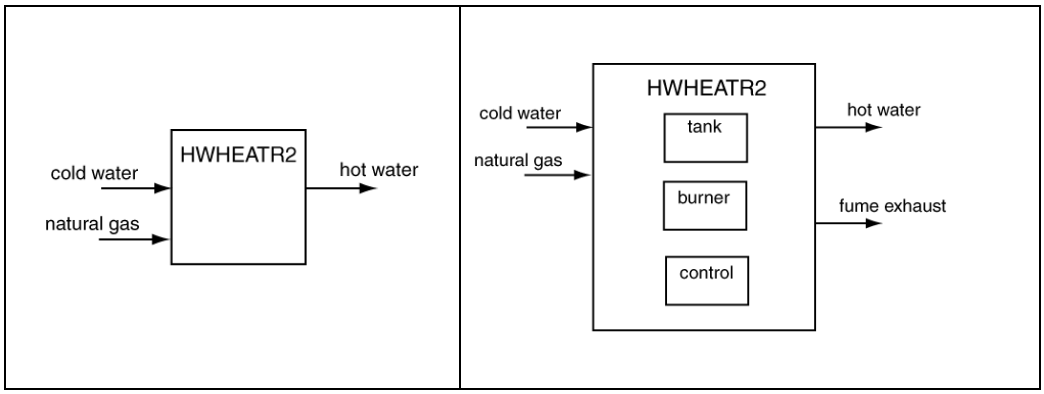
Solution: This is the same problem but now you're expected to know something about hot water heaters. The best approach to take here is not to panic. Without too much effort, you probably know more, or more importantly, you can figure out how a hot water heater works. Note that when this problem asks for subsystems, it is requesting that you do some type of hierarchical design.

[21] Particularly engineering education pursuits.

Step One is a hallmark of all design: don't reinvent the wheel: borrow you solution from the previous example since it was a rather cool hot water heater design. Figure 2.9(a) shows the result of this step (though the name of the black box has changed from the previous example).

Step Two, make a list of all the subsystems that would probably be required for the hot water heater. There must be a storage tank for the hot water. There must be a control unit[22] to maintain a relatively constant water temperature by turning on the gas when the water cools and turn it off when it reaches the desired temperature. There must be gas burner in there too. Come to think of it, there must be a fume exhaust (and I should have included this in the previous example had I thought of it then). I can't think of anything else. Figure 2.9(b) shows the final solution to this example.

Once again, we'll declare this problem completed. We could do more but… why bother since our solution has satisfied the original problem statement. Note that our final solution in Figure 2.9(b) show a two-level hierarchical support with the top-level being the HWHEATER2 black box that the lower levels being the three subsystems.



(a) (b)

Figure 2.9: A possible solution to this example.

Example 2-3

Provide a black box diagram showing a natural gas-powered storage-type water heater and some of its important subsystems. Include enough detail in your model to show the basic interaction of the various subsystems.

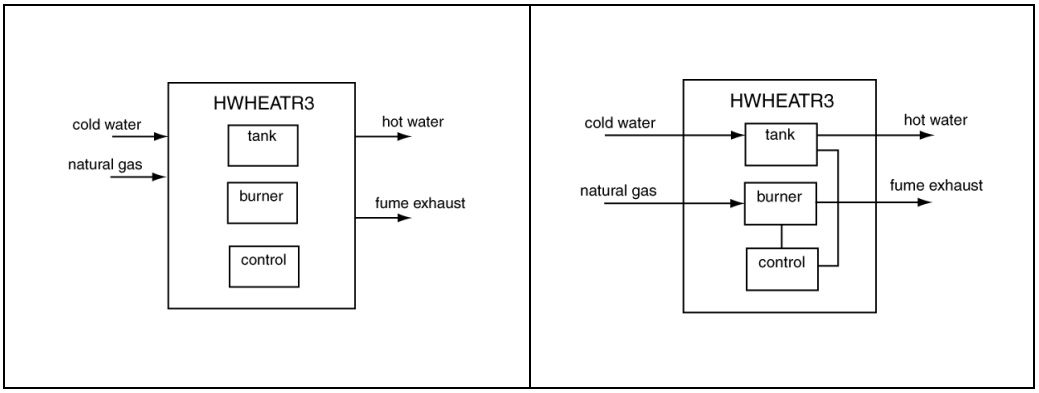
Solution: This example represents a slight modification to the original problem. Whereas in the previous example we were expected to know something about the heater's subsystems, we're now expected to know something about how the subsystems interact with each other. Once again, this is not that big of a deal, unless you're a total nimrod[23].

Once again, the problem didn't state exactly how much detail you need to include. Therefore, in this case, we'll add a small amount of detail and call the problem done. The first step in the solution is to once again borrow from the final solution from the previous example. Figure 2.10(a) show the result of this step with a new label being attached to the top-level black box. The next step is to add some connections

[22] Generally, a thermostat regulates the water temperature; that is, it keeps the water at some desired temperature without letting it get too much above or below that temperature.
[23] But now is your chance to learn something about the art of gas-powered hot water heater design.

between the internal black boxes. The control unit is the brains of the heater; it's going to turn on the burner when the water gets too cold. That means the control unit must monitor the temperature of the tank (one connection goes to the tank) and tell the burner to turn on/off (another connection goes to the burner).

At this point we could be done with the problem, but while we're at it, let's add some more detail. Since we have a few subsystems listed, let's connect the arrows from the higher level to the lower-level subsystems. This notion is fairly intuitive; Figure 2.10(b) shows the final result.



(a) (b)

Figure 2.10: A possible solution to this example.

This set of examples hopefully showed you the power of black box modeling. Although this example had nothing much to do with digital design, the hierarchical design approach used in these examples is the mainstay of viable digital design practice. There are two major things to note about this problem. As you read these, keep the thought of digital design in mind.

> Firstly, these examples only roughly stated the level of detail we should use in the problem solution. As a result, we did the best we could without worrying too much about the fact that the U.S. Patent office probably would not like our black box model. We did what the problem asked, then moved on.
>
> Secondly, while we were working these examples, we started out with nothing as well as very little knowledge about hot water heaters. By the time we were done with these examples, we had an interesting model of our hot water heater, and we're probably a little bit smarter. The black box modeling technique allowed us to take random bits of information and reassemble them in a viable model that seemed to solve the given problem. This is the cool thing about black box modeling: it provides you with a method of creating a path to the problem's solution when you're feeling like you have no idea where to go.

Mostly importantly, never forget Mealy's first and second laws of digital design.

Mealy's First Law of Digital Design: if in doubt, draw some black box diagrams.

Mealy's Second Law of Digital Design: if your digital design is running into weird obstacles that require kludgy solutions, toss out the design and start over from square one.

A result of Mealy's First law of digital design is if you have no idea what you're doing, you'll at least look like a pro[24]. But seriously, start drawing black box models and 1) list what you do know (such as inputs/outputs and given signal name, and 2) label everything (such as the names of the blocks). A result of Mealy's Second law of digital design is to prevent you from becoming stuck out there in digital-land. So if your design in not coming relatively easy, toss it out, rethink it, and start again, preferably from a slightly different angle. Recall that digital design should never be overly complicated.

2.5 Digital Design Overview

Even though we're only a few pages into the introductory verbage[25] of digital design, we're ready to grasp the main ideas behind modern digital design and relate them to the approach taken by this text. If you were somehow required to embody digital design in one short sentence, it would be something such as:

> **digital design:** the creation of digital circuits to solve problems.

The keys to this definition lie with "creating a digital circuit" and the "problem" that is "solved". These ideas are worth expanding upon.

> **Solving a Problem:** "Solving a problem" could mean many things but our approach is specific to digital circuits. The first step in solving a problem is to know something about the problem. Figure 2.11: shows the general model of a problem that we'll use as a starting point for all the problems presented in this text. In other words, although we won't initially know what goes in the box from a given problem description, the problem description generally tells us the "inputs" and "outputs" of the circuit as well as how the circuit will behave.
>
> **Creating Digital Circuits:** There are many ways to create a digital circuit and we'll soon be exploring a few of them. In order to solve the given problem, you'll need to create a digital circuit and place it (figuratively speaking) in the black box of Figure 2.11:. If your digital circuit manipulates the inputs in such as way as to provide the requested functionality on the outputs, then your digital design seemingly works properly. Probably the best way to view the digital circuits you'll be designing is that ***your circuit establishes a structured relationship between the circuit's inputs and outputs in such a way as to solve the given problem***. The problem statements in digital design generally state the desired outputs for a given a specific set of inputs.

In a nutshell, digital design is a matter of "creating" the interior of the Digital Circuit box shown in Figure 2.11: The digital circuit you design can be modeled as a device that generates the correct outputs to the circuit given a set of inputs[26]. There are many approaches to solving digital problems; using digital circuits to solve problems is the underlying theme of this text. Moreover, there are may approaches to designing digital circuits; this text will of course use the most intelligent approaches only.

[24] And that's good enough for the typical administrator in academia.
[25] Definition of verbage: part verbose, part garbage; pronounced *ver-baj*.
[26] In some later chapter, we'll modify this definition of a digital circuit as a device that generates the correct *sequence* of outputs to a specific *sequence* of inputs. The best is yet to come.

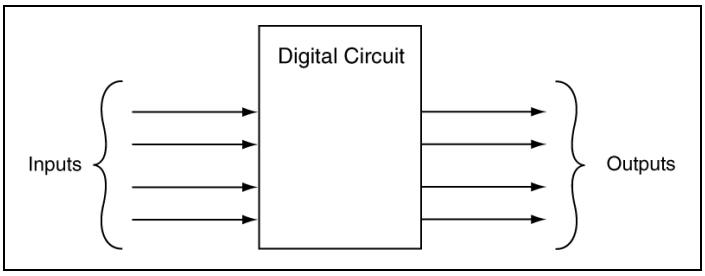


Figure 2.11: "Digital Design" in a nutshell: a general model of a digital circuit.

Chapter Summary

- The world can be divided into to camps: analog and digital. Though we live in an analog world, the computers that run this world are inherently digital. The basic characteristic of analog things is that they are "continuous" in nature while the basic characteristic of digital things is that they are "discrete" in nature. Said in other words, digital things can only take on a pre-determined set of values (thus the discreteness) while analog things can take on an infinite set of values.

- The notion of *digital* things in the context of "digital design" generally only takes on two discrete values. These values are most often associated with ON/OFF, HIGH/LOW, or TRUE/FALSE. Most of the time in digital design, these discrete values are described (or modeled) with "1" and "0".

- The notion of *digital* in "digital design" basically stems from the use of transistors. Being that transistors are a basic electronic element, the discrete values that generates the digital nature of digital design comes from high and low voltages associated with making the transistor operate. Since the exact voltage levels determine the physical characteristics of the devices, different digital devices use different voltage levels. Because of all these different voltage levels, the discrete values of transistors in the context of digital design modeled as either "1" (for high voltage) or "0" (for low voltage).

- The main tool used in any type of design is "modeling". In this context, a model represents a description of something, but not necessarily that thing. Modern digital design uses many types of models including black box models, VHDL models, timing diagrams, written descriptions, etc.

- The main purpose of models is to quickly transfer information to the entity (person or computer) reading the model. Since there are generally no carved-in-stone rules to modeling, the best models are the ones that transfer the most information; this means that good models are inherently clear to the user.

- Models in general promote an overall understanding of the entity being models and thus can become complex. The main mechanism in modeling to handle this complexity is the notion of "hierarchical modeling" which means that models can simultaneously describe many different levels of the design. The construct of "boxes within boxes" embodies hierarchical modeling as it relates to black box modeling.

- Black box modeling and hierarchical modeling is not limited to digital design; they can describe just about anything. In particular, black box models helps people reverse engineer just about anything and thus create knowledge where only darkness previously reigned.

- Digital design is about creating digital circuits to solve problems; problems solutions involve creating a circuit that establishes a structured relationship between the circuit's inputs and outputs in such a way as to solve the given problem.

- Most importantly to digital design are these two laws:

    - **Mealy's First Law of Digital Design:** if in doubt, draw some black box diagrams.

    - **Mealy's Second Law of Digital Design:** if your digital design is running into weird obstacles that require kludgy solutions, toss out the design and start over from square one.

Chapter Exercises

1) The analog world we live in has many people who seem to thrive on the use of digital photography. Practically everyone it has a digital camera, or has the equivalent on their cell phone or computer. A conversion from analog to digital occurs somewhere in the camera. Where exactly does this analog-to-digital (ADC) occur? Explain as best you can.

2) Briefly explain the general purpose for a model.

3) List some of the pros and cons of not having stringent rules regarding basic black box modeling techniques.

4) One of the themes of this chapter is the hierarchical design approach. Would it be possible to have too many levels for a given design? Explain your answer without being too verbose.

5) The dimmers used for incandescent lights mentioned in this chapter can actually be considered digital in nature. Although the dimmer effectively provides what a continuous range of light frequencies between the ON and OFF limit, how can it possibly still be digital in nature? Explain as best you can.

6) If you were required to take a "hierarchical" approach to reading this chapter, briefly describe how you would do it.

Design Problems

1) Draw a block box model of the following devices (be sure to label your model as completely as possible): a) the family dog, b) the tree growing in the forest, c) a bottle of beer, d) your best friend, e) your wallet or purse, f) a typical compost pile.

2) Draw a block box model of the following devices (be sure to label your model as completely as possible): a) microwave oven, b) handheld calculator, c) television, d) portable MP3 player, e) refridgerator/freezer.

3) Draw a two block diagrams, each using a different level of description, for the following devices (be sure to label your model as completely as possible): a) an internal combustion engine , b) a typical soda dispensing machine.

3 Chapter Three

(Bryan Mealy 2012 ©)

3.1 Introduction

We're at the point where for some reason you've opted to read this text. This is the point in most texts where high-level motivating words of motivation are offered; and so here they go… the first step in doing the digital design thing is to get some lingo and basic design approaches out of way. We've sort of already done that, so now we need to put our approach to digital design in its proper context.

Modern digital design is hierarchical in nature as was detailed in a previous chapter. But since this comment carries a lot of truth, we must clearly define where exactly in the digital design hierarchy our approach to digital design resides. This chapter attempts to provide that context. Additionally, there are many different approaches to designing a digital circuit; this chapter gives you a quick taste of those approaches, but you'll have to wait for later chapters for the complete details.

Probably the strongest statement you can make about digital design is that is has about zero relation to computer science. Though you can use digital design to design computers that are programmable by computer scientists (and other wankers), digital design and computer science are two different worlds. Despite the fact that a significant portion of digital design is done with VHDL, which appear similar to higher-level computer programming languages, the relation between the two is extremely thin. This chapter reiterates this point to the level of being nauseating.

Main Chapter Topics

- **DIGITAL DESIGN PARADIGM:** The modern approach to digital design is well structured. This chapter describes hierarchical design and its relation to digital design described in this text.
- **STANDARD DESIGN APPROACHES:** There are many approaches to "design"; this chapter describes some approaches typically associated with designing digital circuits.
- **MODERN DIGITAL DESIGN AND COMPUTER SCIENCE:** Modern digital design borrows many techniques and terminology from computer science; this chapter briefly discusses the more useful similarities

Why This Chapter is Important

This chapter is important because it provides basic information regarding digital design including the basic digital design paradigm and levels of abstraction. These concepts help this text's approach to digital design in an appropriate context.

3.2 The Digital Design Paradigm

As you study digital design, you'll soon discover that digital circuits quickly become complex and complicated (or, complexicated, as some idiots like to say). The good thing about digital design is that it contains a built-in mechanism that indirectly controls the complexity of digital designs and facilitates the understanding of complex digital circuits.

Because you can view a digital circuit at many different levels, you can also design a digital circuit at many different levels. We've seen this notion before in a previous chapter with our discussion of modeling. Many factors decide the level at which you choose to design your digital circuits. Generally speaking, you'll be doing your design at the highest level of abstraction possible in order to increase your effectiveness as a digital designer. Once again, there are many different ways to model a digital circuit and these ways are generally divided into different "levels of abstraction".

The field of computer science often uses the term "abstraction" despite the fact that the typical computer scientist has no idea what the word truly means. The dictionary definition goes something like: *the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances*. This definition relates to digital design in that we typically attach a set of qualities to a "black box" with little regard to the actual implementation details of those qualities. Often times it is your job to generate a set of qualities of a digital circuit; but once you do this, you want to move on by moving to a higher level of abstraction by placing your design into a black box. Other times, you'll be using the black boxes designed by other people. This black box approach and moving to higher levels of abstraction simplifies digital circuit design and thus make you a more efficient digital designer.

In digital design, the concept of levels of abstraction are often referred to as the black box approach to digital design or sometimes as the "object-level" design approach. No matter how it is referenced, it is a form of *hierarchical design*, a notion that we'll live and die with in digital design. The previous chapter contained a few examples of hierarchical design; later chapters provide more details regarding these notions. In reality, any digital design that does anything remotely useful is going to be a hierarchical design. If your design is not hierarchical, then your design is a "flat design". If you're going to do flat digital designs, you might as well hold up a flashing neon sign saying: "**I'm dumbtarted**"[27].

In reality, it is possible to perform digital design at many different levels. This text, as well as digital design in general, only touches upon a narrow window the digital design hierarchy[28]. The following set of figures describes a few different levels of abstraction associated with digital design. Have no fear, however, you won't need to know much about most of these levels other than the fact that they exist.

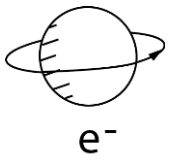


Figure 3.1: This is a cheap model of an electron. Technically speaking, it's the movement of electrons that make operation of digital circuits possible, as it's the electrons moving around in a controlled manner that make useful things happen in electronic circuits. This is an extremely low-level view of digital electronics and is way out of the scope of this text.

[27] As strange as this seems, there was an instructor at a certain California State College who insisted that students do nothing other than flat designs. By chance if that instructor is reading this, I've been meaning to tell you that you're a total idiot.

[28] As you may or may not eventually see, the typical digital design text goes into more detail regarding the underlying transistors used to make digital devices. It's all fun stuff, but only if you're into the physical properties of semiconductors. It is important stuff, but, it is better to learn all that at another time if you really need to learn it.

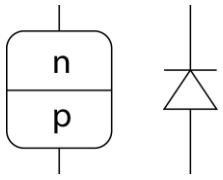


Figure 3.2: The basis of digital electronics is what the electrons are doing in a piece of semi-conducting material (namely sillycone). This figure shows two different flavors of silicone that are sandwiched together to form a device known as a diode. The flavors, known as n-type and p-type, are generated by doping the silicone with special substances. Sounds fun, huh? Why do you think they call it dope? The symbols in this figure are the accepted circuit schematic symbols (they're models, of course) for the diode. Once again, this level of view of a digital circuit is still too low-level for this text. It's interesting to note that many engineering students will eventually need to deal with diodes in those pesky analog-type courses.

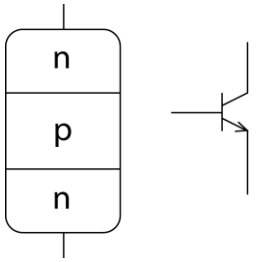


Figure 3.3: This device is somewhat similar in construction to the diode but there's more of it. This device is a model of a Bipolar Junction Transistor (BJT). The interesting thing about this device is that it's operating characteristics forms the basis of digital electronics as the notion of *digital* in "digital design" is derived from this device. This transistor is able to operate in several different modes; in digital electronics, these transistors operate exclusively in only two modes which is where the term "digital" comes from and are typically referred to as OFF-ON, HIGH-LOW, etc. Once again, we're purposely ignoring the terms "voltage" here. This device is still too low-level for this text.

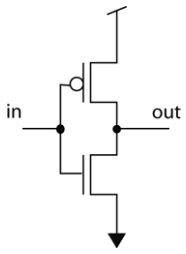


Figure 3.4: This symbol is comprised of two transistors: a PMOS and an NMOS. Together they form what is known as a CMOS element (Complimentary Metal Oxide Semiconductor). The connect of these two transistors is such that they form an element that performs a basic digital function (inversion). Note that we no longer include the "p" and "n" level of modeling in this figure However, since this text does not deal with basic analog electronic devices, we still want to avoid having any direct dealings with transistors.

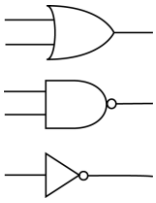


Figure 3.5: This figure shows some of the gates used represent transistor circuits such as those in **Figure 3.4**. These gates perform specific and useful digital operations that you'll learn about later. This text deals directly with gates such as the ones in this figure; you'll soon be learning how to design useful digital circuits using these gates. Digital circuits modeled using these devices are referred to as gate-level circuits or gate-level designs. You could just as easily model the circuit on a transistor level, but using the gates are easier to understand because we prefer to design on the highest level of abstraction possible. In other words, each of the gates in this figure is actually an abstracted model of a transistor circuit.

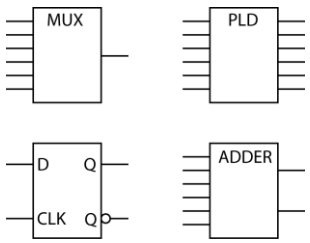


Figure 3.6: Up to this point, we've been abstracting to higher and higher levels relative to digital electronics. The trend continues with this figure. The devices in this figure are models of standard digital circuits, which are generally constructed of the gates shown in **Figure 3.5**. This text also deals directly with this level of digital circuit. These and similar devices, combined with the gates of **Figure 3.5**, are used to construct some useful and interesting digital circuits. I like to refer to digital circuits drawn using these devices as object-level circuits because of the nice analogy made to object-oriented software design. Lucky for us that VHDL strongly supports object-level design.

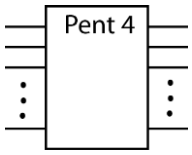


Figure 3.7: And lastly, abstracting to really high levels are digital devices such as microprocessors and microcontrollers. Viewed from this level, these devices are massively complex despite the fact they are comprised of digital devices that are relatively simple to understand. Devices as complex as this one are beyond the scope of this text. Keep in mind that devices such as these are so complexicated (such as millions of transistors) that they necessarily must be designed at very high levels of abstraction.

Although this text only deals with embodies the abstraction levels shown in Figure 3.5 and Figure 3.6, the series of figures in embodies a major theme behind digital design: abstracting to higher levels in order to increase your understanding and effectiveness as a digital circuit designer. State differently, there is always an effort to group a bunch of small things of varying purpose into a special box (sounds like modeling to me). These special boxes perform specific functions though the details of how the special box performs those functions are not necessarily important at that specific box level.

3.3 Digital Design and the Black-Box Diagram

As you'll see later, drawing a black box model (or black box diagram) is generally the first step in any digital design problem. This approach supports the modern notion of digital circuit design in that your design will probably include many black boxes designed by someone else or designed by you at some previous time.

You should not consider the black box diagram approach or the hierarchical design approach to problem solving anything new; it's actually similar to doing many simple everyday tasks such as using a calculator. Though you have (or should have) the ability to do many of the tasks you relegate to your calculator, you'd simply rather not do them and opt to use the calculator for its speed and accuracy. For example, you can use a simple algorithm to perform long division and eventually arrive at the same result as a division operation on a calculator. However, the calculator does it faster and without error: the wisest decision is to use the calculator. The same is true for most of the digital design problems. You'll learn how to design certain standard devices and then abstract them to a black box in order to not worry about the lower-level details. You'll forever use only that black box without worrying about details of what's inside because you have confidence that it works. Similar to the calculator example, you'll have the ability to do the lower-level black box design but you avoid it because you've done it before, you understand it, and you have better things to do.

To drive the point home even further, most every existing digital design tool out there has an extensive list of black-box digital devices. The good new is that modern digital design rarely expects you to design

digital components from the ground up[29]. If you had to design everything starting from their basis low-level digital parts, you wouldn't be a productive digital designer. Besides, when you're out there in the real world, no one is going to pay you the big bucks to designing circuits and components that have already been designed and tested. Digital designers of yesteryear were masters of paper designs since that was about all they could do[30]. The modern digital designer needs to be a master of the tools; embedded in those tools are previously designed devices that are ready for re-use in your design.

3.4 The Top-Down and Bottom-Up Design Approaches

Drawing a black box model (or black box diagram) is generally the first step in any digital design problem. Because of this, it is pertinent to discuss some well-known attributes of black box modeling. Related to the notion of black box modeling are the common notions of *top-down* and *bottom-up* design paradigms. These are labels attached to design approaches that you'll find yourself applying without much effort. The interesting thing to note about both of these design approaches is that they are inherently hierarchical in nature despite the fact that this quality is not included in the name. These design approaches are worth defining and briefly discussing as they come up often in various discussions regarding design paradigms in other fields.

The top-down design approach starts with drawing a box that represents the highly abstracted design (a design at its highest level). This would entail a labeled black box with a listing of inputs and outputs. The understanding with approach to design is that you'll be filling in lower-level details as your design progresses. In this case, including lower-level details typically means drawing black boxes within the highest-level black box. Once again, note the hierarchical nature of this approach.

The bottom-up approach to design starts by drawing boxes at their lowest level of abstraction. This means that these low-level boxes necessarily don't include boxes in their interior. This approach essentially maps out the fine details first and then works upwards to higher levels of abstraction that show how these low-level boxes relate to the higher-levels of abstraction. Once again, note the hierarchical qualities of this approach.

So what is better: bottoms up or top-down? There is no correct answer to this question. Your mission is to design a quality and working digital circuit: exactly how you do this is up to your own personal style. Don't let anyone tell you one approach is better than another[31].

3.5 Structured Digital Design: An Interesting Concept

As you will soon find out, the reality in digital-land is there are only a relative few number of core digital devices. Even the most complex digital circuit is decomposable into a set of these core digital devices. Computer programs refer to this decomposition process as "structured programming"[32]. This decomposition is a reversing of the hierarchical design process. If you are able to understand the operation of the core digital devices, you'll also be able to understand any digital device, regardless of its complexity, if you have the inclination to actually dig that deep into low-level details. More likely than not, if your digital design can be decomposed into these basic elements, your design will most likely be robust, easy to test, and easy to reuse with confidence.

[29] Actually, designing everything from square zero is that hallmark of a really bad digital design course; try not to find yourself there.

[30] This is particularly true in an academic environment were the quarters and semesters pass too quickly to spend time actually implementing real digital circuits.

[31] This is a typical interview question. My suggestion is that however you answer such a question, be sure to support your answer because it's inevitable that the interviewer has a different approach.

[32] And if I recall my computer science knowledge correctly, the notion of a good computer program is one that can be decomposed in to a relatively small set of basic programming structures. If you're computer program is not written correctly, no such decomposition is possible.

In this text, there are only a few digital design modules that we'll be learning about and working with. From these modules, you can design an amazing number of different circuits. Add in the many digital modules contained in typical VHDL design libraries, and you can design any digital circuit without too much effort. The notion of black box designs or object-level design fully supports the presence of large design libraries full of digital devices waiting to be used by a crafty digital designer. VHDL is a tool that allows you easy access to these device libraries.

Figure 3.8 shows a quick overview of digital design as it relates to the introductory digital design topics covered in this text. You don't need to know any of this now, but what you'll hopefully see from Figure 3.8 is that there aren't that many standard digital devices (or modules) out there. Even better is that fact that the basic modules out there are generally simple devices. The key here is that there are not many low-level devices to learn about; the majority of digital design involves the assembly of these basic devices into a larger, more meaningful circuit.

In summary, here's all I know about digital design:

1) Digital design is based on a relatively small set of digital devices.

2) Digital design relies heavily on various modeling approaches.

3) Digital design modeling relies heavily on hierarchical modeling.

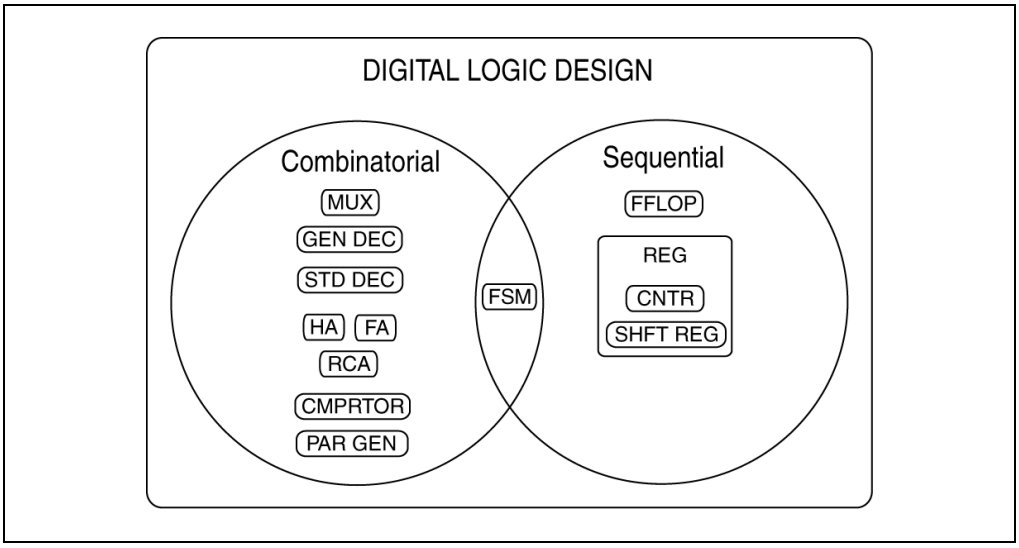


Figure 3.8: The quick digital design overview.

3.6 Computer Science vs. Electrical Engineering

Without doubt, the object oriented approach to software design is analogous to the modern Electrical Engineering approach to digital design. The similarities are wickedly similar and worth mentioning here.

- In computer programming, you have the ability to create programs at many levels such as machine code, assembly language, or higher-level languages such as C or Java. In digital design, you can design at the transistor level, the gate level, the object level, etc.

In both cases, the lower the level you design at, generally speaking, the more time and effort you'll need to put into it.[33]

- In computer science, reusing previously designed and tested code is a good way to increase your productivity. A significant portion of computer program design involves the incorporation of previously designed modules in the current program. As a result, there are many software packages out there that simply provide bunches of modules (functions, methods, subroutines, etc) that do something meaningful. Not surprisingly, much of the productivity software out there is written a very high level[34]. In the modern electrical engineering-based approach to digital design, you have many similar options such as off-the-shelf digital devices and the ability to create and use various design libraries. This distinction between computer science and modern electrical engineering approach to digital design is blurred further by the existence of hardware design languages such as VHDL and Verilog and their associated device libraries.

The point here is that these two fields are surprisingly similar. They also have the interesting relationship that one would not exist without the other despite being healthfully dissimilar. Good engineers are fluent and productive in both electrical engineering and computer science (while bad engineers tend to exhibit hatred towards one or the other or both). Start on the road of being a good engineer by not fearing or avoiding the field that is "not part of your major" or "outside your area of interest". Always put effort into celebrating and working with the interconnectedness of these two fields, particularly the areas of digital design and computer programming.

One major consideration that primarily computer science people should consider is the fact that a great deal of time and effort in the real world go into creating Electronic Design Automation (EDA) tools. These are software tools that increase the productivity of hardware designers. This implies that if your primary interest is software development, having a solid understanding of the needs and methods of the digital hardware design engineer is going to make you a better and/or more marketable software developer. In other words, if it was not for major advances in software tools, digital hardware designers would still be wiring boards to circuits and cutting out rubylith[35].

[33] There are of course instances in both fields where you would definitely want to choose to design on a lower level as opposed to a higher one.
[34] Could you imagine what a nightmare it would be to write Windows application code at a low-level such as assembly language?
[35] Go check this out on www.wikipedia.org…

Chapter Summary

- Digital design is about making complex things form simple components. Moreover, digital design is based on a relatively small set of simple digital devices, relies heavily on various modeling techniques, and particularly relies on hierarchical modeling

- The basic of "digital" circuits is the transistor. Transistors can be operated in many different ways, but they only operate at two discrete levels in digital design: OFF/ON, HIGH/LOW, etc.

- The main drive in digital design is to group a bunch of small things of varying purpose into a special box (namely a black box) for use later at higher levels of abstraction. These boxes are known to perform specific functions though the details of how the special box performs those functions are not necessarily important higher levels of abstraction.

- Good digital designers know their tools and know the basic approached to modeling. Two basic approaches to modeling are the top-down approach and the bottom-up approach; one is not necessarily better than the other. There are libraries full of previously designed digital devices that can be quickly placed into your design. Having knowledge of your development software generally allows you access to these devices.

- Good digital design borrows many techniques and practices from computer science. Modern digital design is typically object oriented; in this case, objects are black-boxes.

- The real important aspects of digital design:

    - Digital design is based on a relatively small set of digital devices.

    - Digital design relies heavily on various modeling approaches.

    - Digital design modeling relies heavily on hierarchical modeling.

Chapter Exercises

1) The analog world we live in has many people who seem to thrive on the use of digital photography. Practically everyone it seems has a digital camera, or has the equivalent on their cell phone or computer. A conversion from analog to digital occurs somewhere in the camera. Where exactly does this analog-to-digital (ADC) occur? Explain as best you can.

2) One of the themes of this chapter is the hierarchical design approach. Would it be possible to have too many levels for a given design? Explain your answer without being too verbose.

3) List a few instances where a "bottom-up" design would be better than a "top-down" design.

4) List a few instances where a "top-down" design would be better than a "bottom-up" design.

5) List a few of the disadvantages of using a "flat-design" approach to designing anything.

6) Briefly describe the similarities between object oriented computer program design and modern digital design. Use as many cool words in your description as humanly possible.

7) Consider a team of programmers who are developing some EDA tools targeted for quickly creating efficient digital circuits. Speculate on the most likely breakdown of fields of study of the team members; be sure to include the team managers in your speculation.

Design Problems

1) Using a top-down design approach, show the steps required and the resulting black box diagram to design a laser-type printer for a personal computer. Don't worry about if you know very little about computer printers; I never allow not know anything stop me from doing stuff.

2) Using a bottom-up design approach, show the steps required and the resulting black box diagram to design a soda vending machine. Do your best to show the process; don't worry about not knowing much about vending machines.

4 Chapter Four

(Bryan Mealy 2012 ©)

4.1 Introduction

The previous chapters hopefully gave you a small taste for what exactly is meant by the term "digital" and the term "model". In this chapter, we'll continue our move towards digital design by discussing some of the underlying details regarding number systems. This information in this chapter represents a brief introduction to number systems; we'll address the topic again in later chapters.

In addition to your "welcome" to number systems in the context of engineering, this chapter also introduces the details of engineering notation. The sad reality is that anyone can use any number of ways to write relatively large and relatively small numbers. While all of these methods may show equivalent numbers, the numbers can appear quite different, thus leading to massive confusion and world chaos. The better approach is to employ some type of standard when representing numbers; the standard we'll use in this text is the standard used by most intelligent people[36]. Engineering notation is the standard we speak of.

Main Chapter Topics

- **ENGINEERING NOTATION:** Writing number in a clear and concise manner is rather important in engineering and thus, digital design. This chapter describes the approach and motivation behind engineering notation.
- **NUMBER SYSTEM INTRODUCTION:** Since number usage has become second nature in our everyday existence, we probably have forgotten some of the underlying characteristics that make numbers "work". This chapter provides a friendly reminder of common definitions associated with number systems as well as a brief introduction to binary numbers.

Why This Chapter is Important

This chapter is important because it provides a description of engineering notation and the basic form of numbers. This chapter also introduces the basic concepts of working with binary number representations.

4.2 Engineering Notation

As you are probably finding out by now, digital designers and engineers are lazy[37]. In order to reduce their workload and thought-load, engineers typically use what is referred to as *engineering notation*

[36] Thus, academic administrators have no such standards. For that matter, neither does the average troglodyte.
[37] They're actually constructively lazy: always searching for a "better" way to do things (which is a really good thing).

when representing numbers out there in engineering land. Unfortunately, problems can arise when attempting to represent a numbers. For example, you can represent the number 34.7×10^{-4} in an infinite number of equivalent ways; Table 4.1 lists a few of the valid representations.

$0.000034.7 \times 10^{2}$	0.347×10^{-2}
$0.00034.7 \times 10^{1}$	3.47×10^{-3}
0.00347	34.7×10^{-4}
0.0347×10^{-1}	347×10^{-5}

Table 4.1: A few ways to represent 34.7×10^{-4}.

The problem is that it's hard to obtain a good intuitive feel for numbers if they are written in different forms. As you'll find out, having a "good intuitive feel" for a problem helps you arrive more quickly at a solution. The solution to this problem is to adopt a standard for representing numbers; engineering notation is the standard used in the engineering profession, typically in the area of digital design. Engineering notation is a subset of scientific notation with some extra rules added. The idea of behind engineering notation is to enhance the intuitive feel of numbers by placing restrictions on their representations.

The main item of interest here is the difference between numbers represented in engineering notation and scientific notation[38]. Engineering notation uses special suffixes to represent the exponential portion of the number; using these prefixes provides the viewer with a quick feel for the number. We all like the notion of a "quick feel". The advantages of using engineering notation are that it allows you to get a quick feel for the magnitude of numbers based on the designated unit prefix as well as the magnitude portion. Figure 4.1 shows the rules for using engineering in every day life.

1. The magnitude portion of the number should be between 0 and 1000. This range can be officially listed as [1,1000)[39].

2. The units portion of the number uses an appropriate prefix. Engineering notation does not use exponential notation The constraint is that all the exponents must be integral multiples of three.

Figure 4.1: The rules for correctly using engineering notation.

Table 4.2 lists the only prefixes you need to know. There are many others, but how often do you have the unsatisfiable urge to use prefixes such as "yocto"[40]. You should be familiar with most of these prefixes already; but if not, now is your chance to learn some lingo that will impress your non-technical friends[41] and allow you to freely converse with your technical friends[42] (real or imaginary). Note that the prefixes in engineering notation only come in multiples of three. You could make up your own prefixes if you wanted but that would pretty much guarantee that no one would know what the **$%#&**!

[38] Look these up for full details; this section provides only an executive summary of sorts.
[39] This notation means that the number is greater than or equal to 1 but less than 1000.
[40] Yep, it sounds more like a personal hygiene problem than a prefix.
[41] If you actually have any friends.
[42] Real or imaginary, if you actually have any friends.

you were talking about[43]. So unless instructed otherwise, engineering notation always uses these prefixes.

Value	Prefix	Abbrev.	Example
10^9	Giga	G	GHz
10^6	Mega	M	MHz
10^3	Kilo	k	kHz
10^{-3}	mili	m	ms
10^{-6}	micro	μ	μs
10^{-9}	nano	n	ns

Table 4.2: Engineering Notation prefixes.

Example 4-1

Represent the value 452300Hz in engineering notation.

Solution: The value 452300 is greater than 1000 (10^3) but less than 1000000 (10^6). This means we'll need to use the K prefix. The given number is then divided by 1000 to obtain the proper magnitude portion of the number and the K prefix is attached. The final answer is 452.3 KHz.

Example 4-2

Represent the value 84.3×10^{-8} s in engineering notation.

Solution: The first order of business here is to convert the exponential portion of this value to a multiple of three. If we multiple the number by 100 (10^2**)** the exponential portion of the number becomes -6 which is OK. However, to compensate for this multiplication, we must also divide the magnitude portion of the number by 100 (10^2**)**. The resulting magnitude value is then 0.843. However, since this value is less than 1, this will not be proper engineering notation. Our only other choice is to adjust the exponential part in the other direction. To do this we divide the exponential portion of the number by 10 to obtain 10^{-9} and then multiply the magnitude portion of the number by 10 in order to compensate. The result is 843ns.

4.3 Number System Basics

Without doubt, humans have spent most of their existence doing quite well without the concept of numbers or number systems. Number systems eventually became an integral part of human life as

[43] Then again, such an approach works great for academic administrators.

humans evolved and progressed[44]. The concept of numbers, for better or worse, has corrected the basic limitation of the human brain in its lack of ability to handle large quantities of "things".

My eighth grade algebra teacher[45] once told the class a story about some primitive culture. I've long since forgotten why exactly he told this story, but I never forgot the story itself; after all, that was the day I found out that I was not much better than a caveman. He told the class about a primitive culture somewhere in the world and about the number system they used. This number system was comprised of three "numbers": *one*, *two*, and *many*. What has always impressed me about this story was the fact that it still nicely describes the way my brain "processes" certain type of situations where keeping track of a certain number of items is necessary. Although this number system seems extremely limited compared to the number systems we currently use, this caveman simple number system remains well matched to limitations of the human brain.

Figure 4.2 demonstrates a basic limitation in the human brain. In Figure 4.2(a), it's fairly obvious to the unencumbered human brain that there is one dot in the square. Your brain can hopefully both see and process this information almost instantaneously[46]. Your brain probably has no problem "counting" the number of dots in the square of Figure 4.2(b) either. However, once you arrive at Figure 4.2(c), your brain cannot instantaneously gather this information: the sheer number of dots in the square instantly overloads your brain (even though there are relatively few). In essence, your brain is no more sophisticated than the brain of the person in the so-called primitive culture.

As you know, we modern humans are able to both conceive and process the dots in the square of Figure 4.2(c). The way we do this is to represent the quantity of dots in the square with a "number". This number is defined by a previously and mutually agreed upon set of rules to ensure that everyone who is processing the quantity of dots in the square arrives at the same result. There is even a mutually agreed upon set of squiggles that are used to represent the numbers.

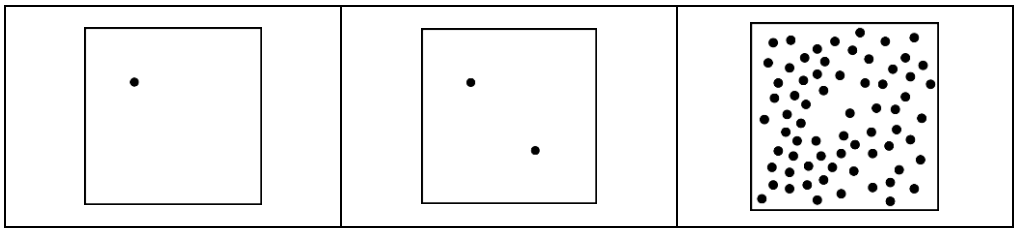


Figure 4.2: An example showing a basic limitation of the human brain.

Also worth mentioning here is the concept of "stone-age unary", which is actually still a viable and relatively popular number system. When cavemen started realizing the needed some way of keeping track of the quantity of things, they started saving a small stone for each thing they had. For example, if they had 12 cows, they would store 12 small stones in the pockets of their stone-age loincloths. This worked great for small quantities, but was less effective for larger herds. This counting system is referred to as a stone-age unary in that each stone represented one thing being counted. We still often use this counting system today with the notion of tick-marks. For example, outlaw cowboys cut one groove in the handle of their six-shooters for each person they killed. Similarly, academic personnel

[44] Although the usage of numbers is often considered an apparent first step in human de-evolution.
[45] It was Mr. Fangman; the year was 1975.
[46] Although some individuals may take longer, particularly individuals who have job titles such as "administrators".

administrators carve a single notch in their desks for each person they harass or fire. Another popular example is what most of us learned early in grade school and probably still use today (I know I sure do).

It is common to use tick marks to count various things; Figure 4.3 shows an example of such a counting system. Note that this method of counting had the added feature of being easy to perceive a total number of things with the standard grouping of "five" things. The number represented by the marks is Figure 4.3 is 23, which also happens to be the IQ of the average academic administrator.

Figure 4.3: A modern and useful usage for "stone-age unary".

4.4 Number Systems and Binary Numbers

Although you've been working with numbers and number systems most of your life up until now, a quick review of the some of the underlying structure and definitions is in order. We'll go more into depth with our study of number systems in a later chapter; providing a brief introduction to the binary number system is the primary focus of this chapter. Keep in mind that the reason binary numbers are so important in the study of digital design is the fact that a binary number nicely models the high-voltage vs. low-voltage relationship in the underlying transistor implementations of digital circuits.

First, here are a few quick definitions. The concepts presented in this section should be nothing new to you but many of you may have never seen or simply forgotten the actual definitions. It's sort of sad… although you're probably able to tweak around with multi-variable calculus but you also probably have forgotten what exactly a radix point is. Welcome to higher education.

> **Number System:** a language system consisting of an ordered set of symbols (called digits) with rules defined for various mathematical operations.
>
> **Digit:** a symbol used in a number system.
>
> **Radix:** the number of digits in the ordered set of symbols used in a number system.
>
> **Number:** a collection of digits; a number can contain both a *fractional* and *integral* part.
>
> **Radix Point:** a symbol used to delineate the fractional and integral portions of a number.

As example, consider a decimal number (radix = ten). Since the number is a decimal number, we can use either one of ten different symbols to represent a decimal number (0, 1, 2, 3, 4, 5, 6, 7, 6, 8, or 9)[47]. If we were only limited to ten numbers in this number system, the number system would be of little use to us. However, by placing digits side-by-side and including some special rules, we can represent just about any possible number. When we place digits side-by-side, we are representing numbers in what is known as *juxtapositional notation*. Using juxtapositional notation allows a given number system to represent numbers greater than the "*radix-1*". Number systems can use juxtapositional notation for any

[47] Keep in mind that these symbols are arbitrary; if you don't like them, feel free to create your own.

radix value. Each of the digit positions in juxtapositional notation can be any of the digits in the ordered set for the given radix. For decimal numbers, the numbered set is: [0,1,2,3,4,5,6,7,8,9].

Figure 4.4 lists some other fun facts regarding numbers and juxtapositional notation. Figure 4.4 shows that numbers are divided into their *integral* and *fractional*. The radix point delineates the integral and fractional portions of the number[48]. Each digit in both the fractional and integral portions of the number is a member of the set of numbers associated with the given radix.

$$\text{NUMBER} = (N)_R = (\text{Integer Part}) \,.\, (\text{Fractional Part})$$

↑

Radix Point

Figure 4.4: The form of a typical number.

Figure 4.5 provides an alternative and more formal definition of a number. This definition also includes some of the typical lingo used to describe numbers. Note that there is nothing too amazing about this approach; it's simply the convention that most everyone happens to use.

$$\text{NUMBER} = (N)_R = (A_{n-1}\, A_{n-2} \ldots A_1\, A_0 \,.\, A_{-1}\, A_{-2} \ldots A_{-m})_R$$

where:

$R \equiv$ Radix
$A \equiv$ a digit in the number
$A_{n-1} \equiv$ the most significant digit (MSD)
$A_{-m} \equiv$ the least significant digit (LSD)

Figure 4.5: Another form of a typical number.

Example 4-3

Describe the integral and fractional portions of the following number: 989.45

Solution: The solution to this problem should be second nature to you. "989" is the integral portion of the number; "45" is the fractional portion of the number. Note that the radix point divides the integral and fractional portions of the number. Also, note that since there is no listed radix value, the radix value of ten is implied and thus the number is a decimal number. The standard we'll use in this text is that if a radix value is not listed, then the number is a decimal number. Numbers listed in other radii should explicitly list the radix or explicitly state somewhere that the radix is something other than ten.

[48] The radix point is that funny dot that you're not supposed to call a decimal point unless the radix is ten.

4.4.1 Common Digital Radii

There are four common radii used in the study of digital design: 10, 2, 8, and 16. These number systems are sometimes referred to as base 10, base 2, base 8 and base 16, where the "base" is the same number as the radix. For this introduction to numbers, we'll only be looking at numbers with a radix of ten (decimal) and a radix of two (binary). Table 4.3 shows the symbol set for the decimal and binary numbers. The important thing to note here is that the set of symbols for a binary number comprises of only '0' and '1', with the highest valued number in each set being the number that equals the radix-1. Also, note that the values in Table 4.3 read lowest values to highest values (left to right). The last thing to keep in mind is that number systems generally share digits; this means that the binary and decimal '1' and '0' are the same symbol. This choice is once again arbitrary, but it's a useful choice to reuse symbols we're already used to using.

RADIX	NAMES	SYMBOL SET
10	decimal	0,1,2,3,4,5,6,7,8,9
2	binary	0,1

Table 4.3: The most commonly used radii in the study of digital things.

4.5 Juxtapositional Notation and Numbers

The use of juxtapositional notation allows a given number system to represent quantities larger than the (radix-1). You are already familiar with such notation since you have spent most of your lives dealing with decimal numbers. The theory behind this notation is no different in other bases but we'll remind you of it here.

Juxtapositional notation means that the symbols in a given number system are placed side-by-side in order to represent quantities larger than the numbers in the given set. Assigning a weight to every digit position in the number allows the number system to represent any value. By convention, the numbers are monotonically increasing (scanning right to left) powers of the radix in question. The main thing to remember here is that the weighting of the digit to the immediate left of the radix point is the radix raised to the zero power[49]. These attributes are widely accepted conventions but are by no means required. The following two examples demonstrate these ideas; these two examples use radii of ten and two, respectively. Be sure to compare and contrast the items that are the same and different.

Example 4-4

Show the weightings associated with each digit in the following number: 987.45

Solution: Table 4.4 shows the solution to Example 4-4. The important thing to notice about this solution is that the radix exponential row uses the radix to monotonically increasing/decreasing powers to designate the weightings. This convention follows the juxtapositional number conventions listed in Figure 4.5.

[49] This is done by convention; there is not rational reason for this except this is the way it's always been done.

Decimal Value of Digit Weight	100	10	1		0.1	0.01
Radix Exponential	10^2	10^1	10^0		10^{-1}	10^{-2}
Positional Value	9 x 100 (900)	8 x 10 (80)	7 x 1 (7)	.	4 x 0.1 (0.4)	5 x 0.01 (0.05)
				↑ Radix Point		

Table 4.4: The solution to Example 4-4.

Example 4-5

Show the weightings associated with each digit in the following number: 101.11_2

Solution: Table 4.5 shows the solution to Example 4-5.

Binary Value of Digit Weight	4	2	1		0.5	0.25
Radix Exponential	2^2	2^1	2^0		2^{-1}	2^{-2}
Positional Value	1 x 4 (4)	0 x 2 (0)	1 x 1 (1)	.	1 x 0.5 (0.5)	1 x 0.25 (0.25)
					↑ Radix Point	

Table 4.5: The solution to Example 4-5.

Since decimal and binary are the two primary radii used in digital design (decimal for humans; binary for actual digital hardware), you should definitely understand the previous examples and memorize the numbers listed in Table 4.6. You'll be using binary and decimal numbers throughout this text. In addition, we'll present some other useful radii in a later chapter.

You'll eventually commit these numbers to your memory because you'll be using them so often in digital design and computer-type applications. However, if you put the time into memorizing them now, you'll save yourself a lot of time, effort, and struggle in the near future. (Hint: take notice of the pattern, as it changes from right to left) Note that the binary numbers listed in Table 4.6 are listed as four binary digits, or *bits*. As you'll see later, a group of four bits has special significance in digital design land.

Including the leading zeros in the binary numbers enables you to read the number quickly; omitting the leading zeros does not change the value of the number.

Decimal (base 10)	**Binary** (base 2)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Table 4.6: Numbers that every successful digital designer has memorized. The binary numbers are presented in groups of four for reasons that will become apparent later.

4.6 Important Characteristics of Binary Numbers

Digital design uses binary numbers quite often because the transistors that form the underlying physical hardware operate in one of two modes: roughly speaking, either "off" or "on". A later chapter presents the low-level details of binary number, but this chapter lists some quick characteristics as a preparation for upcoming design discussions. These two characteristics are massively important; you'll be using them everywhere throughout digital design. Therefore, it's a great idea to make sure you understand them and commit them to memory.

Table 4.6 shows one of the important characteristics to note about binary numbers. Quite often in digital design land, there is an issue of given how many unique numbers can be represented by "X number of bits". There is a special relationship in a binary number system that uses monotonically increasing powers for the bit-position weight values. For example, were you are only considering one bit, you can have two unique numbers: '0' and '1'. If you have two bits, you can have four unique numbers: "00", "01", "10", and "11". If you have three bits, you can form eight unique numbers (too many to list). Table 4.6 shows that with four bits, you can form 16 unique numbers. Figure 4.6 lists this relationship.

$$\text{Number of unique numbers} = 2^{\text{number of bit locations}}$$

Figure 4.6: The relation between the number of bit locations and total number of representable unique numbers.

Example 4-6

How many unique numbers can be represented by an 8-bit binary number?

Solution: The solution to this problem utilizes the formula in Figure 4.6. The quantity of unique numbers = $2^{\text{number of bit locations}} = 2^8 = 256$

The other important characteristic of binary numbers is the notion of the "range" of numbers that a given set of bits can represent. There is a lot more to this notion than meets the eye, so we'll only consider one last characteristic; a later chapter fills in the details.

Our brief discussion of binary numbers so far has only included unsigned binary numbers. In this context, an unsigned binary number is considered to be any number that is not negative; specifically, this means any number that is zero or greater. In real life, binary numbers can be interpreted as either signed binary numbers or unsigned binary numbers. The notion of how to represented signed numbers is slightly more involved which is why we're saving the topic for another chapter.

Figure 4.7 shows the closed form formula that describes the number range that is representable by a given number of bits in an unsigned binary number. A following example shows the ease at which you can apply this formula. This formula uses square brackets to represent the fact that the range is inclusive of the boundary numbers of the range listed in the formula.

$$\text{Number Range for Unsigned Binary Numbers} = [0\text{-}(2^{\text{number of bit locations}} - 1)]$$

Figure 4.7: The formula showing the number range for an unsigned binary number based on the number of bits in the number.

Example 4-7

What range of decimal numbers can be represented by an 8-bit unsigned binary number?

Solution: The solution to this problem utilizes the formula in Figure 4.7.

Number Range for Unsigned Binary Number = $[0\text{-}(2^{\text{number of bit locations}} - 1)] = [0\text{-}(2^8 - 1)] = [0\text{-}255]$

Note that the range [0-255] represented 256 unique decimal numbers, which happily agrees with the solution of Example 4-5.

Example 4-8

How many unique numbers can a 6-bit binary number represent? For this problem, assume that binary number uses standard weightings.

Solution: Since there are six bit locations, and each bit can take on two different numbers (since it is binary). The answer is thus two (the binary radix) raised to the sixth power (with six being the number of bits in the number). Here is the complete formula:

$$\text{Unique numbers} = 2^6 = 64.$$

Example 4-9

Consider a 6-bit binary number; list the maximum value, the minimum value, and the two numbers in the middle of the number range that these six bits are able to represent.

Solution: The list below provides the requested numbers:

The maximum number: this would be all "1's", or "111111" = 63

The minimum number: this would be all "0's", or "000000" = 0

The two middle numbers are based on the fact that there are always an even number of numbers available for a given number of bits. The two numbers in the middle of the range are "011111" and "100000"; these numbers represent 31 and 32, respectively.

Chapter Summary

- Engineering notation is a subset of scientific notation and is always used to represent numbers when being understood is a requirement[50]. Engineering notation uses a magnitude and exponential parts to represent numbers. The magnitude part must be in the range [1,1000); the exponential part must be an integral multiple of three and be represented with standard metric prefixes.

- The development of numbers resulted from the need to process larger "quantities" of things. Human brains are limited in the number of real things they can process, the invention of "numbers" allows human brains to comprehend and process larger quantities of things

- Numbers represent quantities that are too big for our brain to understand and process. Numbers are formed by using a basic set of symbols associated with the particular radix in question. Numbers use juxtapositional notation to represent quantities larger than the numbers represented by the associated symbol set. Digit positions are assigned weightings and each position in a number has a different weighting. Numbers are comprised of both integral and fractional portions, which are delineated by a radix point.

- Digital design uses binary numbers because of the fact that a binary number nicely models the high-voltage vs. low-voltage relationship in the underlying transistor implementation of digital circuits.

- Two important characteristics of unsigned binary number are 1) the number of numbers that can be represented by a given number of bits, and, 2) the range of number that can be represented by a given number of bits. These quantities can be represented by closed form formulas:

$$\text{Number of Unique Numbers} = 2^{\text{number of bit locations}}$$

$$\text{Number Range for Unsigned Binary Numbers} = [0-(2^{\text{number of bit locations}} - 1)]$$

[50] Which is why academic administrators never use engineering notation.

Chapter Exercises

1) Convert the following values to engineering notation.

a) 235500000
b) 45×10^{-4}
c) 241.3×10^{8}
d) -33.8×10^{-4}
e) 0.00303×10^{-4}
f) 0.146×10^{8}
g) $0.0000000253 \times 10^{4}$
h) 8.355×10^{7}

2) Which of the following numbers are larger?

a) 235500000 or 23.55 x x 10^{-6}
b) 4.5m or 45×10^{-4}
c) 241.3M or 241.3×10^{8}
d) -33.8×10^{-6} or -33.81×10^{-6}

3) If you had 153 items in your backpack, can you think of a way to describe those items other than using numbers? If you can think of ways, how much do those ways differ from stone-age unary?

4) How would you classify the Morse code in terms you learned in this chapter?

5) Juxtapositional notation seems like a pretty good idea, but, can you think of anything better?

6) How many unique numbers can be represented by a 4, 8, and 12-bit binary numbers? For this problem, assume that standard weightings are used for the binary number.

7) List the number ranges (in decimal) that can be represented for 4, 8, and 12-bit binary numbers. For this problem, assume that standard weightings are used for the binary number.

8) Briefly described why are binary numbers used so often in digital design?

9) Write closed form formulas that show the middle two decimal numbers of any given number of bits in an unsigned binary number.

10) Consider a 4-bit unsigned binary number that uses the following weighting (listed from left-most to right-most bits): 2^1, 2^2, 2^1, and 2^0. (Don't laugh, people actually do things like this). How many unique numbers can be represented by this range? List the unique numbers that can be represented by this range. List at least one advantage and one disadvantage of using this set of weightings for a binary number.

Design Problems

1) Design a circuit that has three inputs and two outputs. One of the outputs indicates when the 3-bit input value is less than three; the other output indicates then the input is greater than five. Provide the equations that describe your circuit. Implement the final circuit using AND gates, OR gates, and inverters.

2) Design a circuit that has three inputs. One indicates if the value of the 3-bit input is either one or six (decimal). Another of the inputs indicates if the en the 3-bit input value is less than four. The final output indicates when the 3-bit input is five or greater. Provide the equations that describe your circuit. Implement the final circuit using AND gates, OR gates, and inverters.

5 Chapter Five

(Bryan Mealy 2012 ©)

5.1 Introduction

The previous chapters hopefully gave you a small taste for what exactly is meant by the term "digital" and the term "model". In this chapter, we'll combine these two words and actually do something that intelligent and creative people refer to as "digital design". Digital design is not just the pseudo-title of this book; it's where we really want to be.

The "art" of digital design has not as of yet been refined to the point of being able to present the subject in only one way using the same set of definitions and algorithms. From a beginner's standpoint, you can certainly do the digital design process by rote, which is not a good thing unless you intend to kill off your basic creative nature. Although this chapter presents a single approach to digital design, it is by no means the only approach nor is this approach carved in stone. This chapter represents the first step in digital design. Your mission is to realize that digital designs are solvable and representable in many different ways. Have fun; it's really not that big of a deal.

Main Chapter Topics

- **DIGITAL DESIGN OVERVIEW:** This chapter uses a simple design example to introduce a structured digital design process. The chapter presents the "iterative" approach to digital design.
- **BOOLEAN ALGEBRA:** This chapter introduces Boolean algebra including its basic axioms and associated theorems.

Why This Chapter is Important

This chapter is important because it provides a basic approach to solving digital design problems. This particular approach employs Boolean algebra, which represents the foundation of all digital design.

5.2 Digital Design

If you're reading this sentence, it may be because you have opted to pursue the path of becoming an engineer. Although I've officially been an engineer for way too long, I'm still not sure what an exactly an engineer is or what an engineer does. As best that I can figure, an engineer is a person who solves

problems[51]. More specifically, an engineer is a person who solves problems that have enough technical content to such that they deemed "engineering problems". Since an engineer is a problem solver, and since this text is all about digital design, and since you're obviously reading this text, let's take a look at a basic model for solving digital design problems.

Being the average smart person that you are, you've probably solved a lot of problems during your life. However, have you ever really analyzed your approach to solving problems? In case you haven't, we'll talk about a possible approach in this section. As best that I can see it, the following verbage lists the approach that I generally take to solving a problem. Note that this approach is generic enough to be applicable to any problem, not just digital design problems. I suspect that everyone who considers themselves a problem solvers take a similar approach. Here is my basic *algorithm* to solving problems[52].

1) **Define the problem:** understand the starting point and requirements
2) **Describe your solution to the problem:** propose a path to the solution
3) **Implement your solution to the problem:** embodiment of the solution

The following verbage represents an introduction to digital design presented in the context of an actual problem. There is a lot of information presented so try not to lose track of the basic approach. Keep in mind that we're designing a digital circuit; you might want to take a different approach if you were designing a stick in the mud.

5.2.1 Defining the Problem

The basis of any design problem is a relatively clear statement of the problem at hand. As previously stated, in digital design you typically face the notion of designing a digital circuit that processes some set of inputs and generates the desired output. It is understood that for here and evermore in this text, that both the inputs to the circuit and the outputs from the circuit are digital values[53]. Figure 5.1 provides the problem statement for this painfully long design example.

Example 5-1

Problem Statement: Design a digital circuit where the output of the circuit indicates when the 3-bit binary number on the input is greater than four.

Figure 5.1: The problem statement.

The basic concept of all digital design is simple: you're simply creating a circuit that provides the correct output(s) to a given set of input(s)[54]. Read that sentence again, it's important. An issue here for you to realize is that there are many approaches to performing digital design; this section presents only one of them. What you'll find is that you will eventually develop your own style and approach to digital design as you gain experience with the digital design process and problem solving in general. Keep in

[51] As opposed to administrators: their main goals are to create problems for engineers and then do their best to prevent the engineers from solving them. Administrators are basically jealous that they don't have the brains or work ethic to become engineers.
[52] Typical administrator are not aware of any such problem solving algorithms; but be sure to ask them about their many "justifying their existence" algorithms.
[53] Many circuits contain both analog and digital circuitry; we'll only work with digital circuitry.
[54] What you see later in this course is that the outputs can also be based on a sequence of inputs. For now, we'll pretend that the circuit outputs are based solely on the circuit inputs at a given time.

mind that the overall goal is to solve the problem; the main goal here is to familiarize you with a simple digital design process. You'll initially be on a mission to collect tools and experience with digital design: the more you learn, the more you'll realize that you'll never use a significant portion of what you learned.[55]

The first step in defining this problem is to translate what the problem is asking in the problem statement (words) to some other form. A good place to state with any digital design problem is to draw a block diagram that clearly shows both the inputs to and the outputs from the circuit. Drawing a diagram of the circuit should be the first step in solving any digital design problem. From the problem statement, you can see that the digital circuit that satisfies this problem has three inputs (the 3-bit binary number) and one output (states a quality of the inputs that we're interested in).

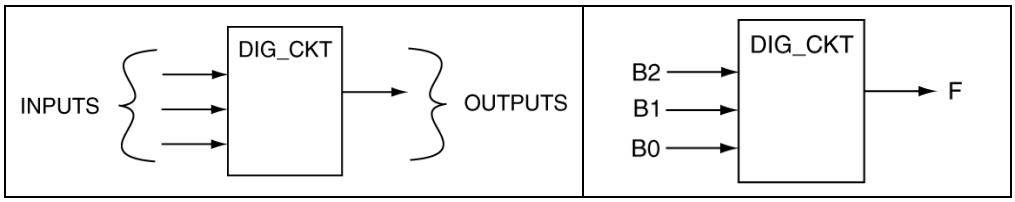
The starting point in any digital design is to draw a black box model that clearly indicates the stated circuit inputs and outputs. We've seen this approach to modeling other things in a previous chapter. Recall that a model in this digital context is simply a description of a digital circuit. This is purposely a loose definition because there are once again about a bajillion-and-one ways to describe a digital circuit. The diagram in Figure 5.2(a) is just one of these ways.

The nice thing about the diagram of Figure 5.2(a) is that it clearly shows that our final circuit has three inputs and one output (as indicated by the direction of the arrows of the labels listed in Figure 5.2). Figure 5.2(b) shows another model of our final circuit. The main difference between these two models is the fact that the model in Figure 5.2(b) has given specific names for the inputs and outputs. Note that the circuit models of Figure 5.2(a) and Figure 5.2(b) show roughly the same thing but the Figure 5.2(b) provides a greater amount of detail and is probably a better model in the context of this problem. Recall that the term "model" does not imply a specific level of detail.

For the purposes of solving this problem, the model of Figure 5.2(b) is better because we need to use the signal names in this approach to solving this problem. The signal names applied to the model in Figure 5.2(b) are nothing special: the "**B**" could mean binary; the numbers following the **B**s are *probably* (we'll comment on this soon) associated with the weighting factors of the binary numbers. The "**F**" is a typical name given to the outputs of a digital circuit because the output is a function of the inputs (more on this later).

There one piece of important information missing from the model of Figure 5.2(b): since the three inputs represent a binary number, what are the binary weights of the inputs? You need to state this in your problem solution in order for the solution to have meaning. For this problem, let's consider the B2 input to be the *most significant bit* (MSB) and the B0 input to be the *least significant bit* (LSB). If you did not state this extra piece of information, your solution in the context of the model of Figure 5.2(b) would not make sense. In general, you must always state this extra information when you are performing digital design. You could probably make an implicit assumption here, and be correct, but it's always better to explicitly state your assumptions somewhere in the design process, preferably early on. It may be tedious, but recall one purpose of having a model is for anyone to look at it once and quickly understand what is going on.

[55] As you will notice later, the design process in this chapter is really inefficient; you'll rarely use it.



(a) (b)

Figure 5.2: Two different models of the proposed digital circuit.

The next step in solving this problem is to establish a relationship between the circuit's inputs and outputs. Since we're the digital designers here, the approach we'll take is to state an input/output relationship such that the given problem is solved. The way we'll do this is to list every possible unique combination of the three inputs and assign an output value that indicates when the inputs satisfy the original problem. This approach of specifying the input/output relationship represents one of many valid techniques used to solve digital design problems. The table used to display this input/output relationship is referred to as a *truth table*. Figure 5.3 shows the truth table for this problem: Figure 5.3(a) shows the empty truth table while Figure 5.3(b) shows the truth table with every possible combination of the three binary inputs and output that indicates when the input combination solves the stated problem.

B2	B1	B0	F

(a)

B2	B1	B0	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b)

Figure 5.3: The empty and completed truth table for Example 5-1.

The following list describes some of the interesting and important things to note about the truth tables shown in Figure 5.3.

- Figure 5.3(a) shows an empty truth table while Figure 5.3(b) shows a truth table that is filled in with 1's and 0's. The fact that 1's and 0's are used is for modeling digital values is traditional in digital design (it's easier and faster to write than other options such as ON-OFF). Digital circuitry and digital models typically use 1's and 0's to model the voltages that drive the underlying hardware of the circuit. Using 1's and 0's at this point in the design allows us to abstract past the need to deal with voltages (which is a good thing for people who don't know what voltage is). We'll be modeling voltages using 1's and 0's for the remainder of this text.

- The tables have eight rows. There is always a binary relationship between the number of inputs to the circuit and the number of rows in the truth table. For this example, since there

are three inputs, there are 2^3 (eight) unique combinations of the three inputs. Be sure to note that the decimal equivalents to the listed input values range from zero to seven (0-7), because in binary, the counting begins at 0 ("000") and ends at 7 ("111").

- The truth table is set up so that F is truly a *function* in every sense of the word. This is no different from the concept of functions in mathematics where there are independent variables and dependent variables. For this example, B2, B1 and B0 are the independent variables while F is the dependent variable. In this case, the value of F is dependent upon the values of the B2, B1, and B0 inputs. In addition, the output F has only one value for each possible input combination. If the output were to have two different values for one row in the truth table, the functional relationship would not exist and the world would be sad.

- The first three columns of the truth table form every unique combination of the three input values. The column for the output shows what we want the circuit output to be if a particular input combination specified by the problem specification appears on the inputs. For this example, we entered 0's for the cases where the inputs bits represent a number less than five. Conversely, we enter 1's for the cases where the input combination is greater than four[56]. In other words, a '1' in the F column shows when the input combination, which represents a binary number, is greater than four as specified by the problem.

- The truth table includes an extra grid line in the middle row of the truth table in order to increase the readability of the table. Truth tables are typically divided into rows of four thus proving that life is good.

This is the end of the first step: the problem is now 100% defined using the truth table in Figure 5.3(b). In case you're thinking that this problem is sort of straight-forward (and boring) in the way that the outputs were specified, you're right. There is no real secret to this style of designing circuits. This particular style of digital design is an exhaustive approach in that the truth table lists every possible input combination. This approach is referred to as the *iterative* approach to digital design (or *iterative design*) but I like to refer to it as BFD (for *brute force design*).

Would an iterative approach be possible if the circuit had 24 inputs? No. Therein lays one of the basic limitations of the iterative approach. But not to worry, the approach taken in this problem serves a good purpose and justifies the learning of other more efficient approaches in later chapters. Onto the next step.

5.2.2 Describing the Solution

Although the truth table has completely defined the solution to this problem, it is generally somewhat klunky to work with, especially as the number of circuit inputs increase. What we need to do is develop a "science" of sort in order to more efficiently and generically describe the problem's solution. If we are able to develop this new science, we'll be able to solve more complex digital design problems. Lucky for us that someone a real long time ago already developed the "science" we're looking for. Here's the shortened version of the story[57].

[56] As you'll see later, this approach is somewhat arbitrary; the problem could also be solved by swapping the 1's and 0's in the output column. We'll deal with this later.

[57] This is really the short version; this is definitely an area you'll want to explore in other digital design textbooks.

About a bajillion years ago, a guy named George Boole developed some methods to deal with a two-valued algebra[58]. Although his original intent was to model logical reasoning in a mathematical context, his work currently forms the basis for all digital design. This two-valued algebra has since come to be known as Boolean algebra. Boolean algebra, similar to normal algebra, is based on a set of operators defined over the set of elements in question. The possible elements in this set are {0,1} which clearly shows the two-values (that binary thang again).

Table 5.1 lists the basic axioms of Boolean algebra. The basic operators in Boolean algebra, namely, the dot (•), the cross-looking symbol (+), and the overbar (¯), are completely defined by the axioms shown in Table 5.1. Table 5.2 and Table 5.3 list the theorems that are provable using the axioms in Table 5.1. In case you actually want to prove these theorems, substitute the binary values into the expressions in the theorems using the axioms. Go for it. I dare you[59].

1a	$0 \cdot 0 = 0$	**1b**	$1+1=1$
2a	$1 \cdot 1 = 1$	**2b**	$0+0=0$
3a	$0 \cdot 1 = 1 \cdot 0 = 0$	**3b**	$1+0=0+1=1$
4a	$\bar{0} = 1$	**4b**	$\bar{1} = 0$

Table 5.1: Boolean algebra Axioms

5a	$x \cdot 0 = 0$	**5b**	$x \cdot 1 = x$	**Null element**
6a	$x \cdot 0 = 0 \cdot x = 0$	**6b**	$x + 0 = 0 + x = x$	**Identity**
7a	$x \cdot x = x$	**7b**	$x + x = x$	**Idempotent**
8a	$\bar{\bar{x}} = x$			**Double Complement**
9a	$x \cdot \bar{x} = 0$	**9b**	$x + \bar{x} = 1$	**Inverse**

Table 5.2: Single variable theorems.

10a	$x \cdot y = y \cdot x$	**10b**	$x + y = y + x$	**Commutative**
11a	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	**11b**	$(x + y) + z = y + (x + z)$	**Associative**
12a	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	**12b**	$x + (y \cdot z) = (x + y) \cdot (x + z)$	**Distributive**
13a	$x \cdot (x + y) = x$	**13b**	$x + (x \cdot y) = x$	**Absorption**
14a	$(x \cdot y) + (x \cdot \bar{y}) = x$	**14b**	$(x + y) \cdot (x + \bar{y}) = x$	**Combining**
15a	$\overline{(x \cdot y)} = \bar{x} + \bar{y}$	**15b**	$\overline{(x + y)} = \bar{x} \cdot \bar{y}$	**DeMorgan's**

Table 5.3: Two and three-variable theorems.

[58] In case you have forgotten what algebra is, it's a mathematical system used to generalize arithmetic operations by using letter or symbols to stand for numbers based on rules derived from a minimal set of basic assumptions. These basic assumptions are referred to as axioms. An axiom is a statement universally accepted as true. From this set of axioms, theorems can be proved true or false. A theorem is a proposition that can be proved true from axioms.

[59] Proving the theorems using the basic axioms is a typical exercise in most digital design texts. We'll opt to move onto more useful things.

The most important result gathered from the basic axioms of Table 5.1 is the definition of the three operators. Although the axioms completely define these operators, the definition of these operators is clearer using a truth table. The three operators actually have names. The dot operator (•) is referred to as the AND operator and used to signify an AND operation (sometimes referred to as logical multiplication). The cross operator (+) is referred to as the OR operator and is used to define an OR operator (sometimes referred to as logical addition). The overbar is referred to as the NOT operator and is used to define a NOT operation (usually referred to as inversion or complementation). Table 5.4 shows the truth tables associated with these three definitions.

AND (logical multiplication)	**OR (logical addition)**	**NOT (inversion)**

x	y	$F = x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$F = x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	$F = \bar{x}$
0	1
1	0

Table 5.4: Truth tables for the three basic logical operators.

Recall that the goal of this section was to produce a scientific method of describing the function associated with the solution of the original problem. Since that problem appeared about five pages ago, Figure 5.4 once again provides the truth table defining the solution to this problem.

B2	B1	B0	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 5.4: The truth table for the original problem.

We now have several different ways of describing the function that provides a solution to the problem at hand. The first representation is the truth table, which we declared as being klunky. A second solution is sort of a verbal and thus non-scientific solution. Figure 5.5 shows the long and drawn out text of this verbal solution. Notice that Figure 5.5 extensively uses of the words “and” and “or” in the solution. However, since we went to all the trouble to describe Boolean algebra, Figure 5.6 shows a better (more efficient and scientific) way to describe the function using Boolean algebra. Note the similarities in the solutions shown in Figure 5.5 and Figure 5.6.

The output of the circuit is a '1' when:

(B2=1 and B1=0 and B0=1) or (B2=1 and B1=1 and B0=0) or (B2=1 and B1=1 and B0=1)

Figure 5.5: One approach to describing the solution to Example 5-1.

$$F(B2,B1,B0) = B2 \cdot \overline{B1} \cdot B0 + B2 \cdot B1 \cdot \overline{B0} + B2 \cdot B1 \cdot B0$$

Figure 5.6: A better approach to describing the solution to Example 5-1.

There are several important things to note about the in Figure 5.6.

- This is truly an equation (note the presence of the equals sign). This equation is referred to as a *Boolean equation* or sometimes as a *Boolean expression*. This expression is written in functional form in that the complete set of independent variables is listed directly on the left side of the equals sign while the dependent value is listed on the right of equals sign.

- The expression implies some form of precedence of the AND, OR, and NOT operators. The NOT operator (represented by the bars above the independent variables) has highest precedence followed by the AND, and then the OR function. Boolean expressions such as these can be written using parenthesis around the individual terms that are being ANDed together[60]. Figure 5.7 shows an example of the equation of Figure 5.6 with a refreshing use of parentheses is

$$F(B2,B1,B0) = (B2 \cdot \overline{B1} \cdot B0) + (B2 \cdot B1 \cdot \overline{B0}) + (B2 \cdot B1 \cdot B0)$$

Figure 5.7: An arguably better approach to describing the solution to Example 5-1.

5.2.3 Implementing the Solution

Up to this point, you've defined your solution (step 1) and described your solution (step 2) which means you're now ready to implement your solution. The reality is that the word *implement* has many connotations; what we mean in this context is that we need some way to implement this function in actual hardware[61]. All you currently know are the basic functions associated with Boolean algebra: AND, OR, and NOT. If actually had to implement this circuit in hardware, how would you do it?

There just so happens to be entities out there referred to as "logic gates" that implement the individual logic functions for you. Just as there are AND, OR, and NOT functions, there also happen to be physical circuits (AND, OR, and NOT gates) that implement these functions. In other words, a *logic gate* is a physical device that implements a logic function.

[60] Use of parenthesis reduces the need to memorize operator precedence. So, if in doubt, use parenthesis.
[61] A typical digital design synonym for implementing a function in hardware is to "realize" the function or "function realization".

There are many types of gates out there in digital design-land, but for now, we'll only deal with these basic three gates. Figure 5.8 shows these three basic gates. Once again, the symbols shown in Figure 5.8 are nothing more than models of gates. In other words, the gates represent the associated logic functions but without providing details as to how the functions are implemented on a transistor level. As you can probably see by now, this form of abstracting is typical in digital design. These gates typically use a relatively small number of transistors in their implementations[62].

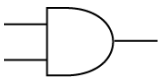
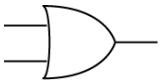
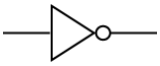
AND gate	OR gate	Inverter
		

Figure 5.8: The basic gate symbols used to model AND, OR, and NOT functions.

Also good to note here are some issues associated with AND and OR gates.

- AND gates and OR gates must have at least two inputs but are not limited to a maximum number of inputs. In the cases of more than two inputs, the functions remain consistent. Figure 5.9 lists a more generic definition of AND and OR gates; these definitions completely describe the functionality of these gates when they have more than two inputs[63]. You can hopefully see from this definition that AND and OR gates can have as many inputs as they need while still exhibiting the basic AND and OR functionality.
- AND gates and OR gates can have only one output.
- Inverters can only have one input and one output.

AND gates: the output is only a '1' when all the inputs are a '1'

OR gates: the output is only a '0' when all the inputs are a '0'

Figure 5.9: A more generic and intuitive definition for AND and OR functions.

These gates now give us the ability to implement the solution in actual hardware. However, for this problem, we're not going to actually implement the circuit. Instead, we're going to provide yet another model for the circuit. Figure 5.10 shows a model of the final circuit implementation. Make sure you understand the relationship between the circuit model shown Figure 5.10 and the Boolean equation shown in Figure 5.6. To test your understanding of this relationship, you should be able to generate the associated Boolean equation shown in Figure 5.6 that describes the circuit from the circuit model shown in Figure 5.10.

[62] Configuring transistors to implement logic functions is a topic covered in most standard digital design texts. The notion of designing gates on a transistor level is massively important when you consider many integrated circuits contain millions of logic gates.

[63] You can add more inputs to the gate symbols as required.

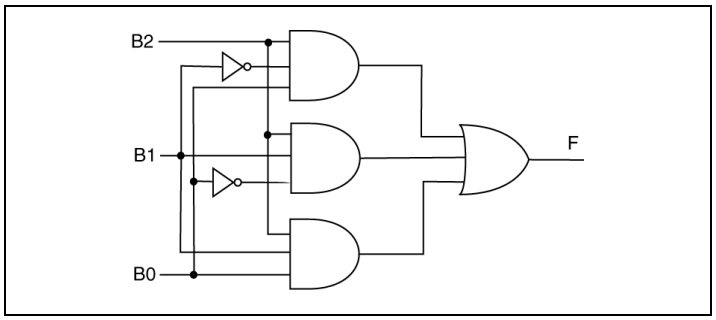


Figure 5.10: The circuit model that solves Example 5-1.

In general, you should be able to go back and fourth between the various representations of a Boolean function. From this example, we learned of and worked with four different representations of a Boolean function: 1) truth table, 2) written description, 3) Boolean equation, and 4) circuit model. As you'll soon find out, there are many more ways to represent a Boolean function. Each one of these representations is officially a model of a digital circuit[64]. Given any one of these models, you should 1) be able to generate any of the other models, and 2) be able to actually implement the circuit.

Example 5-2: Half Adder

Design a circuit that adds two bits. The output of this circuit should show both the sum of the added bits and whether the addition operation has generated a carry-out. This circuit is one of the most basic circuits in digital design and is known as a Half Adder, or HA.

Solution: Although at this point you may not feel as if you know too much about digital design, you truly know enough to design a significant and relatively important digital circuit. With a quick example, we can drive home the design point and have a crapload of fun in the process.

You may require some background in order to do this problem. While performing a mathematical operation using decimal numbers is no big deal for you, performing a similar operation using binary numbers requires a special, slightly altered state of consciousness[65]. What you'll soon discover, if you have not discovered it already, is that performing mathematical operations in decimal and binary follows the same rules. The only difference is that the binary number system only contains two symbols: '0' and '1'.

If you were to add two, single-digit, decimal numbers, your result would either be a single digital number (less than ten) or a two-digit number (greater than nine). Looking at that in another way, results of this addition that are greater than or equal to the radix are represented by two digits while results less then the radix are represented with a single digit. In the case of the two-digit result, one digit represents the result of the addition while the other digit represents the value that "carried-out" from the single-bit addition. The same is true for binary addition. There are only four different possible results for binary addition of single bit: Table 5.5 shows these four possibilities as well as the SUM and Carry-out results.

The only item of relative interest in Table 5.5 is the fact that adding '1' to '1' results in a sum of '0' with a carry-out of '1'. If you consider the Carry-out to be the MSB and the sum to be the LSB, the total

[64] Although not all of these models are dark box models.
[65] Not really.

result is "10" which is the binary equivalent of 2 (two) in decimal[66]. So much for the background, now let's tackle the actual design process.

Operation	SUM	Carry-out (CO)
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

Table 5.5: All possible single-bit addition operations with sum and carry results.

Step 1) Define the Problem: The problem statement for this problem represents a good start at defining this problem. As with all problems, draw a high-level diagram of the final circuit. From the problem statement, you can see that this circuit contains two inputs and two outputs. Figure 5.11 shows the two inputs (arbitrarily named OP_A and OB_B)[67] and two outputs SUM and CO. Table 5.6 and Table 5.7 show the empty truth table (only the independent values are listed) the completed truth table for this design, respectively. In the design process, you generally start out with an empty table and then proceed to fill in the independent and dependent values (the inputs and outputs, respectively) in that order. If you don't do it in this order, no special penalties are applied.

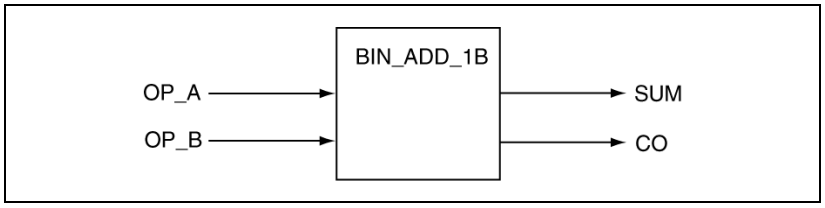


Figure 5.11: The black-box diagram for the example problem.

OP_A	OP_B	SUM	CO
0	0		
0	1		
1	0		
1	1		

Table 5.6

OP_A	OP_B	SUM	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 5.7

[66] OK, I saw a student with the following words written on his t-shirt "There are 10 types of people in the world, those who understand binary and those who do not". Even my TA has the shirt. If this saying is copywritten, then feel free to sue the life out of me. Thanks.

[67] Although you could choose just about any signal names for the inputs and outputs, you should assign names that are self-commenting. In other words, OP_A (operand A) is arguably a better label than FINGER_NAIL although both labels are equally valid.

Step 2) Describe the Solution: Note that for this problem, you'll need to generate two Boolean expressions: one for the SUM and the other for the CO. The final equations are written by logically summing the product terms associated with rows in which 1's appear.

$SUM = \overline{OP\_A} \cdot OP\_B + OP\_A \cdot \overline{OP\_B}$	$CO = OP\_A \cdot OP\_B$

Equation 5-1: The final equations for Error! Reference source not found.**.**

Step 3) Implement the Solution: The final step involves translating the Boolean expressions listed in Equation 5-1 into circuit form. Figure 5.12 shows the final gate-level implementation.

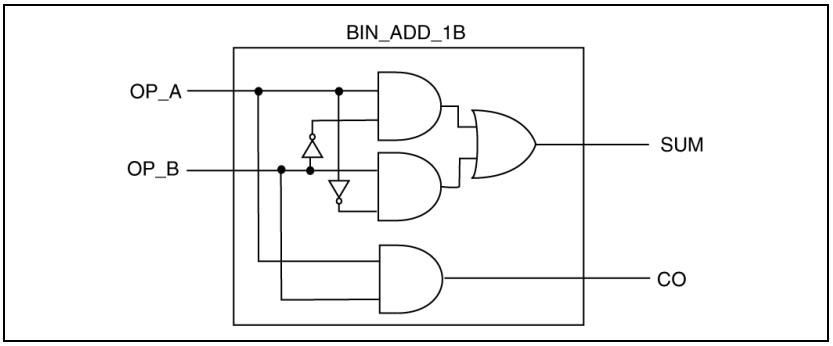


Figure 5.12: The circuit representation of the final solution for Error! Reference source not found.**.**

In case you don't see it yet, you've officially designed one of the basic circuits in digital design-land: the *half-adder (HA)*. In other words, the circuit shown in Figure 5.12 adds two bits and generates a result. The two inputs are digital values as are the two outputs. If you were to purchase the appropriate digital circuitry, you could actually set up this circuit and make it work.

Chapter Summary

- The design of a digital circuit drives by the need to solve a problem. The basic process of digital design can be described in three steps: 1) define the problem, 2) describe the solution, and 3) implement the solution. Solutions to digital design problems are often described with Boolean equations, which have their basis in Boolean algebra.

- There are many possible ways to represent solutions to digital design problems. These many solutions are considered functionally equivalent in that they all describe the same thing but do so in different ways. In other words, if the outputs for two given solutions are equivalent based on the same set of inputs (but the form of the solutions differ), the solutions are functionally equivalent.

- Important Standard Digital Modules presented in this chapter:
    - Half Adder (HA)

Chapter Exercises

1) What entity forms the basis of iterative design?

2) What entity forms the basis of iterative design?

3) Why is the term "brute force" associated with iterative design?

4) Can you, at this early stage in your digital design career, describe a better approach to digital design?

5) Why are truth table-based designs considered severely limited?

6) Generate a Boolean equation that is equivalent to each of the following truth tables.

B2	B1	B0	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(a)

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(b)

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(c)

t	u	v	F1	F2
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

(d)

7) Convert the following Boolean expression to truth table form.

a) $F(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C}$

b) $F(A,B,C) = \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot\overline{B}\cdot\overline{C} + A\cdot\overline{B}\cdot C + A\cdot B\cdot C$

c) $F(X,Y,Z) = \overline{X}\cdot\overline{Y}\cdot Z + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z$

8) Convert the following Boolean functions to truth table form.

a) $F(R,S,T) = (\overline{R}+\overline{S}+\overline{T})\cdot(\overline{R}+S+\overline{T})\cdot(\overline{R}+S+T)\cdot(R+\overline{S}+T)\cdot(R+S+T)$

b) $F(A,B,C) = (A+B+C)\cdot(A+\overline{B}+C)\cdot(A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)$

c) $F(X,Y,Z) = (\overline{X}+\overline{Y}+Z)\cdot(\overline{X}+Y+\overline{Z})\cdot(X+\overline{Y}+Z)\cdot(\overline{X}+\overline{Y}+\overline{Z})$

9) Draw a circuit representation for the following Boolean equations:

a) $F(X,Y,Z) = \overline{X}\cdot\overline{Y}\cdot Z + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z$

b) $F(R,S,T) = (\overline{R}+\overline{S}+\overline{T})\cdot(\overline{R}+S+\overline{T})\cdot(\overline{R}+S+T)\cdot(R+\overline{S}+T)\cdot(R+S+T)$

Design Problems

1) Design a circuit that has three inputs and two outputs. One of the outputs indicates when the 3-bit input value is less than three; the other output indicates then the input is greater than five. Provide the equations that describe your circuit in SOP form. Implement the final circuit using AND gates, OR gates, and inverters.

2) Design a circuit that has three inputs and two outputs. One output indicates when the three inputs (considered a binary number) are even; the other output indicates when the three input bits are odd. Implement the final circuit using AND gates, OR gates, and inverters.

3) Design a circuit whose 3-bit output is two greater than the 3-bit input. The binary count should wrap when the output value is greater than 111_2.

4) Design a digital circuit that controls a switch box according to the following specifications: If either one (and only one) or two (and only two) of the three input switches are on, the output is on. For this problem, assume that "on" is represented by a '1'. Implement your final circuit using AND gates, OR gates, and inverters.

5) Design a digital circuit according to the following specifications. The circuit output indicates when the 3-bit binary input is less than or equal to four but not zero. Provide a proper black box diagram, a truth table, a Boolean equation, and a circuit diagram that model your solution. The circuit diagram should only use only AND gates, OR gates, and inverters.

6 Chapter Six

(Bryan Mealy 2012 ©)

6.1 Introduction

The previous chapters hopefully have given you a sense for the fact that digital design is centered on the use of various model types and representations of digital circuits. What you'll eventually discover is that true digital designers need to be adept at being able to model circuits in a way most appropriate for a given situation.

The previous chapter introduced many theorems based on the basic axioms of Boolean algebra. Aside from DeMorgan's theorem, many of these theorems rarely used in digital design applications. Digital design and other technical fields such as computer science and bowling use DeMorgan's theorem quite often. This chapter presents a foundation for the use of DeMorgan's theorem in digital design; you find extensive use of this theorem throughout this text.

Main Chapter Topics

- **DIGITAL DESIGN REVIEW:** This chapter provides a quick review of the basic design approach presented in a previous chapter.
- **DEMORGAN'S THEOREMS:** Probably the most widely used theorem in digital design, DeMorgan's theorems can describe and generate product of sums and sum of products representations of functions.

Why This Chapter is Important

This chapter is important because it describes how to use DeMorgan's theorem to change Boolean expressions into functionally equivalent forms.

6.2 Representing Boolean Functions

A Boolean function, or simply "function", is an equation that describes an input/output relationship of a module in terms of digital logic. There are many different ways of modeling this input/output relationship. Up until now, you have seen three main approaches: the truth table, a Boolean function, and a circuit model.

There are a few important things to notice about the input/out relationships we've been using. First, these three representations are *functionally equivalent*. In other words, these three forms say the same thing but say it in three different ways. As you become more familiar with digital logic you'll be able to go back and forth between these forms very quickly. Secondly, you'll also quickly realize that some forms of function representations are more appropriate than others in modeling digital circuits. The

other important thing to notice here is that though there are many ways to model the input/output relationship of a digital circuit, only a few of these methods are used most of the time. The good news is that you've seen three of the more standard approaches already.

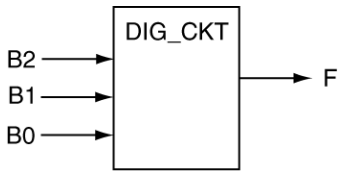
For a quick review, recall that the design process we used in a previous chapter had the three primary steps listed below. We then used these steps to solve Example 6-1 (repeated from a previous chapter).

1) **Define the problem:** understand the starting point and requirements
2) **Describe your solution to the problem:** propose a path to the solution
3) **Implement your solution to the problem:** embodiment of the solution

Example 6-1

Problem Statement: Design a digital circuit where the output of the circuit indicates when the 3-bit binary number on the input is greater than four.

The solution to Example 6-1 included a black box diagram (Figure 5.3(a)), a truth table (Figure 5.3 (b)), a Boolean expression (Figure 5.7), and the final circuit diagram (Figure 5.10).



(a)

B2	B1	B0	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b)

Figure 6.1: The black box model and completed truth table for Example 6-1.

$$F(B2,B1,B0) = (B2 \cdot \overline{B1} \cdot B0) + (B2 \cdot B1 \cdot \overline{B0}) + (B2 \cdot B1 \cdot B0)$$

Figure 6.2: A Boolean expression describing the solution to Example 6-1

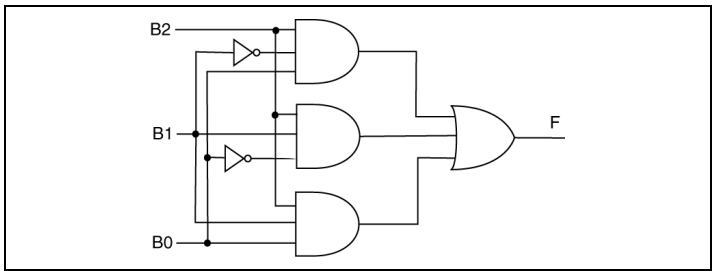


Figure 6.3: The circuit model that solves Example 6-1.

6.2.1 DeMorgan's Theorems

The list of theorems provided in the previous chapter (listed again in the appendix) is relatively long. In standard versions of digital logic courses, students are required to become intimately familiar with these theorems by proving them using the basic digital axioms. The truth is that modern digital design rarely has a direct use for most of these theorems. Then again, the useful theorems are used all the time; DeMorgan's theorem happens to be one of those theorems. Familiarity with DeMorgan's theorem is going to allow us to represent functions in many different ways. This section deals directly with using DeMorgan's theorems to generate new representations, or more specifically, *function forms*, to model digital circuits, and thus, solutions to digital problems.

A new representation of a digital circuit can be derived from an application (or multiple applications) of DeMorgan's theorem after gathering information from a truth table. For this explanation, let's back up to the design example we were previously using. Figure 6.4 shows once again the Boolean equation that describes a solution to that design problem. The form of this equation is referred to as the *sum of products (SOP)* form. Note that this name makes sense in that there are three terms in the equation that are logically multiplied together (these *product terms* have been wrapped in parenthesis). These product terms are then logically added together to complete the equation. The sum of products form, or just SOP form, is probably the most widely used equation form in digital design land. For better or worse, it is not the only form.

$$F(B2,B1,B0) = (B2 \cdot \overline{B1} \cdot B0) + (B2 \cdot B1 \cdot \overline{B0}) + (B2 \cdot B1 \cdot B0)$$

Figure 6.4: The solution to the previous example listed again here.

Another widely used Boolean equation form is referred to as the *product of sums (POS)* form. You can obtain the POS form from the truth table in a way that is similar to the SOP form. Note that in the SOP form, you wrote the Boolean equation based on the rows of the truth table that contained a '1'. In other words, you found which rows contained a '1' in the output and you included the product term for that row in the final Boolean equation.

The technique of inverting the output officially describes the same function (the right-most column shown in Figure 6.5). Note the right-most column in Figure 6.5 is the same as the F column except the associated values are in complemented form. In other words, the two right-most columns of Figure 6.5 have a complementary relationship to each other. Generating an equivalent POS form for the truth table shown in Figure 6.5 is similar to the approach for generating the SOP form. The only difference is that we'll need to apply DeMorgan's theorems multiple times to get to the POS form we're looking for. Let's start this process by taking a look at DeMorgan's theorem.

B2	B1	B0	F	$\overline{F}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Figure 6.5: The truth table for the original problem with a complemented output added.

DeMorgan's theorem is one of the more commonly applied logic theorems in digital design-land because theorem allows for the transformation of a circuit from one form to another. DeMorgan's theorem is also useful in other fields such as discrete mathematics, computer programming, and bowling. Table 6.1 shows DeMorgan's theorems listed in both two variable and generalized forms. The final form listed in Table 6.1 emphasizes the fact the "variables" shown in the original listing of DeMorgan's theorem are not necessarily Boolean variables. The symbols shown in the first two equations can be either simple Boolean variables or Boolean expressions. In either case, the overbar applies to the entire expression that it covers.

$\overline{X+Y}=\overline{X}\cdot\overline{Y}$	$\overline{X\cdot Y}=\overline{X}+\overline{Y}$
$\overline{X_1+X_2+\ldots+X_n}=\overline{X_1}\cdot\overline{X_2}\cdot\ldots\cdot\overline{X_n}$	$\overline{X1\cdot X2\cdot\ldots\cdot Xn}=\overline{X1}+\overline{X2}+\ldots+\overline{Xn}$
$\overline{\text{☯}+\text{☜}+\ldots+\text{}}=\overline{\text{☯}}\cdot\overline{\text{☜}}\cdot\ldots\cdot\overline{\text{}}$	$\overline{\text{☯}\cdot\text{☜}\cdot\ldots\cdot\text{}}=\overline{\text{☯}}+\overline{\text{☜}}+\ldots+\overline{\text{}}$

Table 6.1: DeMorgan's theorem in two-variable and generalized forms.

Let's generate an equation for F in POS form. The key here is to notice that for the SOP form, you were interested in the rows of the truth table that had a '1' for the output. The approach is to list the product terms that have a '1' on the output of the complimented output[68]. This first step is similar to generating SOP form but you're actually generating an equation in SOP form for the compliment of the output[69]. Table 6.2 shows the set of equations generated by seeking a POS expression for the given function. The explanation of each row in Table 6.2 follow the table.

[68] Be sure to note that looking for 1's in the inverted output column is exactly the same as looking for 0's in the non-inverted output column.

[69] Keep in mind that a complement of the output is not the desired output relative to the original problem. In other words, the complement of the output does not represent a solution for the given problem.

(a)	$\overline{F} = (\overline{B2} \cdot \overline{B1} \cdot \overline{B0}) + (\overline{B2} \cdot \overline{B1} \cdot B0) + (\overline{B2} \cdot B1 \cdot \overline{B0}) + (\overline{B2} \cdot B2 \cdot B1) + (B2 \cdot \overline{B1} \cdot \overline{B0})$
(b)	$\overline{\overline{F}} = F = \overline{(\overline{B2} \cdot \overline{B1} \cdot \overline{B0}) + (\overline{B2} \cdot \overline{B1} \cdot B0) + (\overline{B2} \cdot B1 \cdot \overline{B0}) + (\overline{B2} \cdot B2 \cdot B1) + (B2 \cdot \overline{B1} \cdot \overline{B0})}$
(c)	$F = \overline{(\overline{B2} \cdot \overline{B1} \cdot \overline{B0})} \cdot \overline{(\overline{B2} \cdot \overline{B1} \cdot B0)} \cdot \overline{(\overline{B2} \cdot B1 \cdot \overline{B0})} \cdot \overline{(\overline{B2} \cdot B1 \cdot B0)} \cdot \overline{(B2 \cdot \overline{B1} \cdot \overline{B0})}$
(d)	$F = (\overline{\overline{B2}} + \overline{\overline{B1}} + \overline{\overline{B0}}) \cdot (\overline{\overline{B2}} + \overline{\overline{B1}} + \overline{B0}) \cdot (\overline{\overline{B2}} + \overline{B1} + \overline{\overline{B0}}) \cdot (\overline{\overline{B2}} + \overline{B1} + \overline{B0}) \cdot (\overline{B2} + \overline{\overline{B1}} + \overline{\overline{B0}})$
(e)	$F = (B2 + B1 + B0) \cdot (B2 + B1 + \overline{B0}) \cdot (B2 + \overline{B1} + B0) \cdot (B2 + \overline{B1} + \overline{B0}) \cdot (\overline{B2} + B1 + B0)$

Table 6.2: Generating a POS form from multiple applications of DeMorgan's theorem.

(a) This equation is SOP form generated by listing the product terms for the 0's of the F column or the 1's of the complemented F column. Note that there are five product terms in this equation. Also, note that although this is a valid SOP form for the complemented output, we're looking for a POS form for the uncomplemented output. This is an important distinction.

(b) Both sides of the equation in (a) are complemented, which by some Boolean axiom or theorem, preserves the equality. Complimenting both sides of an equation is an extremely common operation when dealing with Boolean algebra.

(c) Since the double complement of a variable equals that variable, the double-complimented F on left side of the equals sign becomes uncomplimented. The result is that we now have an expression for F; our ultimate goal is to have an equation for F in POS form so we still need to massage this equation in order to get it into POS form. The expression on the right side of the equals sign shows the results after the first application of DeMorgan's theorem. Note that the product terms are now complemented and are ANDed together. In other words, the giant overbar is now distributed to the individual product terms and the OR operators were changed to AND operators.

(d) Each of the product terms receives an individual application of DeMorgan's theorem. Once again, the overbar is distributed to the individual components of the product terms and the logic operators are switched from AND to OR.

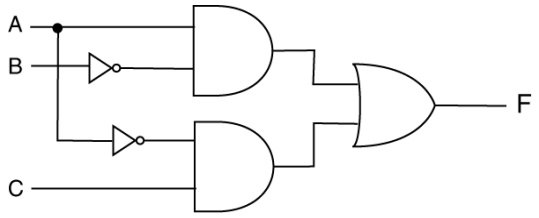
(e) An application of a basic Boolean algebra axiom allows us to remove the double complements from the variables. The final result of this step provides the desired POS form.

In summary, you now have an approach for generating both an SOP and POS form of equations describing a digital relationship. These are massively common forms so make sure you understand them. In particular, understand where these forms came from: the SOP form is generally associated with the 1's of the circuit while the POS form is generally associated with the 0's of the circuit[70]. Once again, The SOP and POS forms are *functionally equivalent*, that is, they describe the same input/output relationship, but they do so in different ways.

[70] This may seem a little "follow the rules" oriented but it will make more sense later as we delve deeper into other digital design topics.

Example 6-2

Change the following circuit implementation from a SOP (AND/OR) to a POS (OR/AND) form.



Solution: First, a comment... As you've seen already, there are many ways to represent a functional relationship in digital-land; you've seen several equation forms (SOP & POS), truth tables, and timing diagrams. That is, for any given relationship, there are functionally equivalent ways to represent the relationship. This being the case, you should be able to go from any one form to any other form. This problem is a case of going from a circuit model implemented in SOP form to a circuit model in POS form. There are actually many ways to solve this problem, but we'll solve it in arguably the most straightforward approach.

We'll take the following steps: 1) write out the equation implemented by the circuit, 2) expand the equation to something that looks like a more familiar SOP form, 3) use the equation to fill in a truth table, 4) solve for the complemented output, 5) complement the equation and *DeMorganize*[71] the result until you have the equation in POS form, and finally, 6) use the derived POS equation to re-implement the circuit. Here we go.

1) Write out the equation implemented by the circuit. The circuit is in SOP form; from the circuit, you can see that you'll have two product terms (two AND gates) that are logically added together (one OR gate). The resulting equation is:

$$F(A,B,C) = A \cdot \overline{B} + \overline{A} \cdot C$$

2) Although this equation is officially in SOP form, we need to make this look like a more familiar SOP form (standard SOP form) in order to transfer the equation to a truth table. The problem right now is that both of the product terms are missing an independent variable, which we'll need to add. The way we'll do this is to logically multiply the equation by '1'. Thinking back to the original Boolean algebra theorems, you'll find that: $x + \overline{x} = 1$. Note the first product term is missing the C variable. We'll add it by multiplying the first product term by $(C + \overline{C}) = 1$ which does not alter the value of the product term. The following equations list the entire procedure for both product terms.

[71] To "DeMorganize" means to apply DeMorgan's theorem. This term was coined by the infamous Professor Freeman Freitag sometime in the mid-1980s.

$A \cdot \overline{B} = A \cdot \overline{B} \cdot (C + \overline{C})$ $A \cdot \overline{B} = A \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C}$	$\overline{A} \cdot C = \overline{A} + C \cdot (B + \overline{B})$ $\overline{A} \cdot C = \overline{A} \cdot C \cdot B + \overline{A} \cdot C \cdot \overline{B}$ $\overline{A} \cdot C = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C$

Here is the final equation:

$$F(A,B,C) = A \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C$$

3) Now that the terms look familiar, we enter them into a truth table. Note that a '1' is placed in the F column for the corresponding product terms in the equation derived in the previous step of this solution.

A	B	C	F
0	0	0	0
0	0	1	1 $(\overline{A} \cdot \overline{B} \cdot C)$
0	1	0	0
0	1	1	1 $(\overline{A} \cdot B \cdot C)$
1	0	0	1 $(A \cdot \overline{B} \cdot \overline{C})$
1	0	1	1 $(A \cdot \overline{B} \cdot C)$
1	1	0	0
1	1	1	0

4) The next step is to solve for the complemented output. We do this by the long way by adding a complemented F column to the previous truth table. From the table below we can write an SOP equation for the complemented output; this result appears below the following table.

A	B	C	F	$\overline{F}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

$$\overline{F}(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot\overline{C} + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

5) The final equation is an expression for F-bar (another way of saying a complemented F). We want an expression for F (as opposed to F-bar) so we must complement both sides of the equation and DeMorganize the result a bunch of times. The following equations list these steps.

$$\overline{F}(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot\overline{C} + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

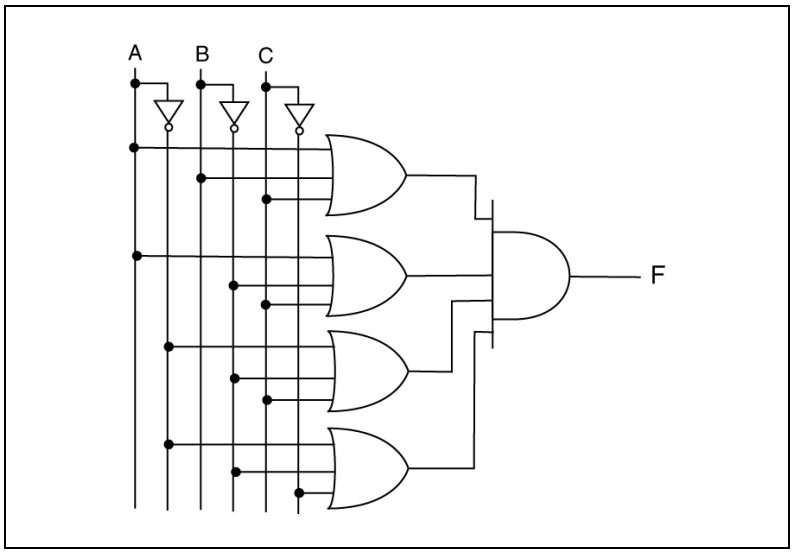
$$\overline{\overline{F}}(A,B,C) = \overline{\overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot\overline{C} + A\cdot B\cdot\overline{C} + A\cdot B\cdot C}$$

$$F = \overline{(\overline{A}\cdot\overline{B}\cdot\overline{C})}\cdot\overline{(\overline{A}\cdot B\cdot\overline{C})}\cdot\overline{(A\cdot B\cdot\overline{C})}\cdot\overline{(A\cdot B\cdot C)}$$

$$F = (\overline{\overline{A}}+\overline{\overline{B}}+\overline{\overline{C}})\cdot(\overline{\overline{A}}+\overline{B}+\overline{\overline{C}})\cdot(\overline{A}+\overline{B}+\overline{\overline{C}})\cdot(\overline{A}+\overline{B}+\overline{C})$$

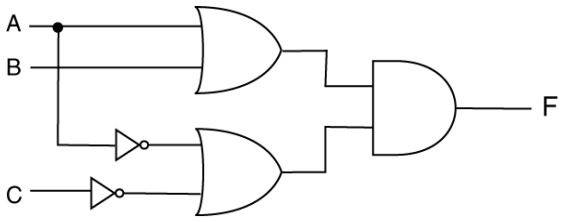
$$F = (A+B+C)\cdot(A+\overline{B}+C)\cdot(\overline{A}+\overline{B}+C)\cdot(\overline{A}+\overline{B}+\overline{C})$$

6) Finally, the last step is to draw a circuit model for the final equation of the previous step. This turned out to be a long problem as it shows many of the useful and versatile properties associated with Boolean algebra. One final thing to note in the diagram below is the fact that the AND gate has some extended wings to handle the larger number of inputs; this is typically done with any gate, as necessary when dealing with multiple inputs.



Example 6-3

Change the following circuit implementation from a POS (OR/AND) to a SOP (AND/OR) form.



Solution: The solution to this problem is similar to the solution of Example 4-7; the steps are basically the same but you need to apply them in a strange reverse order.

1) Write out the equation implemented by the circuit. The circuit is in POS form; from the circuit, you can see that you'll have two sum terms (two OR gates) that are logically multiplied together (one AND gate). The resulting equation is:

$$F(A,B,C) = (A+B)\cdot(\overline{A}+\overline{C})$$

2) We need to put the above equation into SOP form so we can easily enter its information into the truth table. If we complement both sides of the equation and then DeMorganize it, we'll get an expression for F-bar in SOP form.

$$F = (A+B)\cdot(\overline{A}+\overline{C})$$

$$\overline{F} = \overline{(A+B)\cdot(\overline{A}+\overline{C})}$$

$$\overline{F} = \overline{(A+B)} + \overline{(\overline{A}+\overline{C})}$$

$$\overline{F} = \overline{A}\cdot\overline{B} + A\cdot C$$

3) From here, we need to expand each of the product terms to include each of the independent variables. We'll use the same technique as before.

$$\overline{F} = \overline{A} \cdot \overline{B} + A \cdot C$$

$$\overline{F} = \overline{A} \cdot \overline{B} \cdot (C + \overline{C}) + A \cdot C \cdot (B + \overline{B})$$

$$\overline{F} = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot C + A \cdot \overline{B} \cdot C$$

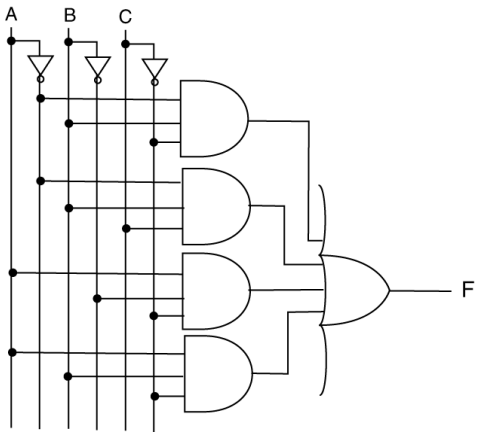
4) The equation above tells us where the 0's live in the truth table. If we know where the 0's live, we also know where the 1's live and that is what we're looking for in order to give us an equation for this function in SOP form.

A	B	C	F	$\overline{F}$
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

5) Now we can write an equation for F.

$$F = \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C}$$

6) The final step is to draw a model for the final circuit implementation.



Example 6-4: Half Adder in POS Form

Provide a circuit diagram for a half-adder (HA) implemented in POS form..

Solution: We all know that a half adder is a one-bit adder with two inputs (the bits being added) and two outputs (a sum and a carry-out). This being the case, we can borrow much of our results from the SOP version of the problem we did in a previous chapter.

Step 1) Define the Problem: As with all problems, draw a black box diagram of the final circuit; Figure 5.11 shows this result. Table 5.6 and Table 5.7 show the original truth table and the truth table including the complemented outputs, respectively.

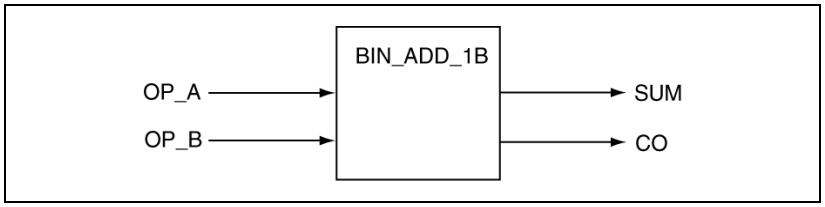


Figure 6.6: The black-box diagram for the example problem.

OP_A	OP_B	SUM	CO
0	0		
0	1		
1	0		
1	1		

Table 6.3: The original truth table.

OP_A	OP_B	SUM	!SUM	CO	!CO
0	0	0	1	0	1
0	1	1	0	0	1
1	0	1	0	0	1
1	1	0	1	1	0

Table 6.4: The truth table including complemented outputs.

Step 2) Describe the Solution: Note that for this problem, you'll need to generate two Boolean expressions: one for the SUM and the other for the CO. For this problem, we'll use SOP form since the problem statement did not specify a preference. The final equations are written by logically summing the product terms associated with rows in which 1's appear.

$$\overline{SUM} = (\overline{OP\_A} \cdot \overline{OP\_B}) + (OP\_A \cdot OP\_B)$$

$$\overline{CO} = (\overline{OP\_A} \cdot \overline{OP\_B}) + (\overline{OP\_A} \cdot OP\_B) + (OP\_A \cdot \overline{OP\_B})$$

Equation 6-1: The starting equations for Example 6-4.

$$\overline{SUM} = (\overline{OP\_A} \cdot \overline{OP\_B}) + (OP\_A \cdot OP\_B)$$

$$\overline{\overline{SUM}} = \overline{(\overline{OP\_A} \cdot \overline{OP\_B}) + (OP\_A \cdot OP\_B)}$$

$$SUM = \overline{(\overline{OP\_A} \cdot \overline{OP\_B})} \cdot \overline{(OP\_A \cdot OP\_B)}$$

$$SUM = (OP\_A + OP\_B) \cdot (\overline{OP\_A} + \overline{OP\_B})$$

Equation 6-2: The SUM path from SOP to POS for Example 6-4.

$$\overline{CO} = (\overline{OP\_A} \cdot \overline{OP\_B}) + (\overline{OP\_A} \cdot OP\_B) + (OP\_A \cdot \overline{OP\_B})$$

$$\overline{\overline{CO}} = \overline{(\overline{OP\_A} \cdot \overline{OP\_B}) + (\overline{OP\_A} \cdot OP\_B) + (OP\_A \cdot \overline{OP\_B})}$$

$$CO = \overline{(\overline{OP\_A} \cdot \overline{OP\_B})} \cdot \overline{(\overline{OP\_A} \cdot OP\_B)} \cdot \overline{(OP\_A \cdot \overline{OP\_B})}$$

$$CO = (OP\_A + OP\_B) \cdot (OP\_A + \overline{OP\_B}) \cdot (\overline{OP\_A} + OP\_B)$$

Equation 6-3: The CO path from SOP to POS for Example 6-4.

Step 3) Implement the Solution: The final step involves translating the Boolean expressions listed in Equation 6-2 and Equation 6-3 into circuit form. Figure 5.12 shows the final gate-level implementation.

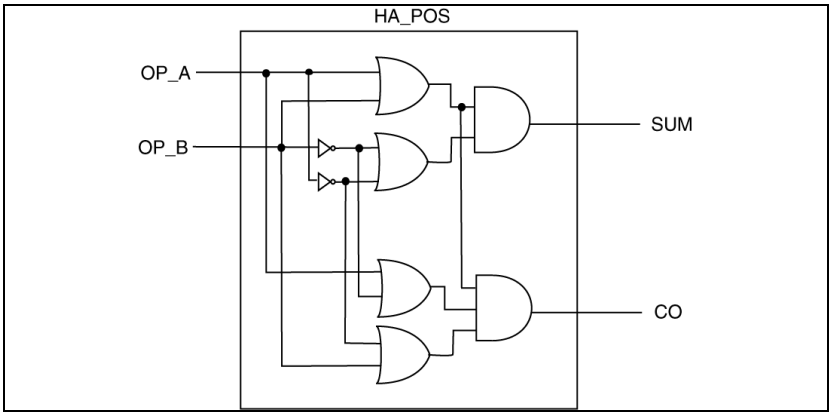


Figure 6.7: The circuit representation of the final solution for Example 6-4.

Finally, it's instructive to list all the solutions we know about thus far. Equation 6-4 lists both the SOP and POS forms for the CO output while Equation 6-5 lists the SOP and POS forms for the SUM output.

Note once again that the SOP and POS forms for a given output are functionally equivalent. Finally once again, Figure 6.8 shows a comparison of the final circuit implementations for both the SOP and POS versions of the half adder.

$$CO = OP\_A \cdot OP\_B$$

Is functionally equivalent to:

$$CO = (OP\_A + OP\_B) \cdot (OP\_A + \overline{OP\_B}) \cdot (\overline{OP\_A} + OP\_B)$$

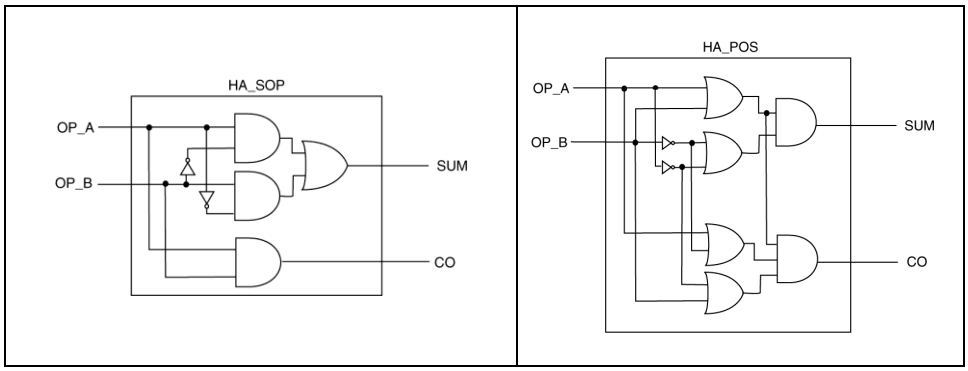
Equation 6-4: The CO path from SOP to POS for Example 6-4.

$$SUM = \overline{OP\_A} \cdot OP\_B + OP\_A \cdot \overline{OP\_B}$$

Is functionally equivalent to:

$$SUM = (OP\_A + OP\_B) \cdot (\overline{OP\_A} + \overline{OP\_B})$$

Equation 6-5: The CO path from SOP to POS for Error! Reference source not found.**.**



(a) (b)

Figure 6.8: A side-by-side comparison of the SOP (a) and POS (b) circuit diagrams for the half adder.

Chapter Summary

- **DeMorgan's Theorem:** One of the basic theorems in digital design typically used to translate from one form to other functionally equivalent forms. Boolean expressions can be simplified using DeMorgan's theorem also. There are two different forms of DeMorgan's theorem; both bring ultimate bliss to the user.

- **SOP and POS Representations:** Two of the most common ways to represent Boolean functions are using sum-of-products (SOP) and product-of-sum (POS) forms. DeMorgan's is typically used to generate a POS equation from a truth table. The SOP form is characterized by multiple product terms that are logically summed together while the POS form is characterized by sum terms that are logically multiplied together.

- **A function can be represented by a truth table in two ways;** either the positive version of the output is presented (a representation of a non-complemented output variable) or the negative version is presented (a representation of the complemented output variable). These two outputs are complements, or inversions, of each other.

Chapter Exercises

1) Generate a Boolean equation that is equivalent to each of the following truth tables in POS form.

B2	B1	B0	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(a)

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(b)

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(c)

t	u	v	F1	F2
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

(d)

2) Convert the following functions to POS form

a) $F(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C}$

b) $F(A,B,C) = \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot\overline{B}\cdot\overline{C} + A\cdot\overline{B}\cdot C + A\cdot B\cdot C$

c) $F(X,Y,Z) = \overline{X}\cdot\overline{Y}\cdot Z + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot Y\cdot\overline{Z} + X\cdot Y\cdot Z$

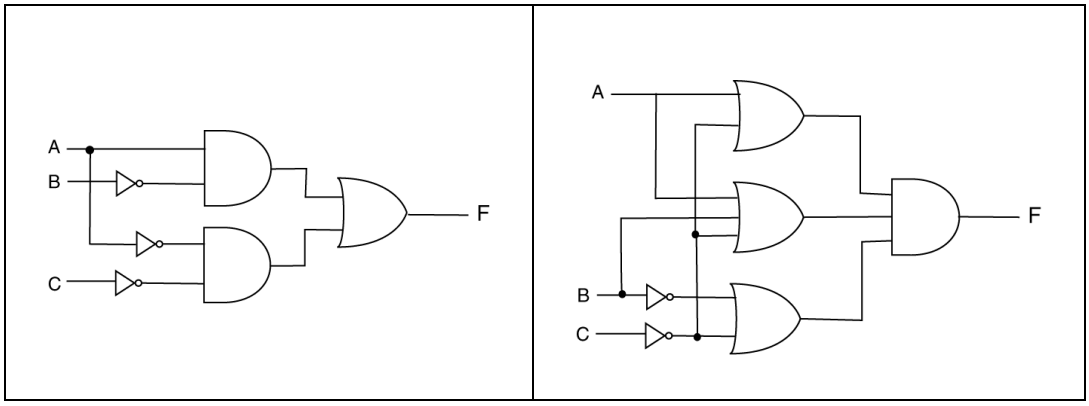
3) Convert the following functions to SOP form.

a) $F(R,S,T) = (\overline{R}+\overline{S}+\overline{T})\cdot(\overline{R}+S+\overline{T})\cdot(\overline{R}+S+T)\cdot(R+\overline{S}+T)\cdot(R+S+T)$

b) $F(A,B,C) = (A+B+C)\cdot(A+\overline{B}+C)\cdot(A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)$

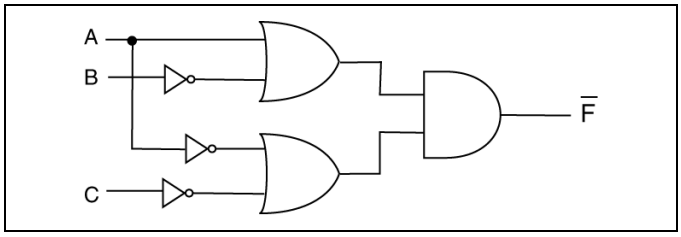
c) $F(X,Y,Z) = (\overline{X}+\overline{Y}+Z)\cdot(\overline{X}+Y+\overline{Z})\cdot(X+\overline{Y}+Z)\cdot(\overline{X}+\overline{Y}+\overline{Z})$

4) For the following circuit diagram, change the form from SOP to POS form.



(a) (b)

5) For the following circuit, change the circuit to a have an output for F in SOP form.



6) Draw a circuit representation for the following Boolean equations:

$F(R,S,T) = (\overline{R}+\overline{S}+\overline{T})\cdot(\overline{R}+S+\overline{T})\cdot(\overline{R}+S+T)\cdot(R+\overline{S}+T)\cdot(R+S+T)$

Design Problems

1) Being that SOP and POS forms are functionally equivalent, describe a few reasons why you would want to use one form over the other.

2) Design a circuit that has four inputs and three outputs. The four inputs are considered to be two 2-bit inputs. One output consider the two inputs to be binary numbers and indicates when the two input number are not equivalent. The other output considers the two inputs to be stone-age binary inputs and indicates when the two binary inputs are equivalent. The third output indicates when the previously described outputs are both in an "on". For this problem, implement the first two outputs using POS forms; implement the third output in any way you deem appropriate, but minimize your use of gates in the implementation.

3) Design a circuit that has four inputs and four outputs. Each input is from a switch that is associated with one of four doors to a room; the outputs control a locking device on each door. There are four different sets of people who need to get into the room but you need to control exactly who gets into the room. Consider the each door to be named A, B, C, or D. Design a circuit that allows the following control (don't worry about how people are going to get out of the room). Provide a model of your circuit using POS form.

- If someone wants in door A, that person always gets in and is always the only person that gets in unless door C wants in also, in which case both door A and C will be opened.
- If some one wants in door B, that person can only get in if someone at door D wants in also. In this case, both door B and D will open.
- The person at door C can never be in the room alone but can be in the room with anyone else.

7 Chapter Seven

(Bryan Mealy 2012 ©)

7.1 Introduction

The previous chapters provided a foundation of digital design. These are the first few steps into a special world; you can travel as deeply into digital-land as you desire. We are still early in the "design" process and the designs are relatively simple. Moreover, the approach we have taken to design is not used much due to the fact its severe limitations.

The truth is that this is not a perfect world but the entire world will expect your designs to work perfectly. Half the battle in the actual implementation of any design is the notion that your design will need modifications along the way in order to ensure the design successfully completes the task it set out to do. This means that you're going to make mistakes along the way. This fact leaves you with two options, both of which you'll find yourself taking: 1) make sure you understand all the parameters before you start the design, and, 2) fully test the design at many stages along the way and particularly when the design is completed. The main topic of this chapter is to introduce timing diagrams, a mechanism to facilitate both of these objectives. You're going to make mistakes when you design; timing diagrams are going to help limit the number of mistakes you make and also help you and/or anyone who's working with your design understand what's going on.

Timing diagrams represent both a design tool and a test tool. This means that timing diagrams can both specify designs and test designs. While it is conceivable that the world can get by without ever seeing a timing diagram, they are massively helpful, particularly as your designs become more complex. Timing diagrams provide a visual representation of what your circuit should be doing (the design process) or what your circuit is actually doing (testing). Either way, timing diagrams are massively helpful. If there ever were a notion that proved that a picture is work a thousand words, it would definitely be in regards to timing diagrams.

Main Chapter Topics

- **TIMING DIAGRAMS:** The operation of digital circuits is often specified, explained, and/or modeled with timing diagrams. Timing diagrams provide both an initial design tool as well as a method to verify the proper operation of completed circuits and are thus an integral part of any meaningful digital design. This chapter introduces timing diagrams and describes their relation to digital circuits.

Why This Chapter is Important

This chapter is important because it describes the use of timing diagrams to model typical digital circuit operations.

7.2 Timing Diagram Overview

We currently have several methods to model digital circuits including truth tables, circuit diagrams, and written circuit descriptions. Although these representations are considered 100% accurate descriptions of the circuits, they are somewhat timeless in nature. This "timelessness" forms somewhat of an artificial representation of a circuit in that digital circuits actually operate over given periods of time[1]. As digital circuits become more complex, it becomes increasingly harder to imagine how the circuit operates over a given span of time[2].

A digital circuit operates over a given time span. During these time spans, the circuit's outputs "adapt" to changes in the circuit inputs. In other words, the circuit's inputs are generally expected to change; when this change occurs, the circuit's outputs must respond such that they continue to match the specifications for a given set of inputs. A digital circuit's outputs react dynamically to the circuits inputs. Yet another reason why digital circuits are widely considered massively exciting.

Timing diagrams detail a digital circuit's operation over an arbitrary period of time. Because of this, timing diagrams are massively important in digital design-land for two main reasons. Firstly, timing diagrams are able to specify and/or model digital circuit operation[3]. Secondly, timing diagrams are used to verify that digital circuits are operating as specified either by using some type of simulator or examining the waveform output from the actual circuit. For now, in a written text such as this one, we'll only be dealing with the first item. When you're actually designing and implementing circuits, you'll be living intimately with the second item as you make working with simulators and working with debuggers a part of your life.

To drive the point home even further… if your circuit does in fact simulate properly, the next step in the design process is to implement the circuit. You'll find out that just because your circuit simulates properly does not guarantee that the physical circuit will work properly. If you implement the circuit, and it does not work, the next step is to figure out why it does not work. You generally don't go back to the simulator as you have probably previously verified that that circuit simulated properly.

Your next step is to analyze the actually hardware implementation. This is often done by using a happy device known as a logic analyzer, or LA. As you will see, one of the two main types of outputs from the LA is a timing diagram-type display. The LA is a magical device that allows you to view signals in your circuit of your choosing for designated windows of time. The LA is a massively useful hardware debugging tool used by all competent digital designers. Don't fear the LA.

There is some special terminology and symbology typically used in timing diagrams; we'll be going over a few of the more important ones in this chapter. You'll find out that although there are many ways to represent timing diagrams, the concepts of timing diagrams and their relation to digital circuits is not complicated. Once you understand the basics of timing diagrams, you'll be able to quickly adjust to any timing diagram symbology that you've not seen or used before. There is somewhat of a learning curve involved, but if you put the time into becoming comfortable with timing diagrams, they'll help you become a better digital designer.

[1] It takes time for the electrons to move around in the underlying sillycone. Keep in mind that nothing is instantaneous in actual digital circuits although we typically can model signal changes in circuits as being so.
[2] As you'll find out later, there are two basic types of circuits. The notion of "time" relative to a circuit becomes more complicated when the circuits outputs are a function of something other than the circuit's inputs.
[3] More often, only the important parts of the circuit are specified. In this context, "important" could have many meanings. As we travel deeper into digital design land, these meanings start to surface.

7.3 Timing Diagrams: The Gory Details

Jumping right into it, Figure 7.1 shows five timing diagrams serving as an introduction to the flavor of most timing diagrams you find out in digital-land. The items below provide an extended description and comments regarding each of the timing diagrams in Figure 7.1. Keep in mind that the horizontal axis is the time axis in each of these timing diagrams[4]. For this example, we only use the term "time" but we have attached no metric such as "seconds" or "milliseconds" to the time. At this point in your digital design careers, we can satisfy ourselves by talking about time in general[5].

(1) Note that all the timing diagrams show a "functional" relationship; that is, at any given time, a given signal is either high, or low, but never both at the same time.

(2) This timing diagram shows line that seems to go randomly up and down. The line represents the value of digital signal in question. The signal is generally given a name, but we've left it out in order to keep this discussion general. Note that the signal has two values, which is what you would expect from a digital signal. The signal shows various transitions from high-to-low and low-to-high; these signal changes are referred to as toggles in digital lingo.

(3) This timing diagram explicitly shows the two values of the signals. In this case, the two values are referred to as '**H**' and '**L**', which represent the high and low values of the digital signals, respectively. Also included in this timing diagram are the horizontal dotted lines, which support the notion that the digital signal is either high or low[6]. These dotted lines are often omitted but are often included in more "busy" timing diagrams in order to increase the readability and understandability of the timing diagram.

(4) This timing diagram is similar to the timing diagram of (b) but the '**H**' and '**L**' have been replaced with '1' and '0', respectively. This emphasizes the point that the two values of the digital signals are actually models representing some actual digital hardware. There are many flavors of digital hardware out there; these flavors can differ in the voltage levels used to drive the hardware. We opt to ignore voltage concerns by abstracting our digital designs to a higher level such that we don't need to deal with voltage levels. In other words, we're describing the digital operation of a circuit; voltage levels are an issue we can often not worry about.

(5) This is another common style of modeling digital signals. While the previous timing diagrams used vertical lines to represent high-to-low and low-to-high transitions, the style of this timing diagram uses slanted lines. Note that the lines are always slanted in the direction of advancing time. In reality, the signals in a digital circuit cannot instantaneously change value as they seem to do in the other listed timing diagrams, but it is sometimes helpful to include the slanted transitions in timing diagrams. It is generally more common in digital design-land to model the signal transitions as occurring instantaneously.

(6) The final timing diagram is nothing new. What we want to do is use this timing diagram to toss some typical timing diagram lingo at you. At (a) in the timing diagram shows that the given signal is initially low at the beginning of the timing diagram. At (b), the signal switches from a

[4] The horizontal axis is always the time axis; if it's not, rotate the text 90 degrees.

[5] The notion of exact time increments on the horizontal axis are somewhat less important at this point than the notion of understanding what exactly the timing diagram is attempting to show. If completely understand the basic function of timing diagrams, the addition of exact time measurements is trivial. We'll be dealing other issues of timing diagram timing issues as they become more appropriate in later chapters.

[6] This is primarily a mechanism to help the person reading the timing diagram figure out what is going on. This becomes important in complex designs where you need to list a page full of signals in order to verify your design is working correctly. After staring at a page full of signals, the "highness" and "lowness" of signals is sometime obfuscated due to brain overload.

low state to a high state, or more simply stated, the signal toggles. At (c), the signal switches from a high state to a low state, or once again, toggles[7]. Around the time indicated by (d), the signal toggles two times (similar to (b) and (c)). At (e), the timing diagram ends with the signal in a low state. In addition, possibly most importantly, we want to express the notion that the more comments you place on a timing diagram, the happier the human reader will be.

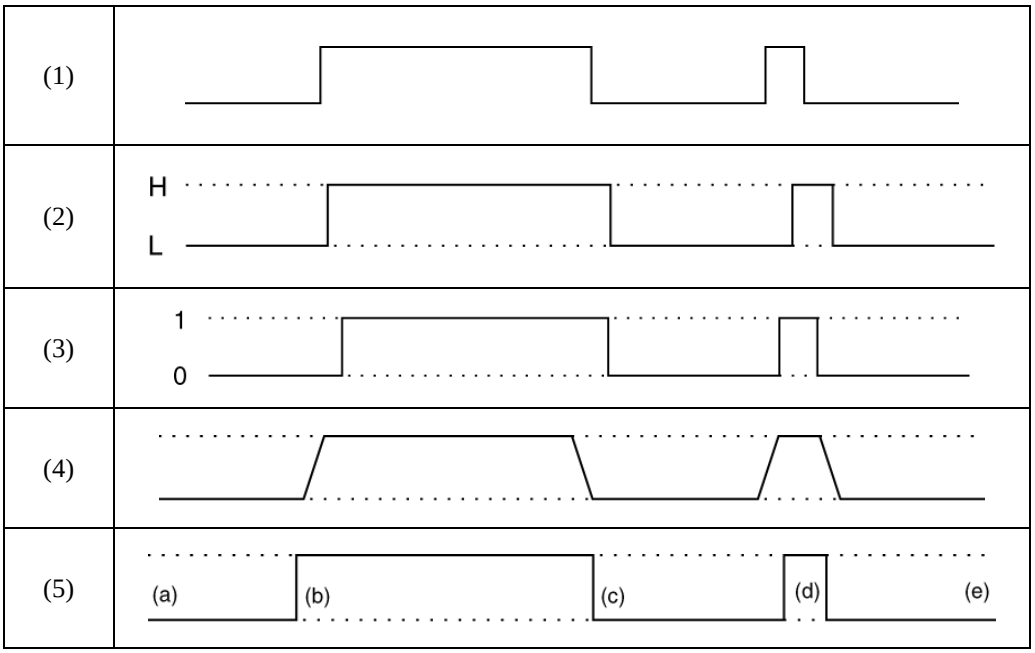


Figure 7.1: Example timing diagrams.

Finally, as we alluded to in the description of the timing diagram in Figure 7.1(5), including notes on timing diagram is something you should always do. Nothing looks crappier than a page filled with timing diagrams but having no notes included describing to the reader exactly what they are looking at. The unstated rule for all timing diagrams is that they should include notes to describe what is important in that specific timing diagram. Stated differently, timing diagrams are either trying to show you something, or they are being used by you to show something to other people. That being said, make sure that you always include notes, or "annotations" in your timing diagrams; you'll find yourself hoping that the person who provided the timing diagram you're being forced to stare at provided you with the same notes.

And really finally, there are many approaches to annotating timing diagrams; we'll get to other approaches in later chapters. The truth is that there is no correct way to annotate a timing diagram, but the following list provides some reasonably intelligent guidelines.

- The overall purpose of any diagram, including timing diagrams, is to quickly present information. Providing annotations facilitates the understanding of the underlying circuit. If you make the timing diagram clear, you've served your purpose.
- Make sure you draw the reader's eye to the important part of the timing diagram; you can easily do this with your annotations.

[7] Toggling refers to switching states and includes both low-to-high and high-to-low transitions.

- Don't try to express too many ideas in one timing diagram. A better approach is to make multiple timing diagrams, each with its own succinct point.
- Only include the signals and information in timing diagrams that help you get your point across; you should strive to omit unused or unimportant.
- The time-span for timing diagram should only include information that helps you solidify your point. The act of including too large of a time-slice diverts the focus away from what you're trying to show.
- All timing diagrams (and all diagram, for that matter) should include a title that quickly describes what the timing diagram is trying to show.

7.4 Timing Diagrams: The Initial Real Stuff

One use of timing diagrams is to model the operation of digital circuits and devices. Since we are currently familiar with three digital devices, let's demonstrate their operation with timing diagrams. Figure 7.2 show an inverter and an associated timing diagram. The signal names '**x'** and '**F**' represent the input and output to the inverter, respectively (as listed in the top diagram of Figure 7.2). The upper signal in the timing diagram is labeled 'x'; the timing diagram shows how the x signal acts as a function of time. The signal activity shown in 'x' line is arbitrary; the intent of this timing diagram is to show the changes in the output (F) as a function of the input (x)[8]. Figure 7.2 does indeed show the complimentary relationship between the input and output, as you would expect from an inverter.

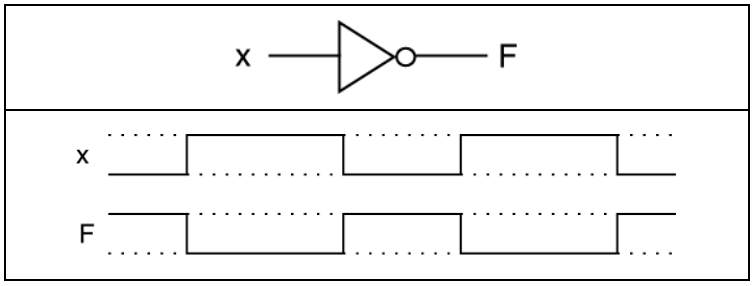
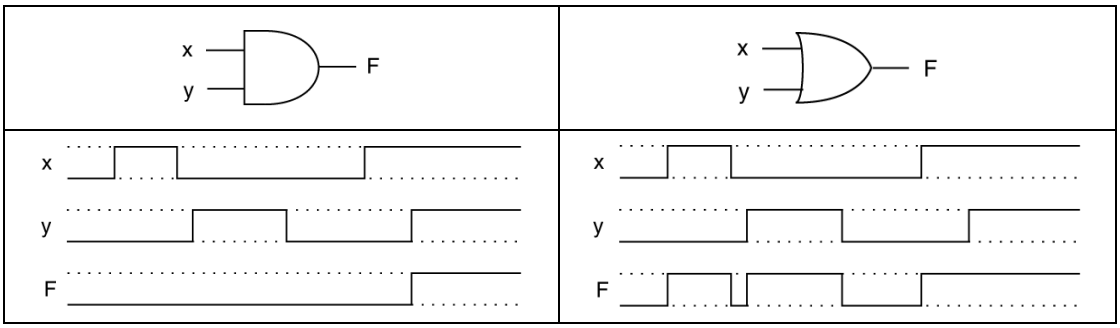


Figure 7.2: Example timing diagram for inverter.

Figure 7.3 shows example timing diagrams for AND and OR gates. Note that in Figure 7.3(a), the output is only high when of both the 'x' and 'y' inputs are high. Likewise, Figure 7.3(b) shows that for an OR gate, the output is only low when both of the 'x' and 'y' inputs are low. The two timing diagrams of Figure 7.3 once again match our previous description of AND and OR gate operation. Moreover, the timing diagrams in Figure 7.3 completely describe the operation of AND and OR gates; recall that truth tables were previously used to described these gates[9]. Keep in mind that for both timing diagrams in Figure 7.3, the value of the input variables is arbitrary; there is nothing special about the input timing. The outputs, of course, are dependent upon the inputs.

[8] Keep in mind that once again shows the true functional relationship between the input (the independent variable) and the output (the dependent variable). In other words, for any one given instance of time, the output is necessarily high or low (but never both).

[9] It's up to you decide which is better. Generally speaking, sometimes one description is better than another based on the context of what you're describing.



(a) (b)

Figure 7.3: Example timing diagrams for an AND gate (a) and an OR gate (b).

For our final example, let's generate a timing diagram for the main example problem from the previous chapters. Figure 7.4 shows the truth table associated with a previous example while Figure 7.5 shows an example timing diagram associated with the truth table. Note that the timing diagram includes the three inputs and one output listed in the truth table. The input signal characteristics in Figure 7.5 are once again arbitrary.

Figure 7.5 uses some special notation to indicate that the timing diagram does indeed reflect the characteristics of the associated truth table. The vertical dotted lines in Figure 7.5 represent particular moments in time. At each of these moments in time, the index into the truth table provides an aid in your perusal of the timing diagram. For example, the (1) label indicates a match between the second row in the truth table where B2='0', B1='0', and B0='1'. Under these particular signal conditions, the output is a '0'.

index	B2	B1	B0	F
(0)	0	0	0	0
(1)	0	0	1	0
(2)	0	1	0	0
(3)	0	1	1	0
(4)	1	0	0	0
(5)	1	0	1	1
(6)	1	1	0	1
(7)	1	1	1	1

Figure 7.4: The truth table for the original problem design problem.).

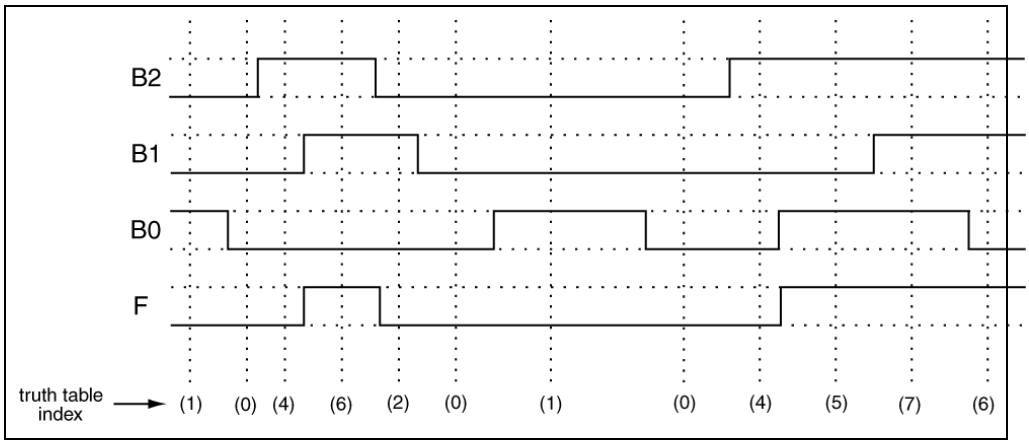


Figure 7.5: Timing diagram for main problem specified in this chapter.

Here are a few more comments regarding timing diagrams.

1. The vertical dotted lines in Figure 7.5 do not overlap any transitions on the input signals. Technically speaking, the "vertical" transitions in the signals indicate a discontinuity[10] in the signal. The truth here is that you're not being told the entire story regarding timing diagram. Later chapters reveal the truth after you prove your worthiness to the digital gods.

2. The input signals B0, B1, and B2 are completely arbitrary. In this particular timing diagram, there happens to be every possible combination of the three inputs. This won't always be the case, but since it is, the timing diagram of Figure 7.5 completely describes the given function. The given function would not be completely specified if one or more particular sets of combinations of the inputs were missing.

7.5 Timing Diagrams: Bundle Notation

As every digital designer knows, the name of the game in the real world is to constantly transform things from one form, to an equivalent form that is simpler. This is particularly true with timing diagrams because they have tendency to become unwieldy and thus unreadable. One way to control this added complication is to exploit the common purpose of some signals by placing them into a special group. The resulting grouping of signals makes a given design easier to understand; the associated timing diagram is also easier to analyze.

Out there in digital-land, the term "bus" refers to a group of signals. In reality, the term bus has multiple definitions in digital land[11], so the more appropriate term for what we're describing here is a "bundle". I'll keep writing bundle, and thus keep fighting off my tendency to use the word bus. You need to get used to the terms "bundle notation" and "bus notation" as digital design uses these terms quite often. This section covers the notion of using bundle notation.

The use of bundle notation appears in three areas in digital design: in schematic diagrams, timing diagrams, and VHDL models. None of these uses are overly complicated, though the standards for

[10] It's one of those calculus terms. Please refer to your bulky math book for clarification.

[11] The term "bus" often refers to a "protocol", which is essentially a pre-defined set of rules that describe a mechanism that digital entities can use to communicate with each other. Additionally, you often see the terms bus and protocol used interchangeably.

bundle notation in timing diagrams is somewhat less standard than they are in schematic diagrams. The general idea is that if you understand the general concepts of bundle notation, nothing much can cause you confusion.

7.5.1 Bundle Notation in Schematic Diagrams

This is a straightforward concept so we'll not spend a lot of time here. The issue at hand is to simplify block diagram and/or schematic diagrams by bundling signals. The use of "slash notation" allows us to do this quite easily; Figure 7.6 shows a few examples. As advertised, a set of signals is bundled; a forward slash indicates a bundled signal and a number indicates how many signals are in the bundle. Some specific information regarding Figure 7.6 and slash notation appear below. Figure 7.6(a) shows the original diagram while the other components of Figure 7.6 show some examples.

- Figure 7.6(a) shows the original block diagram indicating a black box with three inputs and one output. The inputs may be related and can thus be bundled. Note that in each of the subsequent bundles, some small amount of information is lost (namely, the names of the individual signals) in an effort to make the diagram less busy.

- Figure 7.6(b) shows one approach to bundling. This diagram shows an attempt was made to preserve the names of the signals. The slash on the "**B_210**" line indicates that the **B_210** signal is now a bundle and that it contains three individual signals as indicated by the tiny "**3**" near the slash mark.

- Figure 7.6(c) shows an approach that attempts to save even less information than Figure 7.6(b) by using "**B**" instead of "**B_210**". Once again, the diagram presents less information, but there is less clutter in the resulting circuit model.

- Figure 7.6(d) shows yet another approach to bundling; in this case, the signal name also indicates how many signals are associated with the bundle. You see this sometimes, but it is not super clear what the "**_3**" is attempting to indicate. As a result, it is questionable how much the "**_3**" actually helps.

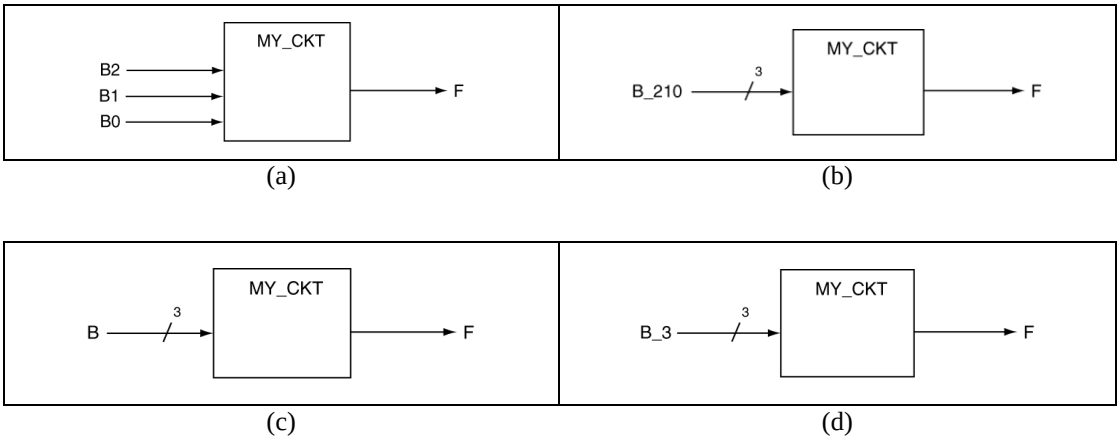


Figure 7.6: Various diagrams showing schematic-based bundling using slash notation. .

The general idea is to use bundling to make your diagrams more readable. However, you need to be careful here, as tossing every signal into a bundle does not always make sense. Figure 7.7 shows an

example where bundling does not make sense. The diagram in Figure 7.7(a) shows a half adder while Figure 7.7(b) shows an attempt to bundle both the inputs and outputs on the device model. The result is a cleaner looking diagram, but… this is a total failure.

The problem with bundling the signal in Figure 7.7(b) is that both the input and output signals are distinct; thus placing them into a bundle has made the diagram more confusing. For example, we know the half adder is a 1-bit adder, but from the Figure 7.7(b) it appears to be some flavor of two bits. Recall that the idea behind bundling is to make the resulting diagrams more readable to humans. The example in Figure 7.7 has completely failed on this mission. The moral of this story is to always make sure whatever you're doing makes things easier to read and understand; simply "looking better" does not necessarily support "being better" because being better is all about making something more understandable.

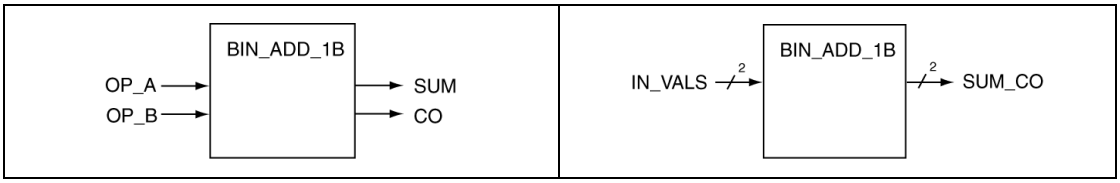


Figure 7.7: Various diagram showing schematic-based bundle notation. .

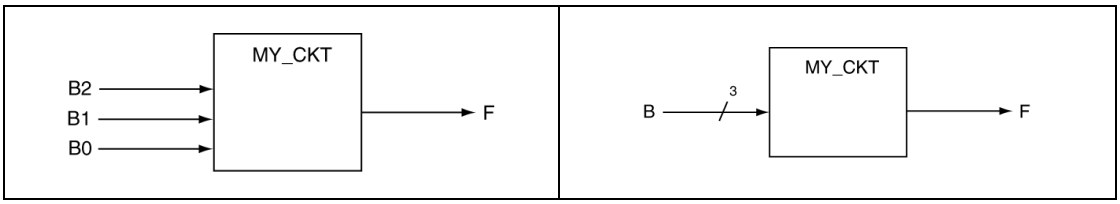
7.5.2 Bundle Notation in Timing Diagrams

Bundle notation in timing diagrams is also a relatively simple concept. Keep in mind, that there are many ways to model bundles in timing diagrams; this section will show a few of them. If you understand these, you'll be able to deal with anything new.

Figure 7.8(a) shows that same tired block diagram we've been using. What we're interested in is a timing diagram associated with Figure 7.8(b). In this case, two things have occurred; one is fairly obvious and the other is rather mysterious. The following list describes these items.

- The block diagram in Figure 7.8(b) represents an equivalent version of Figure 7.8(a), which has been simplified using bundle notation. The one bundled signal in Figure 7.8(b) replaces the three signals in Figure 7.8(a).

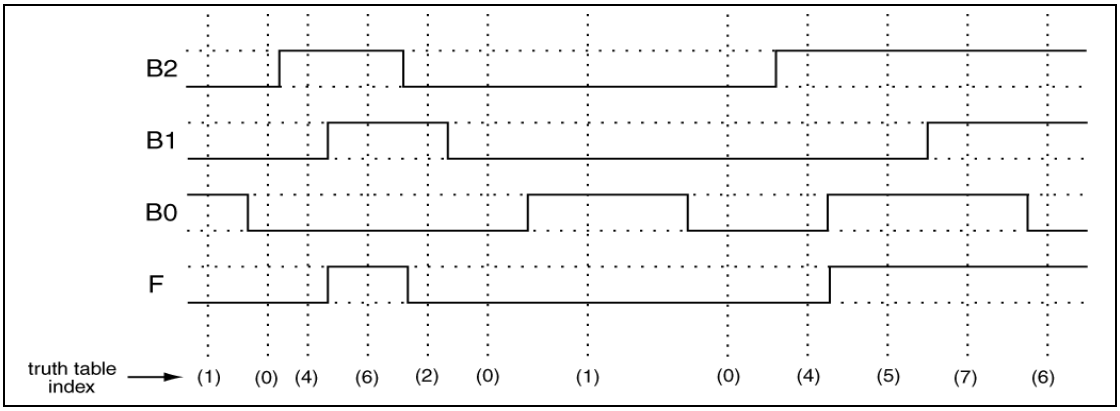
- The signal "B" in Figure 7.8(b) represents the three signals B2, B1, and B0 from Figure 7.8(a). Since the names have been changed, you've lost the notion that there may be an ordering associated with the signals in Figure 7.8(a). If this is the case, you need to state this somewhere; the two places you have to state this are in the schematic diagram or in the timing diagram. This is massively important; it's easy to make the assumption that the reader will know what you're trying to convey. You should never make any assumptions about anything in digital-land; it's always better to make note somewhere that someone will actually see it.



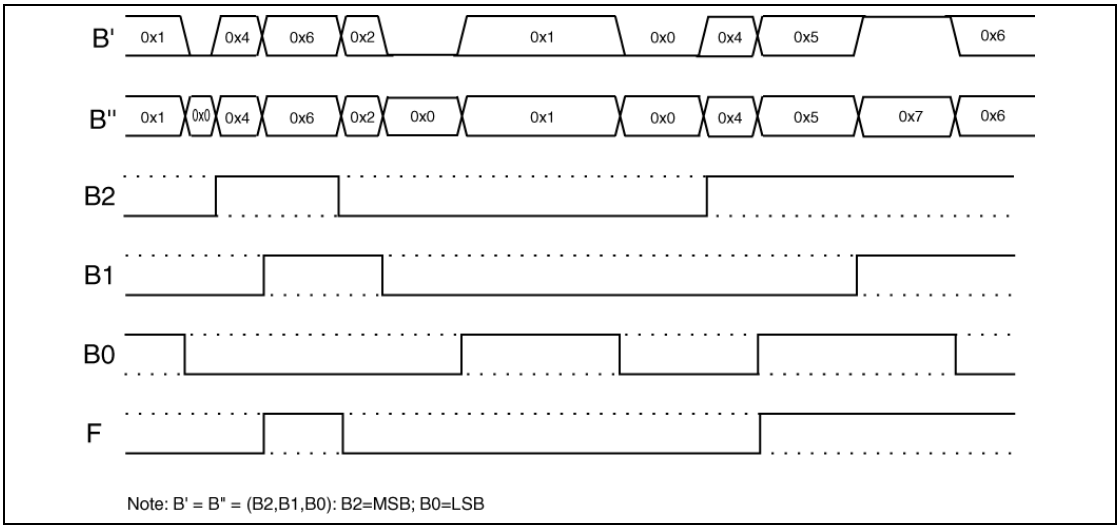
(a) (b)

Figure 7.8: Example block diagrams for the used by Figure 7.9.

Figure 7.9 shows two different but equivalent timing diagrams. The timing diagram in Figure 7.9(a) lists the individual signals while the timing diagram in Figure 7.9(b) uses two forms of bundle notation. There are a few things of interest to note here; these notes follow the diagram.



(a)



(b)

Figure 7.9: Equivalent timing diagrams showing individual signals (a) and timing-diagram-based bundle notation (b).

- In Figure 7.9(b), two parallel horizontal lines indicate that the signal is a bundle. The crosses that appear in these lines indicate that at least one of the subsequent signals in the bundle has change from either a low to a high or a high to a low.

- In Figure 7.9(b), numbers generally indicate the value of the signals within the bundle. You'll see many different ways of representing these numbers; we've opted to use a C programming language-type notation used to represent hexadecimal numbers to represent the individual signals in the bundle. Specifically, the "0x" prefix on a number indicates that you should interpret the number as a hexadecimal number. Note in the Figure 7.9(b) that every time a signal changes, there is also a change in the corresponding number representing the bundle.

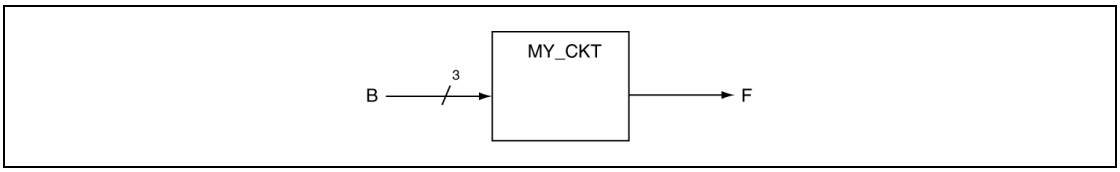
- There are only three signals associated with the bundle while hex notation can specify four bits per hex number. This is not a problem in that the missing signal(s) is always assumed to be the most significant bit (MSB) in the bundle and always assumed zero.

- The diagram should explicitly state that in the hex number used in Figure 7.9(b), the most significant bit represented is B2 while the least significant bit is B0. Notes

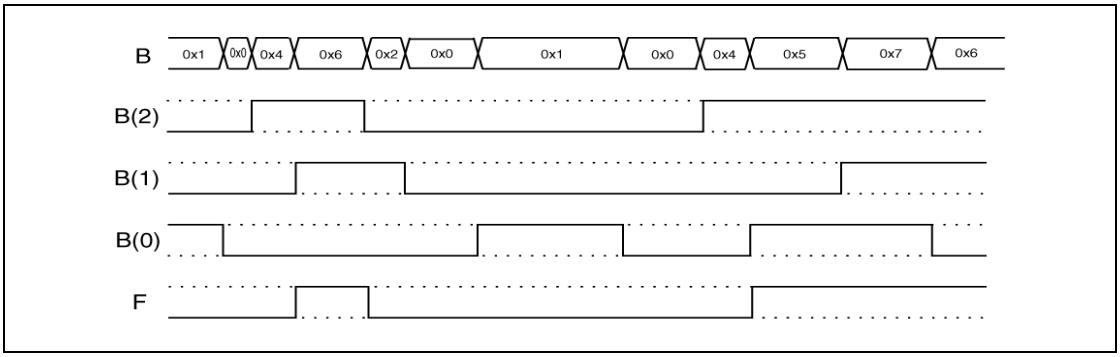
- Figure 7.9(b) show the B' and B" signals. These are equivalent signals but two different styles are used to represent their values. Often time the timing diagram drops the "parallel bar" notation when all the signals in the bundle are all high or all low. Ether approach is fine; the timing diagram in B" is more clear and consistent (one man's opinion).

One other commonly seen notation is associated with the expansion of bundles. Figure 7.10(a) shows a block diagram that includes a bundle while Figure 7.10(b) shows an associated timing diagram .The timing diagram in Figure 7.10(b) includes a "bundle expansion" of the signal labeled "B". The diagram indirectly states that bundle B is comprised of three signals (B(2), B(1), and B(0), with B(2) being he MSB and B(0) being the LSB[12]. This notation is typically used in simulators and is useful because the simulators generally allow you to expand the signals and un-expand the signals upon your whims. This notation also uses "parenthetical indexing" into the bundle and is the preferred approach as it is consistent with the syntax used in VHDL (which we'll get to later).

[12] This notation assumes that the signal with the highest index is the most significant bit. This notation is used quite commonly and it is rarely stated that B(2) is the MSB. If you're not using this approach in your timing diagrams, you need to clearly state the approach you're using in order that you don't confuse the crap out of someone.



(a)



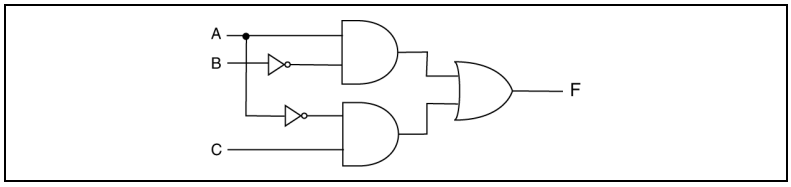
(b)

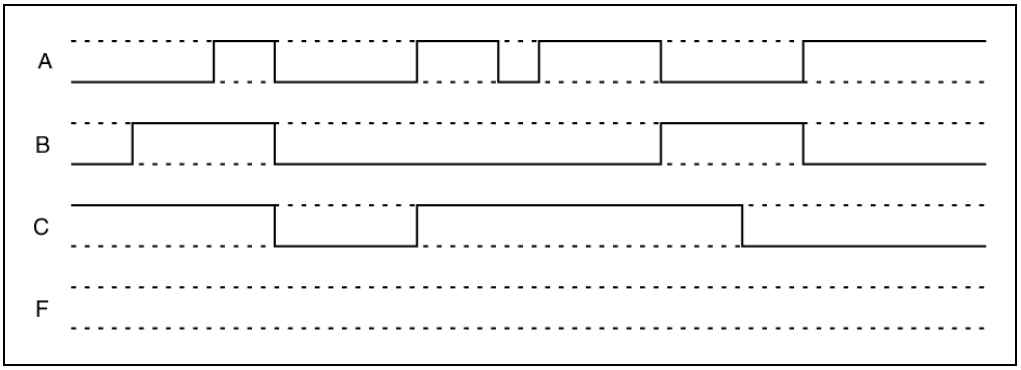
Figure 7.10: An example of bundle expansion showing parenthetical indexing on the expanded bundle.

Finally, there is simply much more to say about timing diagrams that this chapter does not mentioned. If you look at a timing diagram in any typical datasheet, you'll see lots of strange stuff there. When you actually start reading those datasheets, you should be able to understand what is going on from developing a basic understanding of timings. Hopefully, this chapter provided some measure of assistance. We can only hope.

Example 7-1

Use the following circuit to complete the accompanying timing diagram.





Solution: First, a comment. There are many ways to approach this problem; the approach listed here is definitely the long way. This solution shows you all aspects of the problem and is not necessarily the best way to solve the problem. Once you have more experience in digital design, you'll see all those other ways to solve the problem.

Step 1) Write out the Boolean equation implemented by the circuit. While we're at it, we may as well expand the equation into standard SOP form which will help us complete the truth table.

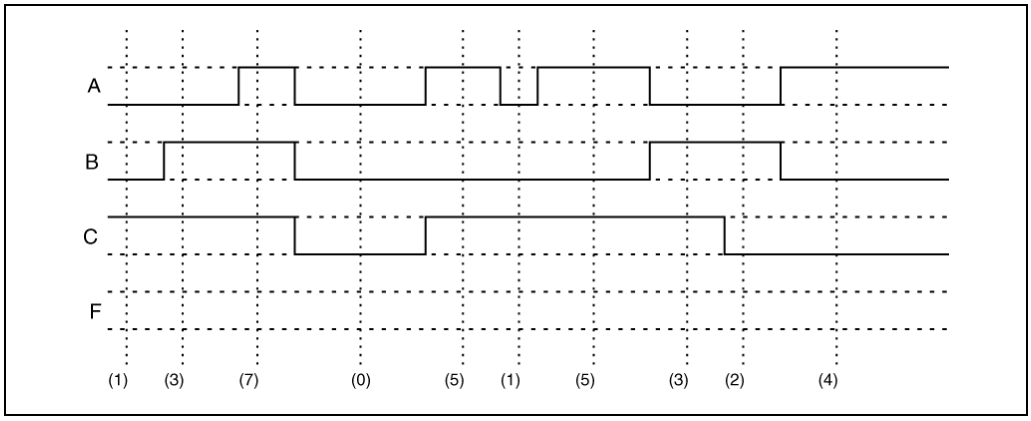
$$F = A\overline{B} + \overline{A}C$$
$$F = A\overline{B}(C + \overline{C}) + \overline{A}C(B + \overline{B})$$
$$F = A\overline{B}C + A\overline{B}\overline{C} + \overline{A}BC + \overline{A}\overline{B}C$$

Step 2) Generate a truth table and fill in a '1' for the output associated with each of the listed standard product terms. The index values are included here as it may help us out later. One super massively important point in this problem is that the problem never stated which of in the inputs was the most significant bit. In this problem, not stating this information will not change the answer. But, since we have decided to list the problem using a numeric index, we must state that input A is the MSB while input C is the LSB[13].

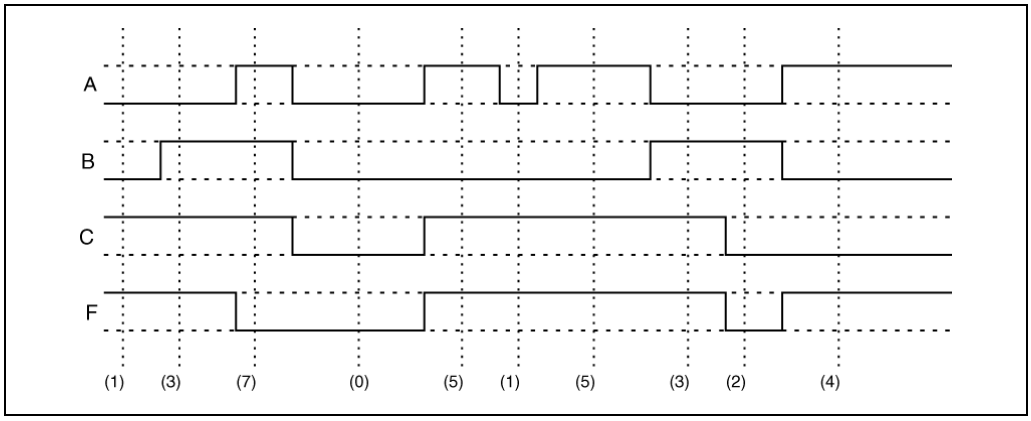
[13] And of course, you should always state any assumptions you make in any problem. You should always make sure that you and the reader of your solution stay in the same mindset.

index	A	B	C	F
(0)	0	0	0	0
(1)	0	0	1	1
(2)	0	1	0	0
(3)	0	1	1	1
(4)	1	0	0	1
(5)	1	0	1	1
(6)	1	1	0	0
(7)	1	1	1	0

Step 3) Use the state of the inputs signals to generate numeric indexes on the original timing diagram. Note that timing diagram includes vertical dotted lines for every notable span of time on the timing diagram. Stated in another way, after every input signal transition, the overall state of the inputs changed and we noted that on the timing diagram using the vertical dotted lines.

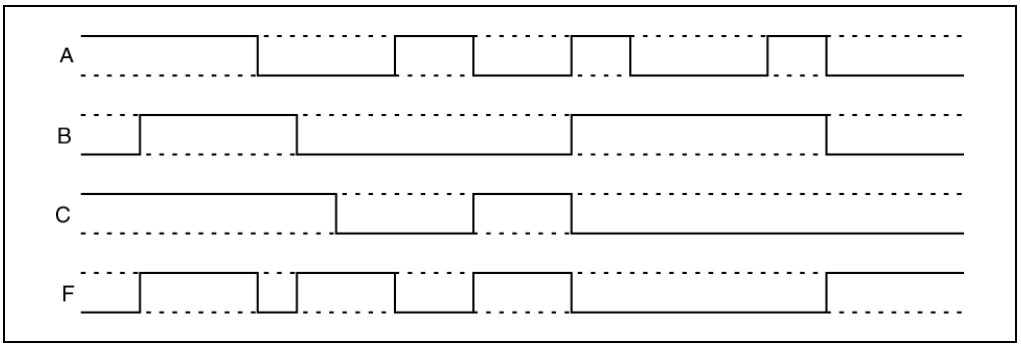


Step 4) Use the numbers you entered on the timing diagram to index into the truth table you generated for this problem. The outputs associated with each row of the truth table are graphically entered into the timing diagram with a 1's and 0's representing the high and low portions of the signal, respectively. The following timing diagram represents the final solution for this problem.



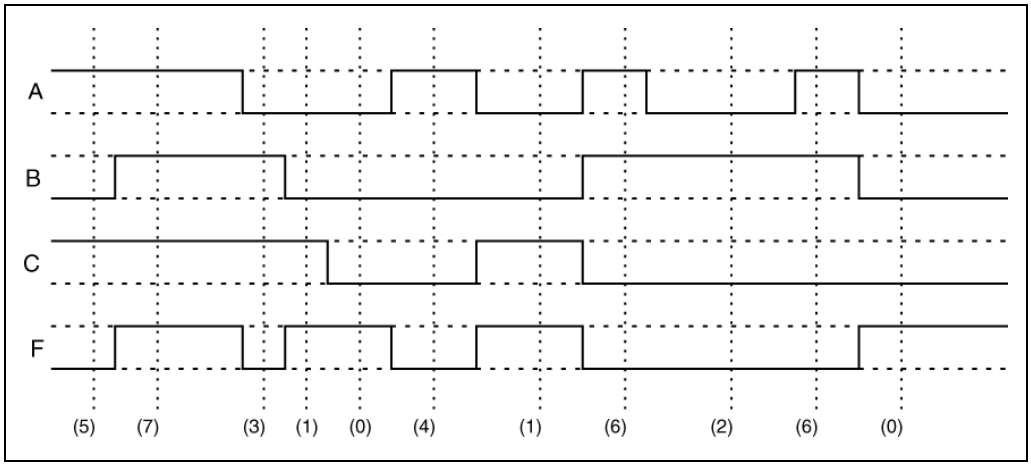
Example 7-2

If possible, use the timing diagram listed below to generate a Boolean equation that that describes the function modeled by the timing diagram. For this problem, consider A, B, and C to be inputs; F is an output.



Solution: Once again, there are many ways to do this problem. For this problem, the timing diagram seemingly models a circuit with three inputs and one output. The first issue we need to deal with is whether this timing diagram sufficiently describes a function. For the timing diagram to describe a function, two things need to happen. First, the timing diagram must represent all possible combinations of the three inputs. Second, for each of those individual combinations, the output must be consistent throughout the timing diagram in order for the timing diagram to model a function in the true mathematical sense of the word. Let's take a look.

Step 1) Find and mark all the input combinations represented in the given timing diagram.



Step 2) Because both of the conditions listed in the previous step exist, the given timing diagram does indeed represent a function. From this point, we can transfer the information from the timing diagram to a truth table. Once again, a "high" signal in the timing diagrams translates to a '1' in the resulting truth table. The diagram below shows the result of this step.

index	A	B	C	F
(0)	0	0	0	1
(1)	0	0	1	1
(2)	0	1	0	0
(3)	0	1	1	0
(4)	1	0	0	0
(5)	1	0	1	0
(6)	1	1	0	0
(7)	1	1	1	1

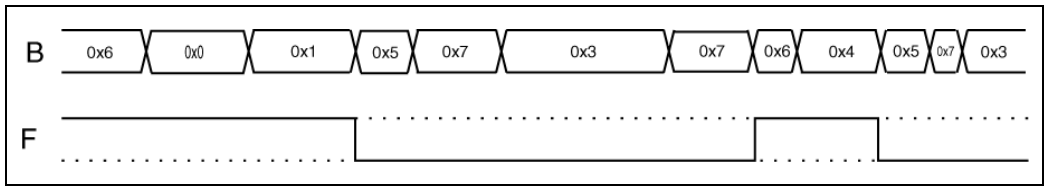
Step 3) From the previous truth table, we can generate the following Boolean equations. The equation below shows this is result in all its glory. We appear to be done with this problem.

$$F = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + ABC$$

Post Problem Commentary: This problem could be categorized as an "analysis" problem, or maybe even better as a "timing analysis" problem. Note that we "analyzed" the original timing diagram in order to arrive at our solution. In addition, if we were totally into the pointless digital self-flagellation thing, we could also draw the circuit associated with the final equation we generated. Let's definitely skip that.

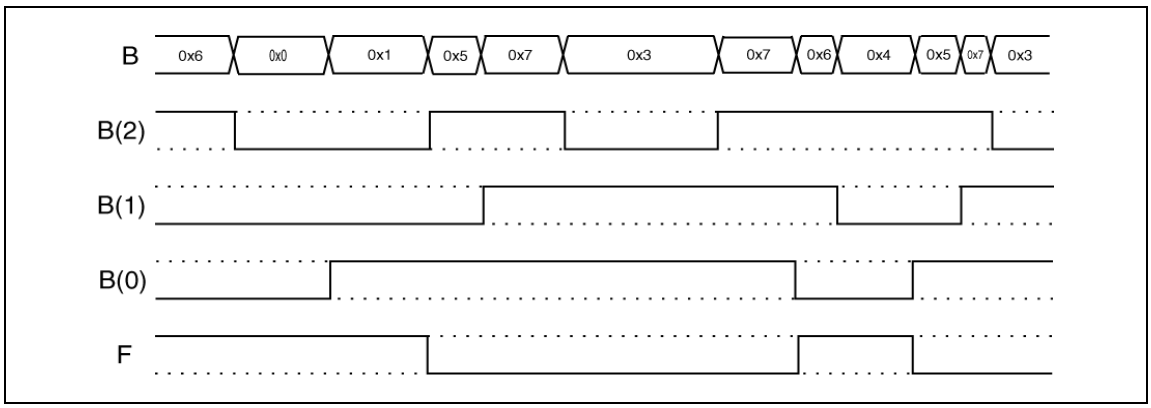
Example 7-3

Using the following timing diagram, expand the listed bundle into individual signal. For this problem, assume that signal labeled "**B**" represented a bundle with three individual signals. Use parenthetical indexing for the signal members of "**B**".



Solution: For this problem, we need to expand the bundle notation and list the individual signals of the bundle in the timing diagram. We will use parenthetical notation as specified by the problem which

dictates that **B(2)** is the MSB of the signal "**B**" and **B(0)** is the LSB of "**B"**. The diagram below lists the final solution. Are you ready for the final solution[14]?



[14] It's a reference to an Elvis Costello song; no need to panic.

Chapter Summary

- **Timing Diagrams:** One common and useful approach to modeling digital circuits is with a timing diagram. Timing diagrams show the state of signals over a given span of time. Timing diagrams explicitly show the functional relationship of digital circuits in that for every unique set of inputs, there is only one unique set of outputs. Timing diagrams use a signal's value (most often either '1' or '0') as the independent variable (the vertical axis) and time as the dependent variable (the horizontal axis). Complete timing diagrams can be used to completely specify a digital circuit's correct operation.

- **Timing Diagrams for Design:** Timing diagrams are often used to define problems. For example, you may see problems stated such as "design a circuit that has an input/output relationship modeled by the following timing diagram. In this way, the timing diagram is part of the circuit specification.

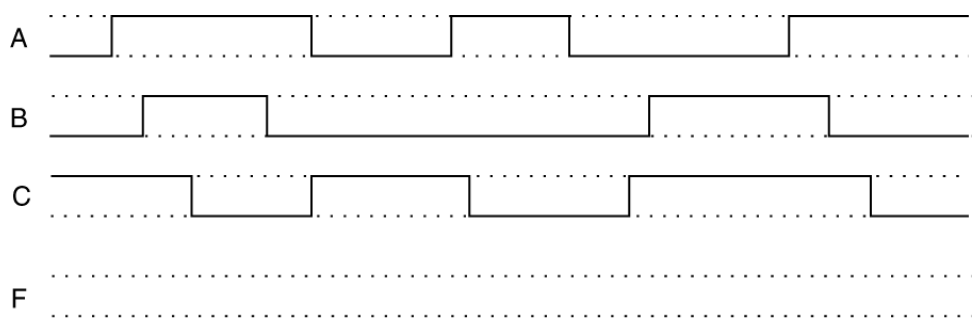
- **Timing Diagrams in Analysis:** Timing diagrams are often used for analysis. There are two aspects to timing diagrams used in analysis. First, the timing diagram may be the output of a "digital circuit simulator". In this way, you're testing the expected output of a circuit that you have not necessarily implemented. Secondly, many test devices typically output timing diagrams. The Logic Analyzer is a standard test device that essentially generates timing diagrams which results from testing an actual implemented circuit. Either way, the thing you're tying to figure out is whether your circuit will do (simulation) or actually does (implementation) the right thing.

- **Bundle Notation:** This notation consists of associating single signals with a common purpose into one signal that has multiple sub-signals. Digital design commonly uses this notation designs in order to simply the design and/or analysis process. Bundle notation is seen often in both schematics and timing diagrams. Bundle notation in schematics uses slash notation (a forward slash with a number indicating the number of signals in the bundle) while bundle notation in timing diagrams uses double bars with some type of indication of the value of the included signals.

Chapter Exercises

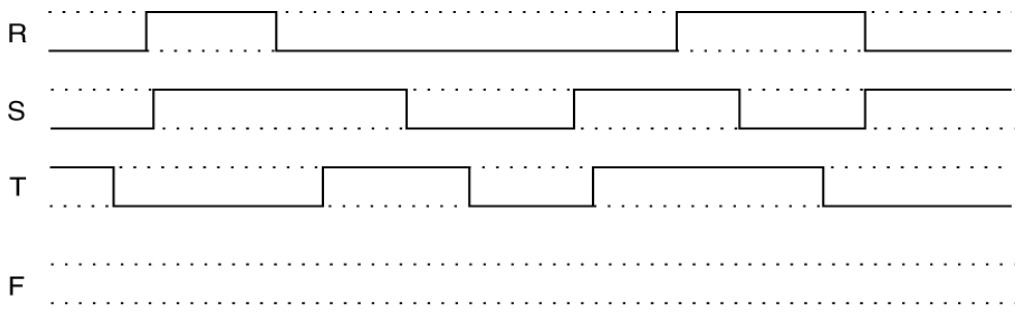
1) Using the following Boolean equation to complete the accompanying timing diagram.

$$F = BC + A\overline{B}C + \overline{A}B\overline{C}$$

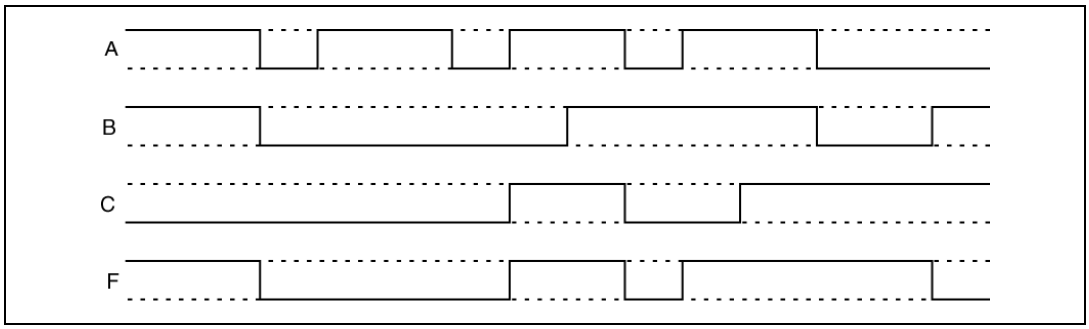


2) Using the following Boolean equation to complete the accompanying timing diagram.

$$F = \overline{R}ST + R\overline{S}\overline{T} + RS + R\overline{T}$$

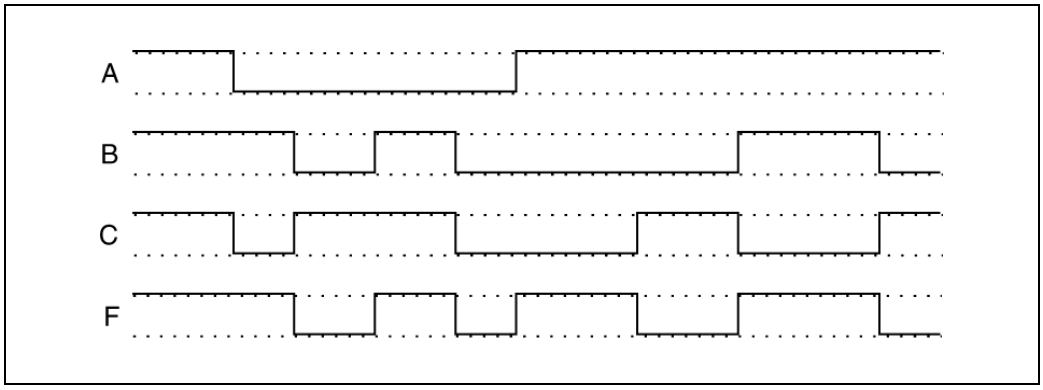


3) Does the timing diagram listed below completely define a function? Why or why not? If it does, write both SOP and POS equations that describes the function and provide a circuit diagram in both SOP and POS form that could be used to implement the circuit.



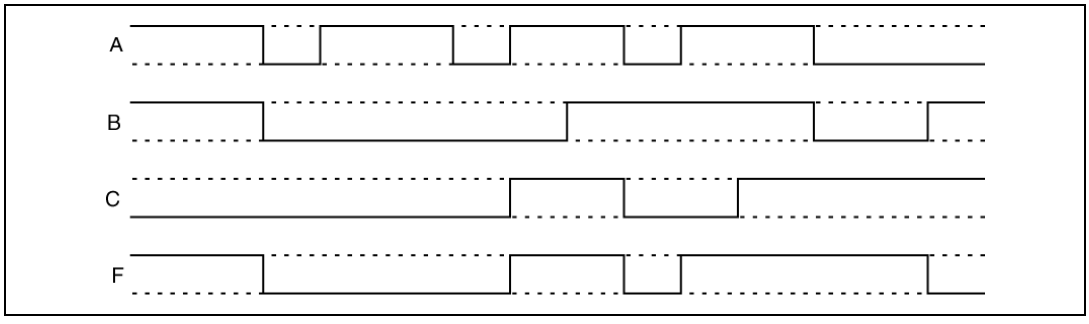
4) The following timing diagram may completely model a function.

- If the timing diagram defines a function, draw a circuit diagram for the function in reduced form.
- If the timing diagram does not define a function, explicitly describe why it does not.



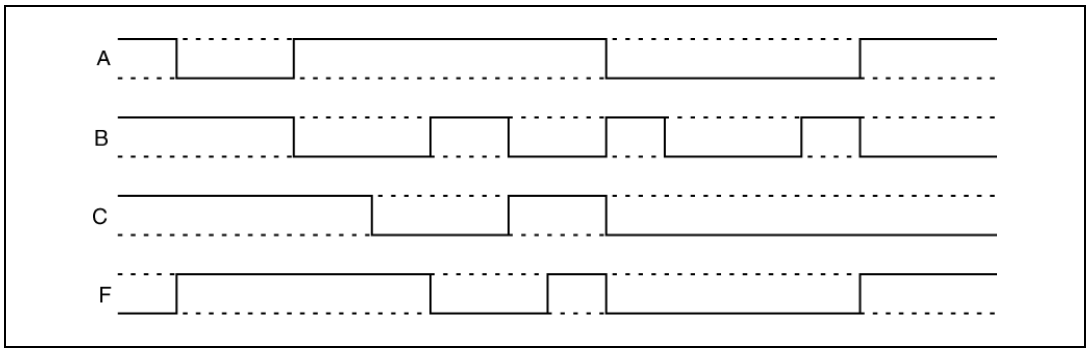
5) Consider the previous problem… can you safely state which of the inputs variables is the MSB or LSB? Be sure to provide a complete explanation.

6) Does the timing diagram listed below completely define a function? Why or why not? If it does, write both SOP and POS equations that describes the function and provide a circuit diagram in both SOP and POS form that could be used to implement the circuit.



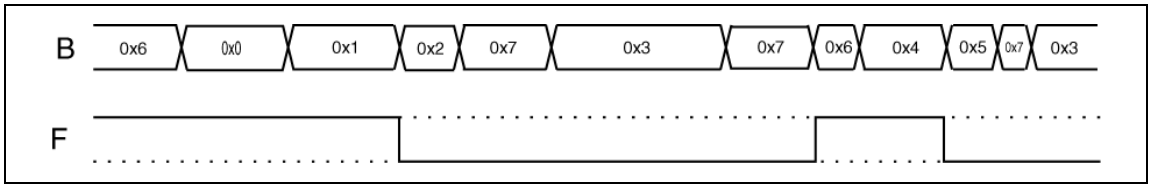
7) Consider the previous problem… how does the ordering of the labels of A, B, and C change the outcome of the problem? Be sure to provide a complete explanation.

8) Does the timing diagram listed below completely define a function? Why or why not? If it does, write both SOP and POS equations that describes the function and provide a circuit diagram in both SOP and POS form that could be used to implement the circuit.

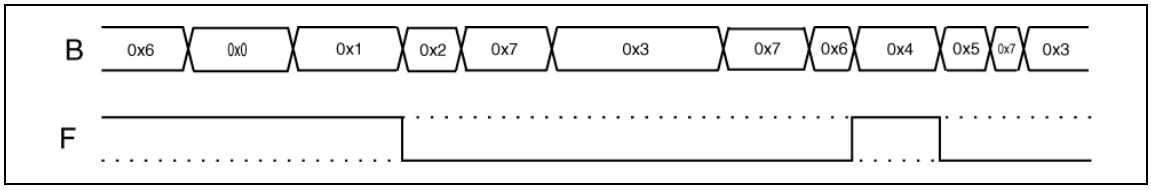


9) In your own words, under what conditions could the timing diagram of the previous problem ever be used in a real circuit setting?

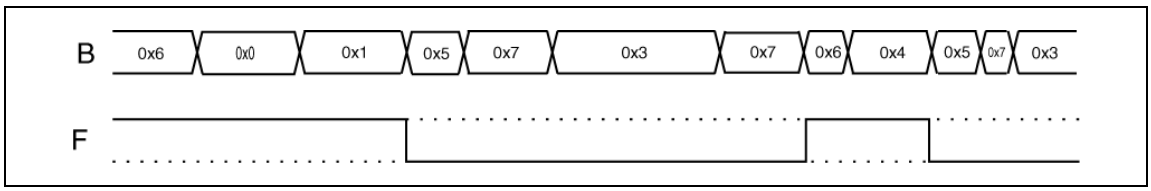
10) If the following timing diagram completely specifies a function, write a Boolean expression for that function.



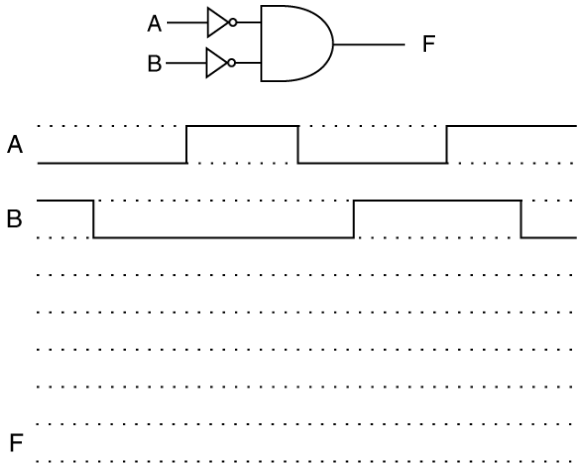
11) If the following timing diagram completely specifies a function, write a Boolean expression for that function.



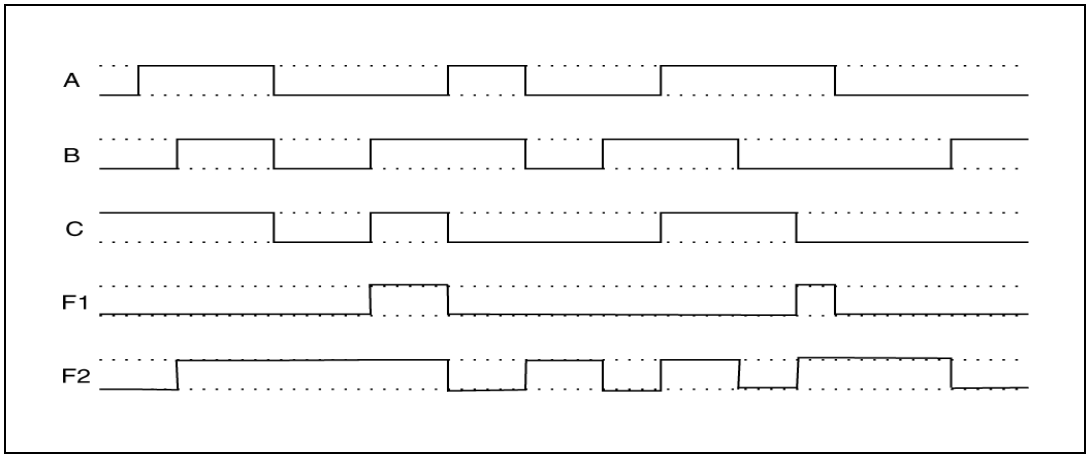
12) Does the following signal completely specify a Boolean function? Briefly explain why or why not.



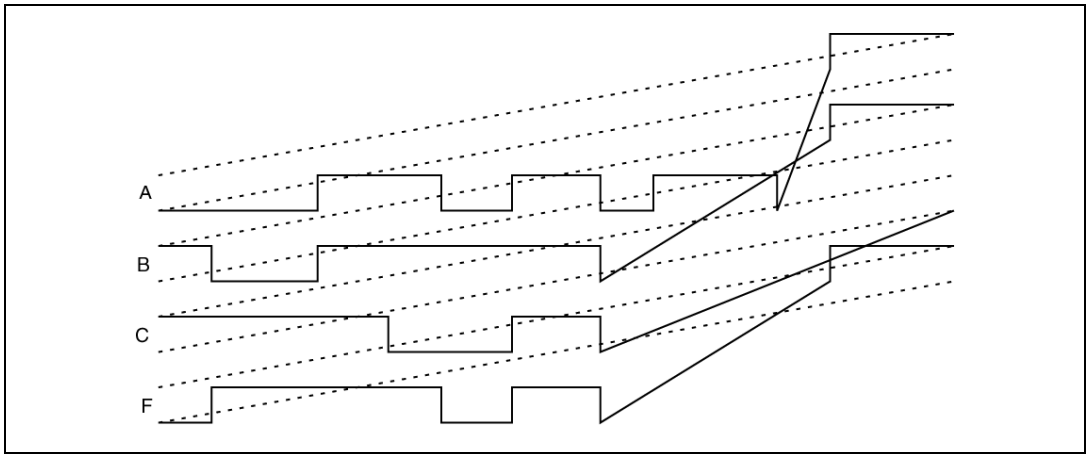
13) Complete the following timing diagram for the F output based on the given circuit.



14) For this problem, consider the input variables to be A, B, and C and the outputs to be F1 and F2. The timing diagram below completely described functions F1 and F2. Write a Boolean expressions that describe F1 and F2

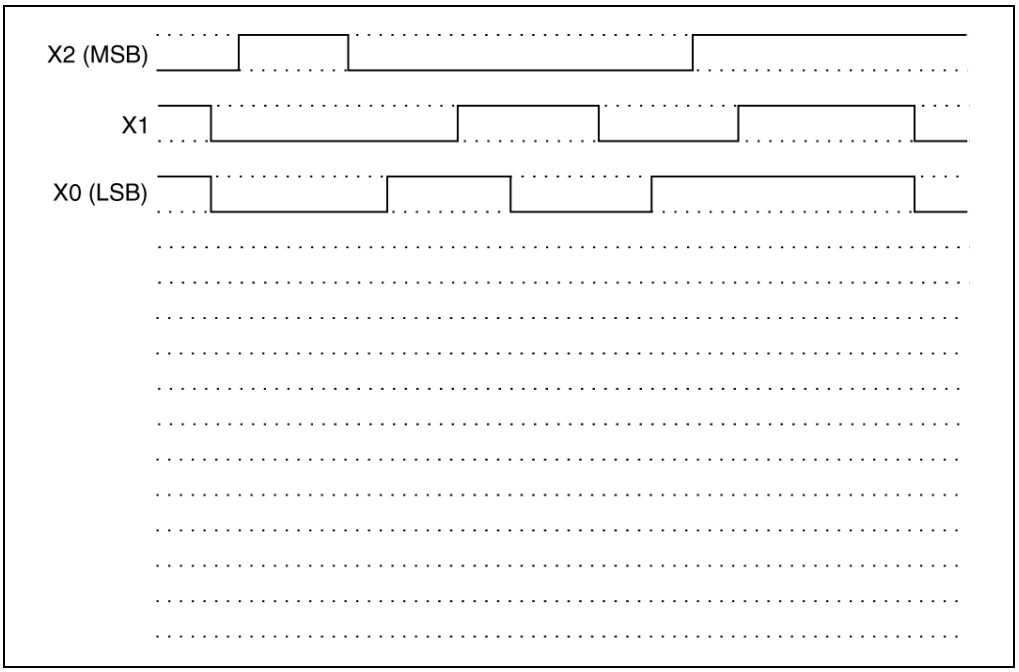


15) For those aspiring digital designers on drugs, state whether the timing diagram listed below completely defines a function? Why or why not? Does anyone really freaking care?

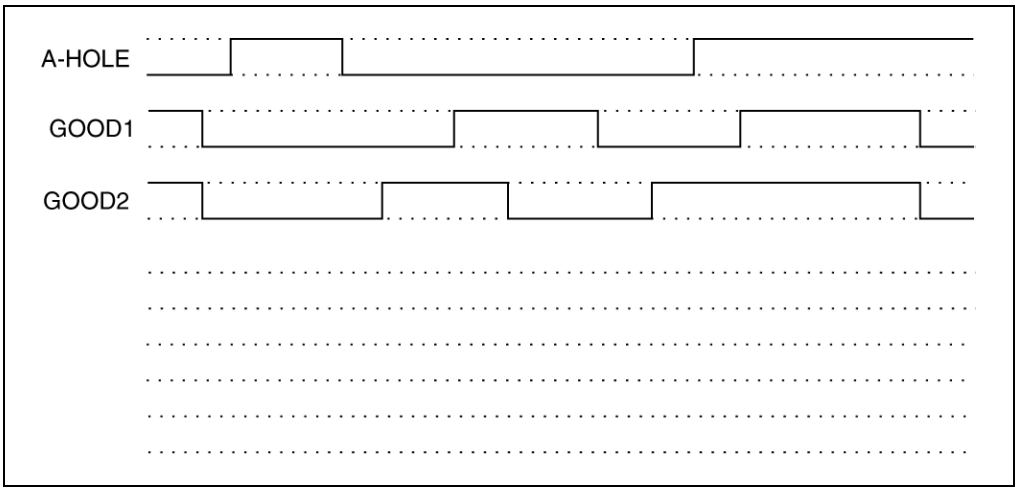


Design Problems

1) Design a circuit whose output represents a square of the input. For this problem, describe your design using SOP or POS equations. Also, waste yet even more time by completing the timing diagram listed below.



2) Design a digital circuit that will be used by the head of a typical committee in academia. The input labeled "A-HOLE" is the head of the committee; the other two committee members are labeled "GOOD1" and "GOOD2". Being a typical head of a committee, the chairman of the committee has commissioned you to build this circuit in order to better serve himself. The committee has a set of switches that are used for a "secret" vote. Your mission is to modify the circuit inputs such that there is always a majority in any way the head of the committee votes. Provide a truth table and equations for your circuit; also, complete the following timing diagram in order to prove that you may know what you're doing.


A-HOLE
GOOD1
GOOD2

8 Chapter Eight

(Bryan Mealy 2012 ©)

8.1 Introduction

This chapter continues up the digital design learning curve by introducing four new logic gates. Though you've been using AND and OR gates (and inverters) to implement your designs, it turns out that these types of gates are used the least out there in digital land (not including the inverters). After you've pounded through this chapter, you'll know all of the basic gates used in digital logic. At that point, we will have gained much more flexibility in our digital designs, which is good. Then soon after that, we do just about all we can never to use simple gates in our designs ever again[86]. Kind of strange.

Main Chapter Topics

- **STANDARD LOGIC GATES:** This chapter introduces four new gates: the exclusive OR (XOR) and exclusive NOR (XNOR) gates, and the NAND and NOR gates.

Why This Chapter is Important

This chapter is important because it describes three of the more common logic gates used in digital design.

8.2 More Standard Logic Gates

Although AND and OR gates implement the most basic digital logic functions, AND and OR gates are not the most widely used gates in digital design-land. In reality, AND and OR gates have some limitations that are not shared in the two most common digital logic gate-types: NAND and NOR gates. In addition to these gates, digital design uses the XOR and XNOR gates rather extensively.

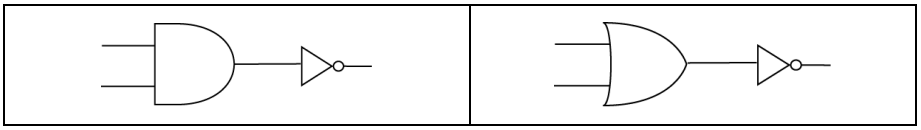
8.2.1 NAND Gates and NOR Gates

The NAND gate and NOR gate are formed by complementing the output of AND and OR gates, respectively[87]. The names NAND and NOR are a shortened version of NOT-AND (for NAND) and NOT-OR (for NOR). Figure 8.1 shows that the NAND and NOR gates can be modeled by adding an inverter on the output of the AND and OR gates. Inverting the AND and OR gate outputs can be

[86] Though this sounds dramatic, the real reason is based on the propaganda introduced in earlier chapters. Recall that in digital design, we always want to be designing at the highest level possible. While sometime you really need to use gates in your designs, they are relatively low-level devices and we generally try not to use them if possible.

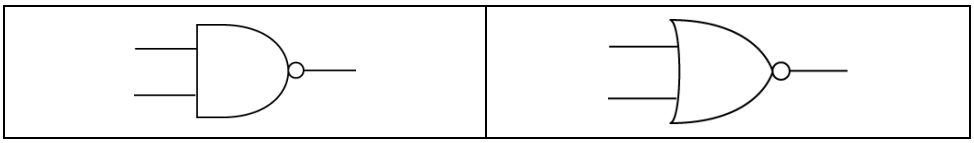
[87] Warning: this is only sort of true. You can think of these NAND/NOR gates with inverters on the outputs but there is a better way to model them. Don't worry about the better way for now. Pretend I didn't mention it.

considered as creating a new functional relationship and is thus rewarded by a unique gates symbols for these two new functions. Figure 8.2 shows the two new gate symbols for the NAND and NOR gates. Most appropriately, Figure 8.3 shows the truth tables associated with the NAND and NOR functions. Note that in Figure 8.3, the truth tables show that the outputs of the NAND and NOR gates are in fact complimented versions of AND and OR gates, respectively. Wow! Too much excitement one paragraph.



(a) (b)

Figure 8.1: Functional equivalent models for the NAND (a) and NOR (b) logic gates.



(a) (b)

Figure 8.2: The NAND (a) and NOR (b) logic gates.

A	B	$F = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

(a)

A	B	$F = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

(b)

Figure 8.3: Truth tables for the NAND (a) and NOR (b) logic functions.

There are several reasons why digital design uses NAND and NOR gates more often than AND and OR gates. One thing to keep in mind is that all logic gates are implemented by placing transistors into a circuit such that they create the desired logic functions. From a standpoint of the underlying transistor implementation, there is no amazing advantage to using a AND gate instead of a NAND gate.

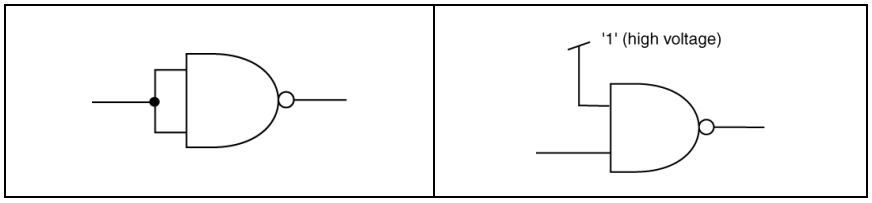
One of the advantages that NAND and NOR gates do have over AND and OR gates is that they are considered to be *functionally complete*. This means that a NAND gate (or a series of NAND gates) can implement any Boolean function[88]. In other words, a single NAND gate can be used to generate an AND function, an OR function, or a complement function (INVERTER). While some of the details of this statement are beyond what you need to know at this point, you can see from the truth table for the NAND gate in Figure 8.3(a) that there are two possible ways to create an inverter from a NAND gate.

[88] The same is true of a NOR gate; the details are not provided here.

Using a NAND gate to generate an AND function is obtained by adding an inverter to the output of the NAND gates. Using a NAND gate to generate an OR function is a mixed logic topic and is covered in a later chapter.

How does one make an inverter out of a NAND gate? Rows 1 and 4 of the NAND gate's truth table (shown in Figure 8.3(a)) indicate that if the two inputs to the NAND gate are equivalent, the output is an inversion of the input. Actual hardware can implement this characteristic by connecting the same signal to both inputs[89] of the NAND gate; in Figure 8.4(a) shows this result.

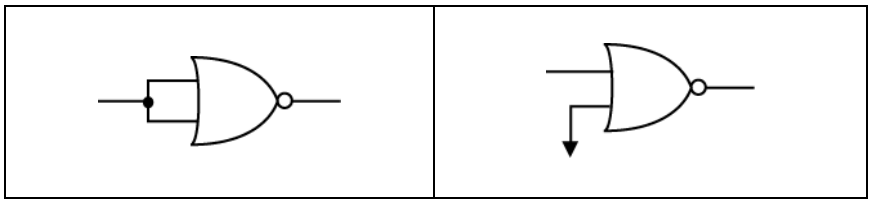
There's actually another way to turn a NAND gate into an inverter. Notice that Rows 3 and 4 of the NAND gate's truth table indicate that if one of the inputs to the NAND gate is fixed to a logic '1', the output of the NAND gate exhibits an inversion function based on the other input. This is implemented in hardware by connecting one of the NAND gate inputs to the high voltage in the circuit; Figure 8.4(b) shows a schematic of this relationship.



(a) (b)

Figure 8.4: Making an inverter from a NAND gate.

And as you can probably guess from knowing that NOR gates are functionally complete, there are a few ways to force a NOR gates to have an inversion function. Figure 8.1 shows the two approaches to making an NOR gate into an inverter. We state these without proof[90]; there are some chapter problems that may make you break a sweat with these concepts. Woohoo!



(a) (b)

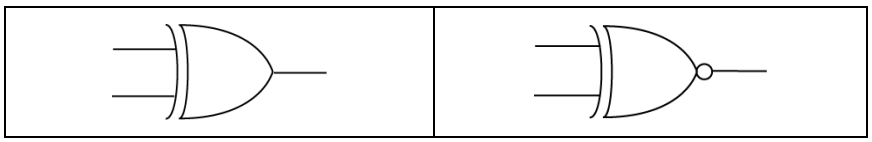
Figure 8.5: Making an inverter from a NOR gate.

8.2.2 XOR and XNOR Gates

The final type of logic gates that we'll introduce in this chapter are the *exclusive OR* gate (or the XOR gate) and the *exclusive NOR gate* (or *XNOR* gate). Figure 8.6 shows the schematic symbol for these two gates. Note the similarity between these gates and the OR and NOR gate symbols. Moreover,

[89] Or all of the inputs if there gates has more than two inputs.
[90] As is most stuff in this text…

sometimes the simple OR gate is referred to as an *inclusive OR* gate as opposed to the exclusive OR gate we're dealing with now.



(a) (b)

Figure 8.6: The exclusive OR (XOR) and exclusive NOR (XNOR) gates.

Figure 8.7 shows the truth tables that provide the official definitions for the XOR and XNOR functions. Note that the XOR and XNOR functions are complements of each other as is true with the OR and NOR gates. Figure 8.8 shows the official Boolean equations describing the XOR and XNOR functions. These equations differ slightly from the definitions of the previous gates we've dealt with. Whereas the AND/NAND and OR/NOR gates are defined by the basic axioms of Boolean algebra, the XOR/XNOR gates are not. The leads to the fact that the XOR and XNOR are often described in terms of the equations listed in Figure 8.8. Note that in these equations the XOR function has its own special operator symbol: the *circled cross*. There is also a special operator for XNOR gates which is not shown[91] in Figure 8.8(b): the *circled dot*.

The equations in Figure 8.8 are both massively important and useful; you'll use these equations often in digital design. You may want to stare at them for a while; I know I sure do[92]. One final thing to note here is that the XNOR gate is often referred to as an equivalence gate because the gate output is a logical '1' when the two gate inputs are equivalent. You can see this relationship from the XNOR definition of Figure 8.7(b).

A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

(a)

A	B	$F = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

(b)

Figure 8.7: Truth tables for the exclusive OR (XOR) and exclusive NOR (XNOR) functions.

(a) $F = A \oplus B = \overline{A}B + A\overline{B}$

(b) $F = \overline{A \oplus B} = \overline{A}\,\overline{B} + AB$

Figure 8.8: The official equations describing the XOR and XNOR functions.

[91] The equation editor I used when writing this does not contain the required symbol and I'm too lazy to seek a work-around.
[92] Not really.

In addition, the last thing to note about XOR and XNOR gates… They have one huge significant difference from other logic gates. While AND, OR, NAND, and NOR gates can have two or more inputs, XOR and XNOR gates can only have two inputs by definition of the gates. In the most general definitions of AND-type and OR-type gates, there is wiggle room to include multiple inputs. This is not true with XOR and XNOR gates. In reality, XOR-type gates are somewhat similar to inverters in that inverters have a fixed number of inputs.

8.3 Digital Design Gate Abstractions (whatever that means)

In the previous section, you saw that NAND and NOR gates can be configured as inverters. This fact is relatively useful and it comes up quite often in digital design-land. There is a bit more to the story so we'll be filling in the details in this section and hint at where these things can be used in digital designs. Gates are useful items in that they form the basis of digital design, but they also have other special functionality. You know the logic behind these gates, but applying basic intuitiveness to these gates allows them to be used in other ways, namely as a type of switch. Once you have an intuitive feel for the complete functionality of basic logic gates, you can use them in many digital designs in clever manner, so plan on placing these items in your digital bag of tricks.

Basic gates have three relatively useful functions beyond modeling their use as logic elements. These three functions include gates as inverters, gates as switches, and gates as buffers. The following verbage more fully describes these functions while Figure 8.9 provides some of the visual details. For each of these gates, we'll only consider the case of 2-input gates. Also worthy of note, for simplicity, we've omitted all mention of the XNOR gates as they are basically a special case of the XOR gate.

The key to making gates do these seemingly new and wonderful things is the notion of connecting an input (or inputs) to the power (logic '1') or ground (logic '0') of a circuit. Often times ground is referred to as "GND" and is shown with a down-pointed arrow in a circuit diagram. The thing that makes the physical logic devices in your circuit actually work is their connection to power and ground. Connect an input to logic '1' is often referred to as "tying the input high" or "tied high" while connecting an input to logic '0' is often referred to as "tying the input low" or "tied low".

Gates as Inverters: If you wire a gate properly, they can act as inverters. When one input is connected to power or ground (logical '1' or '0', respectively), the gates act as inverters. Table 8.1 lists the connections required to create inverter functions from various gates.

Gate Type	Gate Connected as Inverter
NAND	connect one input to '1'
NOR	connect one input to '0'
XOR	connect one input to '1'

Table 8.1: Gate connection for inverter functionality.

Gates as Switches: In this context, the notion of a switch means something we can turn on and turn off. This is the notion of "gate killing" which is quite useful in many digital design applications. The notion here is that if one gate input is a special known value, the output is always known and does not change. In other words, one input has the ability to

essentially disable the gate and force the output to a certain value that does not change as long as the given input remains the same.

Gate Type	Gate Connected as Switch
AND & NAND	connect one input to '1'
OR & NOR	connect one input to '0'
XOR	connect one input to '1'

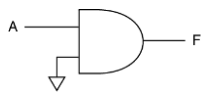
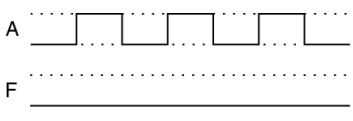
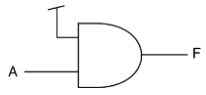
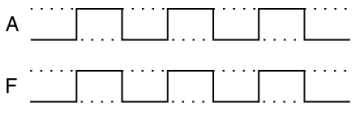
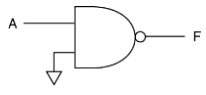
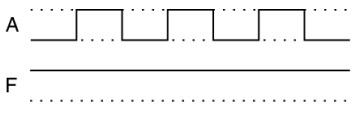
Table 8.2: Gate connection for switch functionality.

Gates as Buffers: The word buffer is a common term in all electronics including digital electronics. For digital electronics, a buffer function is essentially one that does not change the logic level of a given signal. This is generally useful because often times you want to pass a signal along in a circuit unchanged. This buffering action is often combined with either a switch or inverter functionality[93].

Gate Type	Gate Connected as Buffer
AND	connect one input to '1'
OR	connect one input to '0'
XOR	connect one input to '0'

Table 8.3: Gate connection for inverter functionality.

In addition, since it is purported that a picture is worth a gazillion words, Figure 8.9 graphically shows what we've been attempting to describe in the previous few paragraphs and tables. Good luck!

Gate Configuration	Timing Example	Comments
		When you ground an input to an AND gate, the output will always be zero, no matter how many inputs there are and no matter what the state of these inputs are. You've killed the gate.
		When you tie one input to an AND gate high, this input will essentially have no effect on the output of the AND gates. This creates a pass-though effect for the signal in the case where there is only one other input.
		This case is similar to the AND gate with one input tied low. The NAND gate is dead when one input is grounded. The output is thus stuck at '1' in this case.

[93] For example, for a given input, the value is either high or low and the resulting gate function is a buffer and an inverter, or a buffer and a switch (depending on which gates you're working with).

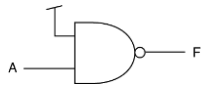
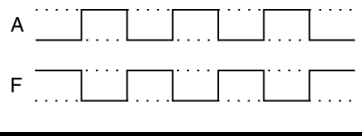
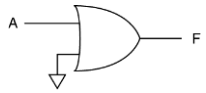
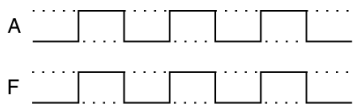
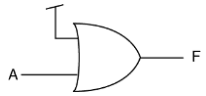
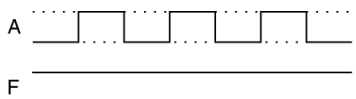
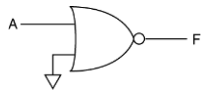
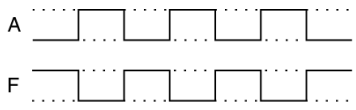
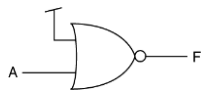
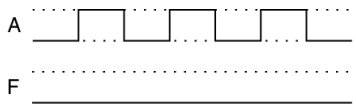
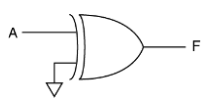
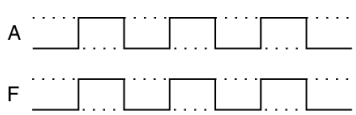
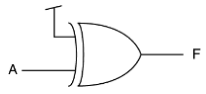
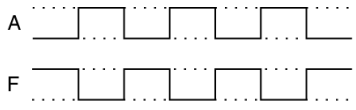
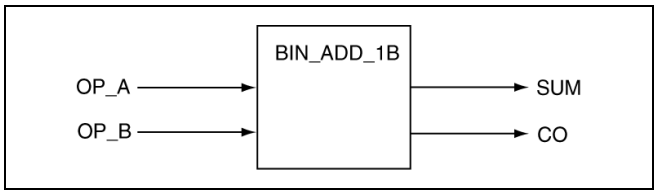
		When an input to a NAND gate is tied high, the other input becomes an inverter. This is the opposite of tying an AND gate high which gave a pass-through effect.
		Tying one input of an OR gate low prevents the input from having an effect on the output. This is similar to tying an input to an AND gate high.
		Tying an input to an OR gate '1' effectively kills the gate by forcing the output to always be '1'. None of the other inputs to the OR gate matter at this point.
		Tying one input of a NOR gate to '0' effective disables that input in that is can no longer effect the output.
		Tying a NOR gate to '1' effectively kills the gate in that the output is always low. This is similar to tying the OR gate high.
		EXOR gates by definition only have two inputs. When one input is tied to '0', the gate effectively becomes a pass-through for the other signal.
		EXOR gates by definition only have two inputs. When one input is tied to '1', the gate effectively acts as an inverter for the other signal.

Figure 8.9: Everything you didn't want to know about the secret lives of basic logic gates.

Example 8-1

Implement a half adder (HA) using a minimal amount of gates; use any type of gate you're familiar with in order to minimize the final gate count.

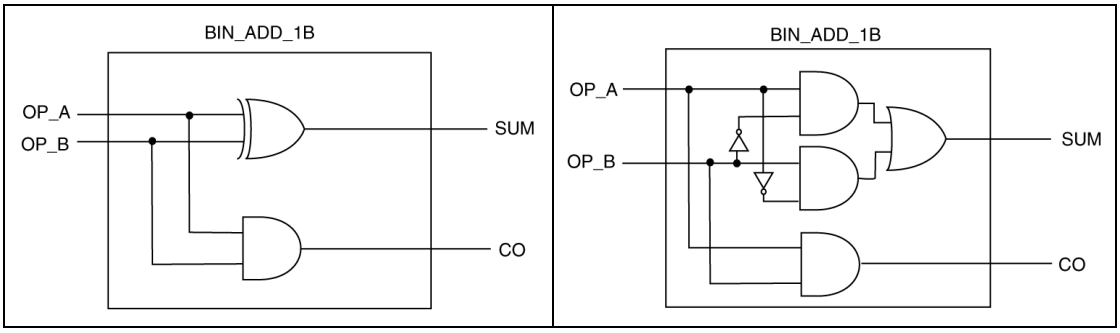
Solution: The first thing to note is that the HA was fairly simple; the second thing to note is that the problem drops a giant hint that you should use one of the new gates you're familiar with. Recall that the HA has two inputs and two outputs as shown below.



We all remember how a HA works but we provide the associated truth table once again without description.

OP_A	OP_B	SUM	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From inspection, you can see that the SUM output is an XOR function and the CO is an AND function. Seriously, you need to inspect things quite often in digital design as items such as XOR functions aren't delivered on flaming pies. The circuit diagram below on the left is the resulting circuit. The circuit below on the right is the final circuit from the first time we did this problem. The result is two devices for the XOR enabled counter compared to six devices for the original version. The world is saved.



Example 8-2: The Full Adder (FA)

Design a circuit that adds three bits: two bits are associated with a standard addition operation while the third bit is considered a carry-in bit. In other words, this circuit completes the following operation: $(a + b + c_i)$ where a and b are the standard additive operands and c_i represents the carry-in bit. The outputs of the circuit are identical to the half adder: SUM and Carry-out.

Solution: The first thing to notice about this design is its similarity to the half adder (HA). The difference between the HA and the full adder (FA) is that the FA contains an extra input, the carry-in bit. The bottom line is that while the HA was a two-bit adder, the FA is a three-bit adder; the outputs of the HA and FA are identical in that they both have a sum and a carry-out.

As always, let's start this design out with a black box diagram. Figure 8.10 shows the black box diagram for the full adder.

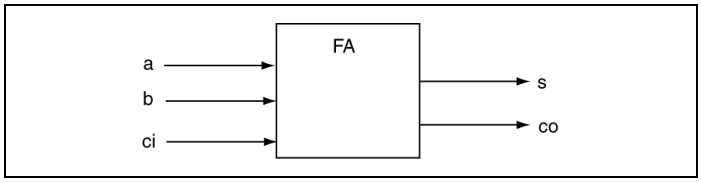


Figure 8.10: Black box diagram of the full adder.

The next step in the design is to specify the input/output relationship of the design. In other words, we must specify the outputs we want (based on the problem specification) for a given set of inputs. The truth table is one approach for defining this relationship since it lists every possible combination of the input variables (the independent variables). The outputs are then assigned based on the problem's original specification of adding the three input bits. Figure 8.11 shows the result of this step.

a	b	ci	s	co
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

(a)

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b)

Figure 8.11: The truth tables associated with the FA design specifications.

The next step is to translate the information in the two output columns of the truth table shown in Figure 8.11 into equation form. Equation 8-1 shows the final equations for the two output variables. From these output equations, you could easily draw the final circuit model. We leave drawing the final circuit model for the FA as an exercise for the reader (because I simply don't feel like doing it).

$$s = \bar{a} \cdot \bar{b} \cdot ci + \bar{a} \cdot b \cdot \overline{ci} + a \cdot \bar{b} \cdot \overline{ci} + a \cdot b \cdot ci \qquad co = \bar{a} \cdot b \cdot ci + a \cdot \bar{b} \cdot ci + \bar{a} \cdot \bar{b} \cdot ci + a \cdot b \cdot ci$$

Equation 8-1: Boolean equations describing sum and carry-out outputs of the FA.

Example 8-3:

Show that the following equations contain XOR functions.

$$F(A,B,C) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C}$$

Solution: This is one of those problems where there is actually an easier way to do it than the way it is done in this example (we'll be discussing those approaches in a later chapter). For now, the only way we know how to do this problem is to factor it. Factoring using Boolean algebra is simply something none of us wants to do, but sometime we must really do it. Here we go.

There is no better way to do this problem other than to stare at it and look for a starting point. The starting point we're looking for is a something that can be factored. This problem happens to be set up sort if nicely in that the natural ordering of the terms makes the problem easier. Without too much description, listed below is the final solution. Note that including the XOR function in the output significantly reduced the amount of logic in the final Boolean equation.

$$F(A,B,C) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C}$$
$$F(A,B,C) = \overline{A}(\overline{B}C + B\overline{C}) + A\overline{B}\overline{C} + A(\overline{B}C + B\overline{C})$$
$$F(A,B,C) = (\overline{B}C + B\overline{C})(\overline{A} + A) + A\overline{B}\overline{C}$$
$$F(A,B,C) = (\overline{B}C + B\overline{C}) + A\overline{B}\overline{C}$$
$$F(A,B,C) = (B \oplus C) + A\overline{B}\overline{C}$$

Chapter Summary

- NAND and NOR are formed from complimenting the outputs of the AND and OR gates, respectively. NAND and NOR gates are generally used more often that AND and OR gates in digital design.

- Exclusive OR (XOR) and exclusive NOR (XNOR) are two additional standard gates used in digital logic. These functions are somewhat useful for some basic digital circuits such as the Full Adder (FA).

- NAND and NOR gates are considered to be functionally complete which means that a NAND gate can be used to generate and AND function, and OR function, or an inversion function. AND and OR gates, however, are not functionally complete.

- Basic logic gates can be connected to work as inverters, switches, and buffers. These connections represent an extended functionality of basic gates and are quite useful in digital design.

- Important Standard Digital Modules presented in this chapter:
    - Full Adder (FA)

Chapter Exercises

1) Briefly describe why AND and OR gates are not considered functionally complete.

2) Briefly describe whether you feel XOR gates are considered functionally complete?

3) Explicitly describe how to make a NOR gate into an inverter.

4) Draw a diagram of a 4-input NAND gate that has been configured as an inverter. Don't combine inputs for this problem.

5) Draw a diagram of a 4-input NOR gate that has been configured as an inverter. Don't combine inputs for this problem.

6) What extended functionality can be obtained from a XNOR gate by connecting one input to either '1' or '0'? Briefly explain.

7) Write a reduced Boolean equation in SOP form for each of the following functions. Make sure you pull out the XOR functions where humanly possible.

$$F1(A,B,C) = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C + ABC$$

$$F2(A,B,C) = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}\overline{C} + ABC$$

Design Problems

1) Design a circuit that controls the locking mechanism of a room that contains three doors: door A, door B, and door C. Each door will allow only one person into the room when the controller you're designing unlocks the lock on that door. For this circuit, the door will remained locked under the following conditions:

a. When one person wants into each door
b. When no people want in any door
c. When one person wants in Door B but no one wants in any other door
d. When two people want in but no one wants in at Door B.

For this problem, provide an equation and a final circuit diagram for your solution. Be sure to extract any exclusive OR-type functions that may be present in your equations.

2) Design a circuit that controls the watering controller for you three precious plants. Assume each of your girls contain a sensor that indicates to the controller when each individual plant requires water. You've consulted the horticulturist and they told you that the water should only turn on when only one or only two plants require watering; the water should be off at all other time. For this problem, provide an equation and a final circuit diagram for your solution. Be sure to extract any exclusive OR-type functions that may be present in your equations.

9 Chapter Nine

(Bryan Mealy 2012 ©)

9.1 Introduction

Our approach to this point in digital logic design was to present digital logic basics and the basics of digital circuit modeling. Unfortunately, this introduction was somewhat on the quick side and omitted many of the finer points of both topics in order to allow you to actually design and model some digital logic circuits from start to finish. This chapter adds some more knowledge and techniques that help you design and/or represent digital circuits.

Main Chapter Topics

- **STANDARD FUNCTION REPRESENTATIONS:** This chapter introduces several standard methods used to represent functions and the terminology associated with these representations.
- **FUNCTION REDUCTION USING KARNAUGH MAPS:** This chapter introduces the concept of function reduction, and in particular, function reduction using Karnaugh maps.

Why This Chapter is Important

This chapter is important because it describes more methods of representing functions including reduced representations using Karnaugh maps.

9.2 Representing Functions

A significant part of designing digital circuits involves implementing functions. Once again, functions definitions describe an input/output relationship of a digital circuit. The design part of digital design has to do with deriving the function, but once it is derived (from whatever method you used to derive it), you need to somehow represent it[1]. There are many ways to represent functions in digital-land; this section presents a few of the more popular ways. We'll be picking up other ways as we progress through this text. Once again, keep in mind that you should be able to translate any one model (or representation) to any other model out there in the world of digital design.

[1] This is statement is something a bean counter would say: it sounds good, but is misleading and generally not completely true. The fact is that equations are somewhat klunky and hard to work with. Equations are most often associated with beginning stages of digital design; but as you move on in digital-land, you quickly move to other design approaches. Later chapters discuss these approaches; we need to deal with just a few more basics for now.

9.2.1 Minterm & Maxterm Representations

Without you knowing it, you've already been exposed to *minterm representations* and *maxterm representations* of functions. For this section, let's return to the design overview example used in the previous chapters. For your convenience, Figure 9.1 shows the equation for the function we were previously working with. From the truth table of Figure 9.1, you generated the Boolean function shown in Equation 9-1 to describe the information contained in the truth table. We eventually went on to describe Equation 9-1 as sum-of-products form (SOP) but that is not the whole story. As it turns out, this equation is actually listed in what is known as "standard SOP form". You know that the equation is in SOP form because you can see the product terms are being summed together. So what makes it a standard SOP form?

B2	B1	B0	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 9.1: The generic function used in a previous chapter.

$$F = B2 \cdot \overline{B1} \cdot B0 + B2 \cdot B1 \cdot \overline{B0} + B2 \cdot B1 \cdot B0$$

Equation 9-1

Equation 9-1 is a standard SOP form because each of the product terms contains only one instance of each of the function's independent variables. Note that the independent variables in the product terms can appear in either complimented or uncomplimented form. The product terms in Equation 9-1 are considered something special in that they are *standard product terms*[2]. When we're describing a function using product terms, we simply list the product term associated with the row in the truth table that contains an output of '1'. Each row in the truth table has a unique product term associated with it; Table 9.1 shows the product terms for three-variable (A, B, C) function.

As you see from Table 9.1, the product terms are also labeled as "minterms"s which is simply another name for a standard product term. Another term that is used sometimes is that notion that an equation listed in standard SOP form is often referred to as a *minterm expansion* of the function. Equation 9-2 shows the standard SOP form of the function from the previous example (note that we've switched from B2, B1, and B0 to A, B, and C to make things easier for the lazy author).

[2] Later in this set of notes you'll see that listing all the terms as standard product terms not generally done.

$$F = A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$$

Equation 9-2

As you have probably guessed, there is also going to be a standard product of sums (POS) form which contains standard sum terms that are logically multiplied together. In this case, a standard sum term is referred to as a *maxterm*. The main difference between minterms and maxterms is that maxterms describe the locations of the 0's in the function's output[3]. Or equivalently, maxterms describe the 1's in the output of the complemented function. Equation 9-3 shows the standard POS form of the function; this form is sometimes referred to as a *maxterm expansion*.

$$F = (A + B + C) \cdot (A + B + \overline{C}) \cdot (A + \overline{B} + C) \cdot (A + \overline{B} + \overline{C}) \cdot (\overline{A} + B + C)$$

Equation 9-3

A	B	C	minterm	maxterm	F	index
0	0	0	$\overline{A} \cdot \overline{B} \cdot \overline{C}$	$A + B + C$	0	0
0	0	1	$\overline{A} \cdot \overline{B} \cdot C$	$A + B + \overline{C}$	0	1
0	1	0	$\overline{A} \cdot B \cdot \overline{C}$	$A + \overline{B} + C$	0	2
0	1	1	$\overline{A} \cdot B \cdot C$	$A + \overline{B} + \overline{C}$	0	3
1	0	0	$A \cdot \overline{B} \cdot \overline{C}$	$\overline{A} + B + C$	0	4
1	0	1	$A \cdot \overline{B} \cdot C$	$\overline{A} + B + \overline{C}$	1	5
1	1	0	$A \cdot B \cdot \overline{C}$	$\overline{A} + \overline{B} + C$	1	6
1	1	1	$A \cdot B \cdot C$	$\overline{A} + \overline{B} + \overline{C}$	1	7

Table 9.1: A listing minterms and maxterms for the each combination of circuit inputs.

As a final note here, there is a special relationship between the minterms and maxterms. For a given row in the truth table, the minterms and maxterms are compliments of each other as is shown by the equations in Figure 9.2. To generate a minterm from a maxterm (or vice versa), you first complement it and then tweak it using DeMorgan's theorem (if you need to). Figure 9.3 shows an example of this relationship for the fourth row in Table 9.1. In Figure 9.3(a), the equation for the given minterm is complimented and then DeMorganized which generates the associated maxterm.

[3] More specifically, maxterms describe the location of the 0's in the rows containing 0's for the uncomplimented output.

$\text{Minterm} = \overline{\text{Maxterm}}$	$\text{Maxterm} = \overline{\text{Minterm}}$

Figure 9.2: The secret relationship between minterms and maxterms.

$F(A,B,C) = F(1,0,0) = A \cdot \overline{B} \cdot \overline{C}$	$F(A,B,C) = F(1,0,0) = \overline{A} + B + C$
$\overline{F(A,B,C)} = \overline{F(1,0,0)} = \overline{A \cdot \overline{B} \cdot \overline{C}}$	$\overline{F(A,B,C)} = \overline{F(1,0,0)} = \overline{\overline{A} + B + C}$
$\overline{F} = \overline{A} + \overline{\overline{B}} + \overline{\overline{C}}$	$\overline{F} = \overline{\overline{A}} \cdot \overline{B} \cdot \overline{C}$
$\overline{F} = \overline{A} + B + C$	$\overline{F} = A \cdot \overline{B} \cdot \overline{C}$
(a)	(b)

Figure 9.3: Examples showing the complimentary relationship between minterms and maxterms.

9.2.2 Compact Minterm & Maxterm Function Forms

As you can clearly see, representing functions in standard SOP or POS forms is klunky. To remedy this, we use the compact minterm form or the compact maxterm form instead. This is easy to do, and as the names implies, it's a lot less work than the standard forms. The compact minterm and maxterm forms are simply a case of listing the decimal index (shown in the right-most column of Table 9.1) associated with the rows where either the 1's or 0's of the circuit reside in a given truth table.

Compact forms traditionally use Greek symbols in their representations; the summation symbol used for listing minterms (since it is a "summing" of product terms) and the capital Pi symbol used for listing maxterms[4]. Figure 9.4 shows the compact minterm and compact maxterm forms for the example we're working with. One important thing to note here is that these compact forms always need listing as a function of the independent variables. If you did not include all of the independent variables, you would not be able to expand the list into standard sum or standard product terms.

$F(A,B,C) = \sum(5,6,7)$	$F(A,B,C) = \prod(0,1,2,3,4)$
(a)	(b)

Figure 9.4: Compact minterm and maxterm forms for the current example.

One final comment on these different function forms. You've now have learned the following ways to represent functions: truth tables, standard SOP, standard POS, compact minterm, compact maxterm, and circuit forms. The forms relate to each other in that they essentially provide multiple ways of representing the same thing. In other words, all of these different forms are functionally equivalent. This being the case, you should be able to change from any one of the forms to any other one of the

[4] If you consult the right source, you'll find that the Pi symbol is associated with multiplication. This text, however, is not the right source.

forms. We'll work more with these translations later. However, while switching from one form of a function to another is excruciatingly exciting, it certainly does not represent digital design, as most digital design textbooks lead you to believe. Though it makes for easy-writing exam questions, it ain't digital design.

9.2.3 Reduced Form Representation: Karnaugh-Maps

There are three main recurring themes present in modern digital design: 1) make your circuit smaller, 2) make your circuit faster, and 3) make your circuit consume less power. As you'll see later in your engineering careers, you've never be able to achieve all of these things with the current digital technology[5].

Beginning digital design students are primarily concerned with the first issue: making circuits smaller. Generally speaking, if your circuit is smaller, your circuit operates faster and consumes less power; but it's actually more complexicated than that. Some of the function forms are easier to represent when the circuit is in a "reduced form" (no one wants to draw a large circuit on a piece of paper if there is a smaller equivalent circuit). There is one problem with the previous statement: we have not defined the word "smaller". As you may guess, the word "smaller" has many connotations, particularly in digital design-land. For the purpose of this chapter, we'll define one circuit to be smaller than another circuit if it uses less gates and/or if the gates in the circuit contain less inputs[6]. Implicit in this definition is that the two circuits in question are functionally equivalent, regardless of the number of gates or gate inputs.

You've already have been exposed to functionally equivalent circuit of different sizes with the example we've been using over the course of this and the previous chapters. Figure 9.5 shows the two main equation forms for the previous example. As you know, these are the standard SOP and standard POS forms; these two forms look significantly different but are actually functionally equivalent.

What should be obvious from looking at Figure 9.5 is that if you had to implement this circuit using discrete gates and wires and all that ugly stuff, you would choose the standard SOP form because there appears to be less work involved because there are a fewer number of gates than the POS form. The result is that if you can make the circuit "smaller", life is good. Making the circuit "smaller", or "reducing" it, is the main idea in this section. Reducing functions, or *function reduction* refers to the practice of making the function representation "smaller" without changing the input/output relationship of the function.

$$F(B2,B1,B0) = (B2 \cdot \overline{B1} \cdot B0) + (B2 \cdot B1 \cdot \overline{B0}) + (B2 \cdot B1 \cdot B0)$$

$$F(B2,B1,B0) = (B2+B1+B0) \cdot (B2+B1+\overline{B0}) \cdot (B2+\overline{B1}+B0) \cdot (B2+\overline{B1}+\overline{B0}) \cdot (\overline{B2}+B1+B0)$$

Figure 9.5: Two functionally equivalent forms of the example circuit.

Unfortunately, a significant component in most introductory digital design courses is to make students reduce functions using the list of theorems presented in a previous chapter. If this were the only way to reduce a function, everyone would need to learn to reduce functions with the various digital theorems.

[5] There is a famous quote out there in digital land: "smaller, faster, less power… choose any two". I'm not sure who said this.

[6] Or some combination of number of gates and number of inputs. Once again, this is not overly important as gate count is not a huge priority in the flavor of digital design we concentrate on in this text.

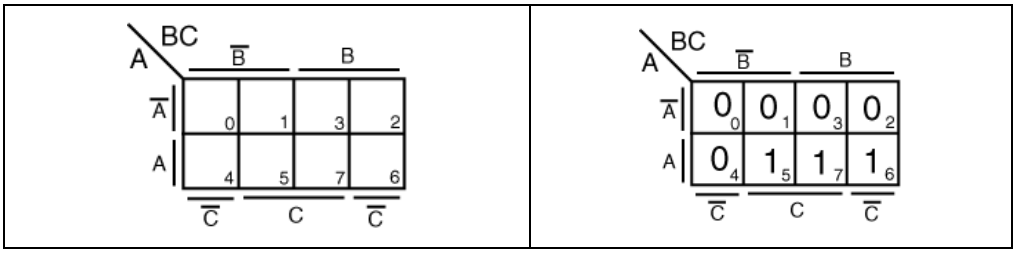
However, this is simply not the case for several reasons. First, there are better ways to reduce functions by hand (and we'll be learning those ways in this section). Secondly, if you really had to reduce a function and needed to know that you did the right thing, you would use a the appropriate software on a computer. The computer is limitless in the number of independent variables it can handle and does not make mistakes. Your time is valuable and you make mistakes. Moreover, out there in engineering land, no employer is stupid enough to pay you an engineering salary to reduce Boolean equations by hand[7]. Use a computer to reduce functions when you can; if you can't use a computer, use Karnaugh Maps. Many programs out there reduce functions. In addition, if you felt inspired to write your own function reduction program, there are many algorithms out there that you could code up.

This section describes using Karnaugh maps to reduce functions. Karnaugh maps are essentially a tool to visually apply the Combining theorem (which was listed in the previous chapter and once again here for you convenience). The Combining theorem is usually referred to as the Adjacency theorem which is a better definition for our purposes.

14a	$(x \cdot y) + (x \cdot \overline{y}) = x$	14b	$(x + y) \cdot (x + \overline{y}) = x$	**Combining**

Table 9.2: The Adjacency theorem in living color.

Jumping right into it… what we want to do is enter the information shown in Figure 9.1 into a special diagram referred to as a *Karnaugh map*. Figure 9.6 shows an empty three-variable K-map and a three-variable K-map including the output data from Figure 9.1.



(a) (b)

Figure 9.6: An empty Karnaugh map and a K-map that contains a functional relationship.

There are a few important things to note about the K-map of Figure 9.6:

- Each square, or cell, in the K-map contains a number, which is an index into the associated truth table of Table 9.1. In other words, the data associated with the output variable is placed into the corresponding cell in the K-map. There is necessarily one K-map cell for each row in the truth table.

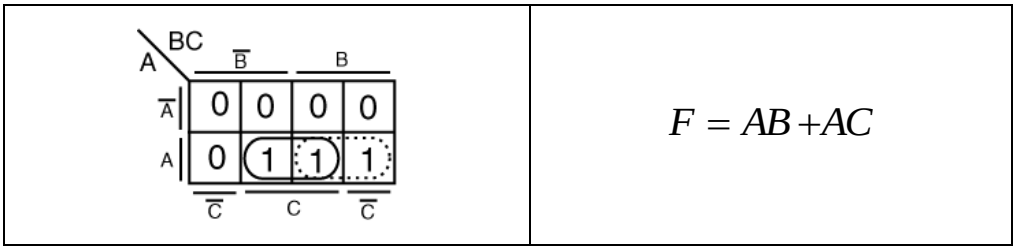
- The K-map includes a funny numbering system, which is referred to as a *unit-distance code* (discussed in a later chapter). What you need to know now is that the numbering system used in the K-map has a funny jog in it. This jog becomes more pronounced for the 4-variable K-map, which we'll deal with soon. Since a unit-distance code is used, each cell in the K-map is adjacent to cells in the horizontal and vertical directions. Of course, vertical does not mean

[7] Unless of course you work for a government contractor; in this case, you'll probably never have anything meaningful to do.

much in this example. What this does mean is that cell 4 is adjacent to cell 6 despite the fact that they don't appear to be connected.

- There are major portions of the K-map that are associated with a given state of each input variable. For example, the top row of the K-map is associated with the complement of input A while the bottom row of the K-map is associated with the uncomplimented A. Similarly, the right-most four cells of the K-map are associated with the uncomplimented B variable while the left-most four cells of the K-map are associated with the complimented B. A similar argument can be made for the C variable but we'll not bore you with the details. Note that the complimented C variable appears on left-most and right-most rows of the K-map. Figure 9.6 shows these boundary issues with the variables listed on the outside of the K-maps.

The approach from here is to start making groupings of 1's in the K-map in Figure 9.6(b). This is going to seem somewhat funny at first but… you'll quickly get the hang of it[8]. Figure 9.7(a) shows the groupings we'll make; we'll deal with why and how we made these grouping next. Figure 9.7(b) shows the final grouping associated the two groupings made in Figure 9.7(a). Note that the equation in Figure 9.7(b) is much "smaller" or more "reduced" than the equations shown in Figure 9.5. Once again, aside from the fact that these equations appear different, they are functionally equivalent.



(a) (b)

Figure 9.7: The K-map with groupings (a) and the reduced equation (b).

There are three distinct parts to performing K-map reduction: 1) entering the function data into the K-map, 2) making the groupings in the K-map, and 3) generating the equation to represent the groupings. We've covered the first topic for 3-variable K-map, and now we'll cover how to write equations for the groupings you made in step 2).

Writing down the equations from a K-map is a simple matter of listing (in product-term form) where the grouping resides in the K-map in terms of the independent variables. For the grouping made with the solid line in Figure 9.7(a), you should be able to see that it lives in the A row and the two C columns. Since this grouping resides in both the B and complemented B columns, the B variable is not included in the product term. The result is the product term "AC". A similar argument can be made for the grouping made with the dotted line: this grouping lives cleanly in the A row and the two B columns and thus generates a product term of "AB". Figure 9.7(b) shows the two product terms associated with the grouping of Figure 9.7(a).

The next step is learning the basic rules of making the groupings. Many digital design textbooks provide a set of rules that instruct you on how to make the groupings. These rules are generally hard to follow because they generally span about a full page of the text. These rules, however, are condensed into the four rules listed in Figure 9.8. It's sort of a drag to follow rules such as these but you'll quickly

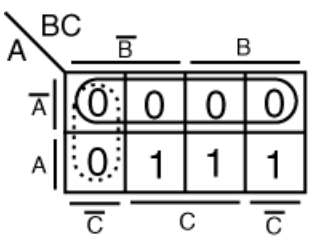
[8] Have no fear; no one has ever had long-term issues with K-maps.

get the hang of things. Notice in the groupings made in Figure 9.7(a) follow the rules provided in Figure 9.8.

1) Groupings must contain either 1, 2, 4, 8, or 16 cells
2) Groupings must have four corners (such as squares or rectangles)
3) Make the groupings as large as possible (cover as many cells as possible with one grouping while following the previous two guidelines)
4) Make as few groupings as possible to cover the "cells of interest"

Figure 9.8: The three rules of generating K-map groupings.

In case you didn't notice, Karnaugh maps inherently generate SOP forms. Despite this fact, K-maps can also generate reduced POS forms, but with several extract steps in-between. The grouping of terms to generate POS forms follows the same rules as the SOP forms but with one exception: for POS forms, you must start the process by grouping the 0's in the K-map[9]. Once you have grouped the 0's, you're able to generate a SOP equation for the complimented output function. From there, as with the truth table approach to generating a POS form from a SOP for of the complemented output, you must apply DeMorgan's theorem. Figure 9.9 shows an example of this process.



$$\overline{F} = \overline{A} + \overline{B}\overline{C}$$
$$\overline{\overline{F}} = \overline{\overline{A} + \overline{B}\overline{C}}$$
$$F = \overline{(\overline{A})} \cdot \overline{(\overline{B}\overline{C})}$$
$$F = A \cdot (\overline{\overline{B}} + \overline{\overline{C}})$$
$$F = A \cdot (B + C)$$

(a) (b)

Figure 9.9: The K-map with groupings (a) and the reduced equation (b).

Keep in mind that it is massively important to be able to generate either a reduced SOP or POS form using K-maps. The K-map techniques are similar: for the SOP form, you first need to group the 1's in the K-map and then write down the product terms associated with the groupings. For the POS form, you're grouping 0's which creates an SOP expression for the complemented function. From there, you need complement the generated equation and apply DeMorgan's theorem (maybe several times) to generate the final POS expression.

There are a few tricks[10] associated with using K-maps; once you see and use these tricks a few times, you'll be a K-map superstar. The following set of figures show all the interesting K-map tricks that you need to be aware of in order to use K-maps to reduce functions. The following figures list examples in conjunction with the compact minterm and maxterm forms. These are all the tricks I know for three and

[9] Recall that grouping the 0's for a function is equivalent to grouping the 1's for the complimented function.
[10] These are not really tricks; they're actually only groups that are not patently obvious from the stated K-map rules.

four variable K-maps. For those you on drugs, take a gander at what you need to do for K-maps of five and greater variables[11].

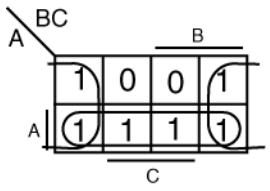
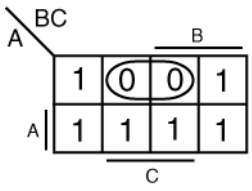
	
$F(A,B,C)=\sum(0,2,4,5,6,7)$	$F(A,B,C)=\prod(1,3)$
$F=A+\overline{C}$	$\overline{F}=\overline{A}\cdot C \Rightarrow F=A+\overline{C}$
(a)	(b)

Figure 9.10: K-map grouping for SOP (a) and POS (b) forms.

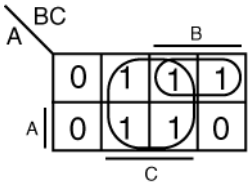
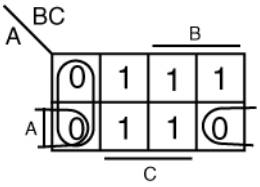
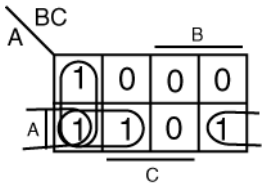
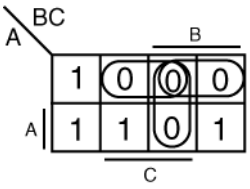
	
$F(A,B,C)=\sum(1,2,3,5,7)$	$F(A,B,C)=\prod(0,4,6)$
$F=\overline{A}B+C$	$F=A\overline{C}+\overline{B}\overline{C} \Rightarrow F=(\overline{A}+C)(B+C)$
(a)	(b)

Figure 9.11: K-map grouping for SOP (a) and POS (b) forms.

[11] This is a total waste of time. K-maps themselves are somewhat of a waste of time; it's only a matter of time before no one uses them anymore. I hope that time comes soon.



(a)



(b)

$$F(A,B,C) = \sum(0,4,5,6)$$

$$F = A\overline{B} + A\overline{C} + \overline{B}\overline{C}$$

$$F(A,B,C) = \prod(1,3)$$

$$\overline{F} = \overline{A}B + \overline{A}C + BC$$

$$F = (A + \overline{B})(A + \overline{C})(\overline{B} + \overline{C})$$

Figure 9.12: K-map grouping for SOP (a) and POS (b) forms.

The four variable K-maps are only slightly different from the 3-variable K-maps. In addition to the jog in the count that appears in the third and fourth columns in the 3-variable K-map, a similar jog appears between the third and fourth rows in the 4-variable K-maps. Figure 9.13 shows a blank 4-variable K-map with the funny count listed. The number appearing in the cells are indexes into a truth table that contains 2^4 or 16 entries [0,15]. The most common error in when using 4-variable K-maps is to forget to make the row jog when entering values into the K-map; please don't forget the jogs.

Listed in the figures following Figure 9.13 are all the known tricks for dealing with 4-variable K-maps. There are not that many and they are not that complicated as they do follow the four K-map grouping rules. These may seem strange at first, but after you do a few of these, they become second nature and there won't be anything you need to commit to memory (except for the row jog).

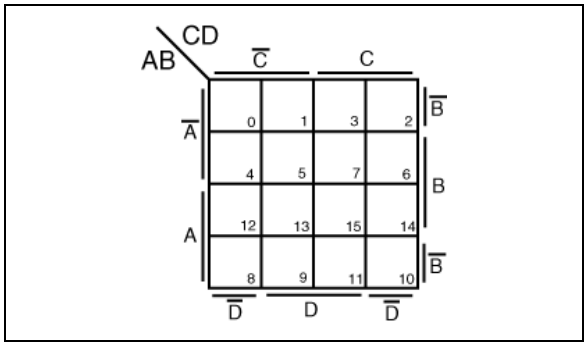


Figure 9.13: The blank 4-variable K-map; note the special numberings.

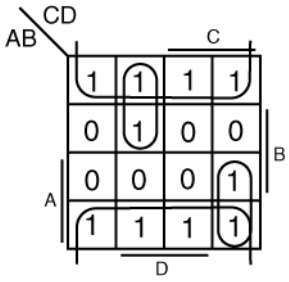
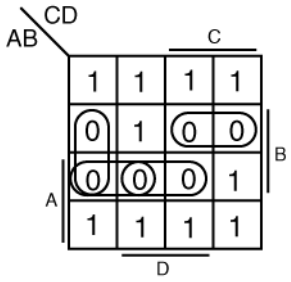
	
$F(A,B,C,D)=\sum(0,1,2,3,5,7,8,9,10,11,14)$	$F(A,B,C)=\prod(4,6,712,13,14)$
$F=\overline{B}+\overline{A}\overline{C}D+AC\overline{D}$	$\overline{F}=\overline{A}BC+ABD+AB\overline{C}+B\overline{C}\overline{D}$ $F=(A+\overline{B}+\overline{C})(\overline{A}+\overline{B}+\overline{D})(\overline{A}+\overline{B}+C)(\overline{B}+C+D)$
(a)	(b)

Figure 9.14: K-map grouping for SOP (a) and POS (b) forms.

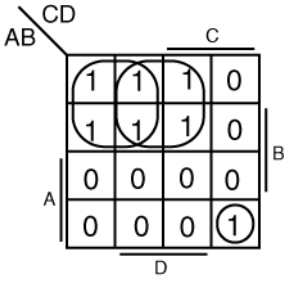
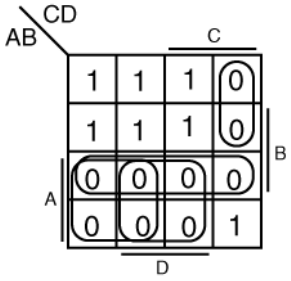
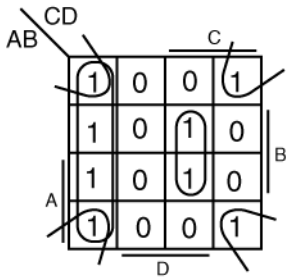
	
$F(A,B,C,D)=\sum(0,1,3,4,5,7,10)$	$F(A,B,C,D)=\prod(2,6,8,9,11,12,13,15)$
$F=\overline{A}\overline{C}+\overline{A}D+A\overline{B}C\overline{D}$	$\overline{F}=A\overline{C}+AD+AB+\overline{A}C\overline{D}$ $F=(\overline{A}+C)(\overline{A}+\overline{D})(\overline{A}+\overline{B})(A+\overline{C}+D)$
(a)	(b)

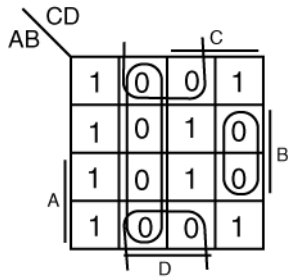
Figure 9.15: K-map grouping for SOP (a) and POS (b) forms.



$F(A,B,C,D) = \sum(0,2,4,7,8,10,12,15)$

$F = \overline{B}\overline{D} + \overline{C}\overline{D} + BCD$

(a)



$F(A,B,C,D) = \prod(1,3,5,6,9,11,13,14)$

$\overline{F} = \overline{C}D + \overline{B}D + BC\overline{D}$

$F = (C + \overline{D})(B + \overline{D})(\overline{B} + \overline{C} + D)$

(b)

Figure 9.16: K-map grouping for SOP (a) and POS (b) forms.

9.2.4 Karnaugh-Maps and Incompletely Specified Functions

As you'll find out later in your digital design career, there many instances where there are rows in a truth table (and thus cells in a K-map) that do not have a specified output. Functions such as these are considered incompletely specified functions. Although this may sound somewhat problematic, it generally helps in the function reduction process.

The lines in the truth table that have no specific output can be thought of as follows: if the input conditions associated with an incompletely specified row appear on the circuit inputs, you *don't care* what the circuit outputs are. What is good about this is that if the output in that row does not matter, you can assign it either a '1' or a '0' which has the potential effect of making your final equation "smaller", or more reduced. The potential for increased reduction is a result of assigning "don't cares" such that they make larger groupings.

The process of reducing functions that are incompletely specified, or as them are more commonly known, functions that contain *don't care* entries, is easier than it sounds. The assignment of 1's or 0's to the don't care cells in the K-map is done implicitly; you never actually make the assignments directly. The approach is to use a "-" in the associated K-map cells that are "don't cares". In this way, you can either choose to include the "don't cares" in your groupings or leave them out. Including the don't cares can help you make bigger groupings and thus increased reduction of the function.

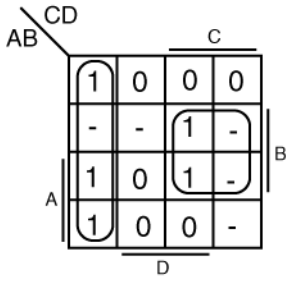
For example, if you have a function with don't cares in it and you are aiming to generate a reduced SOP equation, you group all the 1's in the map and use the don't cares to make your groupings larger[12]. In this case, while you are obligated to group all the 1's in the K-map, you should only group the "don't cares" if it makes your grouping larger which has the effect of making you product terms contain a fewer number of independent variables. You are not obligated to group the "don't care" entries.

A similar argument can be made for the case of generating a reduced POS form; in this case, you're grouping 0's and using the don't cares to help make the 0-groupings larger. Keep in mind that if you

[12] Keep in mind that the larger the grouping, the smaller the associated product term will be. This is why they call it reduction.

include a “don’t care” in a grouping of 1’s, you’re implicitly assigning the cell with the “don’t care” to a ‘1’; if you’re grouping the 1’s and you opt not to include a don’t care in a group, you’ve implicitly assigned a ‘0’ to that cell.

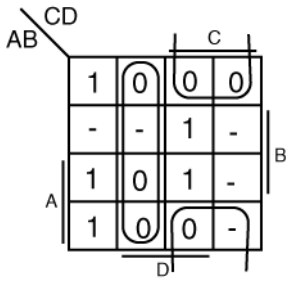
Figure 9.17 and Figure 9.18 show examples of function reduction with an incompletely specified function. In addition, worthy of note in Figure 9.17 and Figure 9.18 is the nomenclature used to specify the “don’t cares” for the compact minterm and maxterm forms. You’ll sometimes see this nomenclature, so consider committing it to memory.



$$F(A,B,C,D) = \sum(0,7,8,12,15) + \sum md(4,5,6,10,14)$$

$$F = \overline{C}\,\overline{D} + BC$$

Figure 9.17: The K-map groups for a reduced SOP form.



$$F(A,B,C,D) = \prod(1,2,3,9,11,13) \cdot \prod Md(4,5,6,10,14)$$

$$\overline{F} = \overline{C}D + \overline{B}C$$

$$F = (C + \overline{D})(B + \overline{C})$$

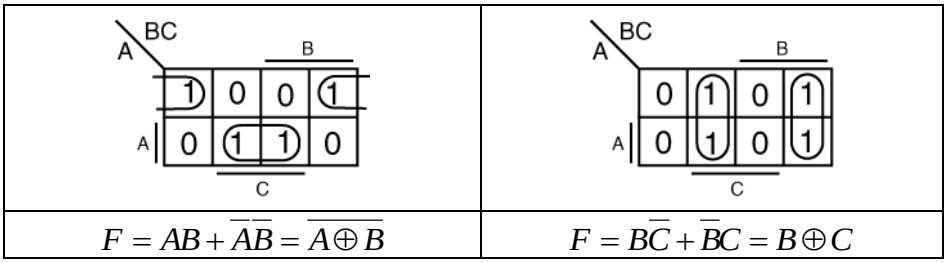
Figure 9.18: The K-map groups for a reduced POS form.

9.2.5 Karnaugh-Maps and XOR/XNOR Functions

As a final note on K-maps, they also can help you obtain XOR and XNOR functions from an equation reduced by normal K-map methods. Although there are explicit rules on how to take XOR-type functions directly from a K-map, these rules are once again long and complex. The preferred approach uses characteristics of the K-map groupings to ascertain whether the K-map contains XOR-type functions.

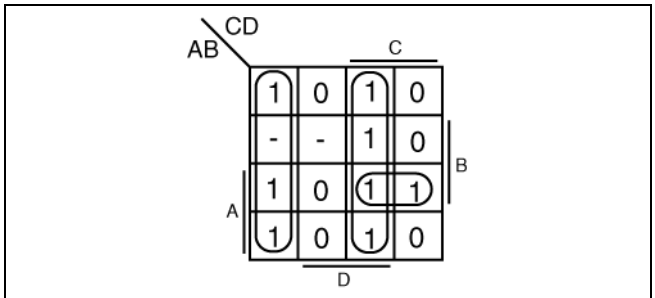
The approach to extracting XOR-type functions from K-maps is to, 1) use the K-map to alert you to the fact that a XOR-type function is present, and, 2) use Boolean algebra to factor the XOR-type function out of the equations resulting from reducing the K-map. The key to noticing whether a K-map contains a XOR-type function is to see that your groupings have formed some stripes or diagonals. If you see these conditions in your K-map, look closely at your final equation and factor out the XOR-type functions. A few examples should drive home this point.

Figure 9.19(a) shows an example of *diagonal groupings*. Once you note this, you'll know that the equation generated from applying standard K-map techniques can be further reduced as is shown in Figure 9.19(a). Figure 9.19(b) shows an example of striped groupings. Once noted, you can further reduce the K-map-generated equations as shown in Figure 9.19(b). Figure 9.20 shows two large striped groupings while Figure 9.21 shows two large diagonal groupings.



(a)	(b)
$F = AB + \overline{A}\overline{B} = \overline{A \oplus B}$	$F = B\overline{C} + \overline{B}C = B \oplus C$

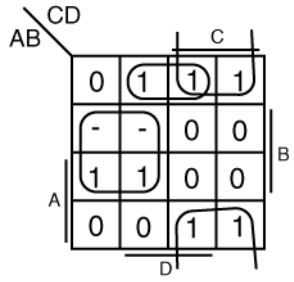
Figure 9.19: 3-Variable K-maps with diagonal (a) and stripes (b).



$$F = \overline{C}\overline{D} + CD + ABC$$

$$F = \overline{(C \oplus D)} + ABC$$

Figure 9.20: 4-Variable K-map with stripes.



$$F = B\overline{C} + \overline{B}C + \overline{A}\overline{B}D$$

$$F = (B \oplus C) + \overline{A}\overline{B}D$$

Figure 9.21: 4-Variable K-map with diagonals.

9.3 Function Form Transfer Matrix

We're to the point now where we've learned many different ways of representing functions. The two underlying truths are that 1) you rarely see some of these representations, and 2) you don't generally spend a lot of time going from one representation to another as it not typically done in modern digital design. However, Figure 9.22 may aid you in learning and understanding the representations.

In reality, Figure 9.22 represents a bunch of "rules". Rules in digital design are not good because if you become pre-occupied with following and/or memorizing rules, you loose sight of the fact that your main goal is to develop a basic understanding of digital design principles. The justification for providing Figure 9.22 is that it will hopefully, 1) remind you off all the ways to represent functions[13] and how they are related, and 2) it will get you past this early phase of a digital design course where we're not really doing much real digital design. The standard digital design problem at this phase of digital design is to do what you need to do to transfer between the various function representations. Yes, it's boring, but it's fairly straightforward to learn. Here are a few things to notice about Figure 9.22.

- The matrix in Figure 9.22 represents the "easy" paths between the various function representations. By easy, I mean, you could conceivably go from any one representation to any other representation if you thought enough about it or were interested in breaking a sweat.

- At this point, the only way to generate a reduced form is by using a K-map. There are actually other ways, such as applying Boolean algebra theorems, but life is simply too short for that. The key here is to realize that any time you hear the word "reduced", you automatically know you have to use a K-map.

- All of the lines drawn in Figure 9.22 are bi-directional arrows, except one. Note that if you have a VHDL model, it is not necessarily straightforward to go from a VHDL model to a reduced form. We'll be delving into VHDL later.

- There is no easy way to go between reduced SOP and POS forms. The best way to do this is to transfer the information back to the K-map first.

[13] Keep in mind, there are still a bunch more ways to represent functions that we'll wade through later.

- There truly is an easy way to go back and fourth between compact minterm/maxterm representation and VHDL. We'll get to those later when we do more VHDL.

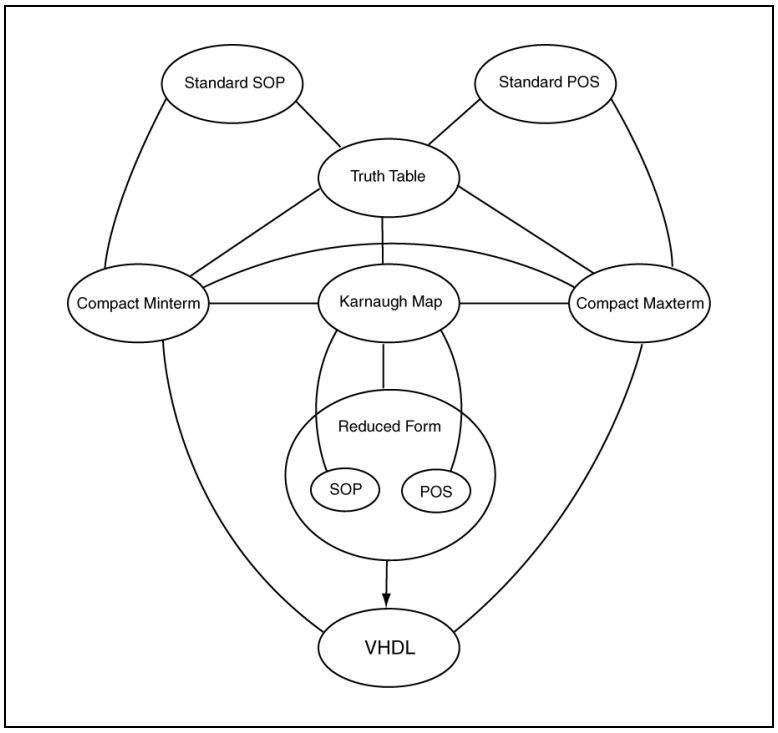
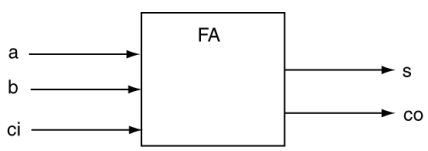


Figure 9.22: Function representation transform matrix (be sure to read the explanation for this).

Example 9-1: Full Adder

Re-implement full adder in reduced form using any gate you deem appropriate to reduce the overall gate count.

Solution: Figure 8.11 shows the black box diagram and the truth table for the full adder. We've derived and explain this previously so no more verbage is provided here.



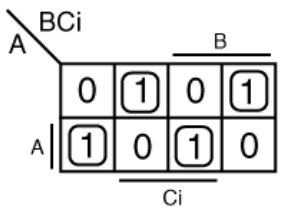
a	b	ci	Sum	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

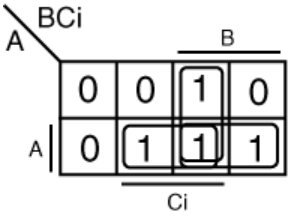
(a) (b)

Figure 9.23: The black box diagram (a) and the truth table for a FA.

Figure 9.24 shows the completed K-maps for both the Sum and Co outputs of the FA. Note that in Figure 9.24(a), there are not obvious reduction opportunities for the Sum output. However, since the K-map exhibits diagonals that practically jump out of the page, there are going to be some XOR type qualities we can factor out of the resulting equations.

Figure 9.25 shows the original equations taken directly from the grouping made in the K-maps. Note that the equation for the sum must be factored in order to extract the XOR functions they contain. Figure 9.26 shows the final circuit diagram for the full adder.





(a) (b)

Figure 9.24: The K-maps associated with the Sum (a) and Co (b) outputs for the FA.

$$Sum = \overline{A}\,\overline{B}Ci + \overline{A}B\overline{Ci} + A\overline{B}\,\overline{Ci} + ABCi$$
$$Sum = \overline{A}(\overline{B}Ci + B\overline{Ci}) + A(\overline{B}\,\overline{Ci} + BCi)$$
$$Sum = \overline{A}(B \oplus Ci) + A\overline{(B \oplus Ci)}$$
$$Sum = A \oplus (B \oplus Ci)$$

(a)

$$Co = AB + ACi + BCi$$

(b)

Figure 9.25: The paths to the reduced equations for the sum output (a) and the Co output (b).

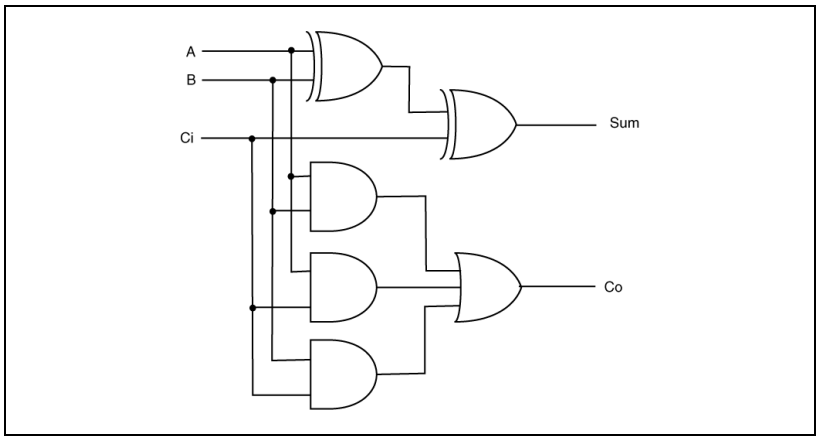


Figure 9.26: The final circuit diagram for the full adder.

Example 9-2: Special Switch Network

A circuit is required in order to control a network of switched. The circuit has four inputs: three of the switches are normal on/off switches while the fourth switch is an enable switch for the entire circuit. The circuit should have the following characteristics (this is a verbal model of the circuit):

- If the enable switch is off, the single output will be low (off)
- Otherwise, if any two (and only two) switches are one, then the output is high (on)
- Three switches will never be on simultaneously
- The circuit output is low under all other conditions

Solution: The first step in these types of problems is to draw a block diagram of the circuit. The circuit has four inputs and one output. In this step, it would be a great idea to give both the inputs and outputs intelligent looking names. Figure 9.27 shows the black box diagram for this circuit.

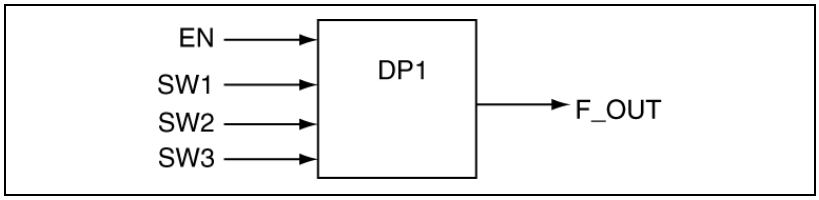


Figure 9.27: Black box diagram of circuit.

The next step will be to create a relationship between the circuit's outputs and the circuit's inputs. Keep in mind that this is a functional relationship in that each combination of digital inputs has only one unique value on the digital output. The best approach in this relatively not-too-complex circuit is to start with a truth table. Figure 9.28 shows the completed truth table for this example.

EN	SW3	SW2	SW1	F_OUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	-

Figure 9.28: Truth table for problem solution.

Now that you've completed the truth table, take a quick look at it and make sure that it actually makes sense. Note that in the truth table shown in Figure 9.28, the rows with the 1's in the output have EN asserted in each case. This means nothing is going to happen unless the enable input is a '1', which is in the problem statement. Also note that the final row is listed as a *don't care*; because it was stated that the condition will never happen. We take advantage of the fact that this condition never happen by assigning a *don't care* to the output associated with those input conditions. This will hopefully allow for a smaller final equation.

Equation 9-4 shows the resulting expression for the output. If you stare at this equation, you can see that it is somewhat intuitive based on the original problem description. As with all of these problems, you should stare at your final solution for a few minutes to verify that it's not totally wanky. Equation 9-4 does actually make a lot of sense.

$$F\_OUT = (EN \cdot SW3 \cdot SW2) + (EN \cdot SW2 \cdot SW1) + (EN \cdot SW3 \cdot SW1)$$

Equation 9-4: The expression for the problem solution.

Figure 9.29 shows the circuit model the implements a solution to the problem. This problem has a common digital input: the *enable* input. In many of these cases, it is possible to somewhat simplify the design by handling the enable input separately from the main problem. Figure 9.30 shows an alternate solution for this problem. Note that the enable controls the final output by having ultimate control over the AND gate on the output. This is an important feature in digital design: always keep it in mind.

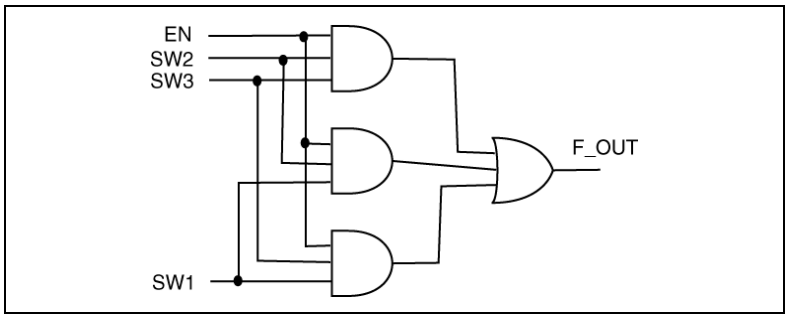


Figure 9.29: The brute-force logic solution to the problem.

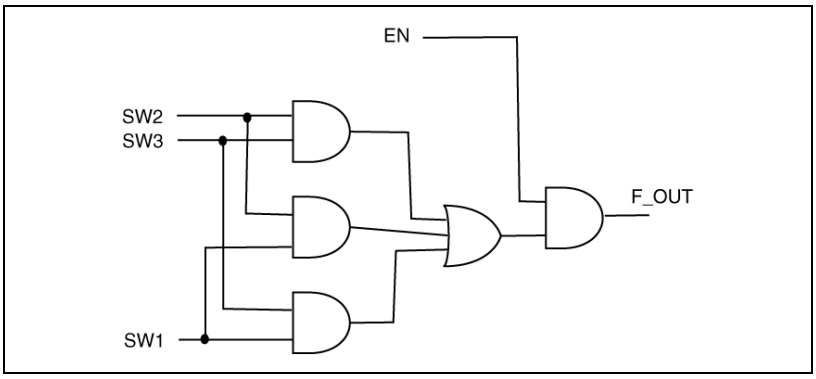


Figure 9.30: A more intuitive logic solution to the problem.

Chapter Summary

- Functions can be represented in many different forms. The forms presented in this chapter include standard SOP and POS forms, compact minterm and maxterm forms, and reduced forms.

- Reducing functions, or *function reduction* refers to the practice of making the function representation smaller without changing the overall input/output relationship of the function. We declare one circuit to be smaller than another circuit if it uses less gates and/or if the gates in the circuit contain less inputs. Implicit in this definition is that the two circuits are functionally equivalent, regardless of the number of gates or gate inputs.

- There are three main recurring themes present in modern digital design: 1) make your circuit smaller, 2) make your circuit faster, and 3) make your circuit consume less power. Rumor has it that you can only choose two of these options.

- Three and four-variable Karnaugh maps are used to reduce Boolean equation representations of functions. The K-map inherently generates a SOP form but can be also generate POS form by multiple applications of DeMorgan's Theorem.

- Incompletely specified functions, or don't cares as they are commonly known, are sometimes found in digital design. These outputs can be assigned such that the final function representation is more reduced.

- Karnaugh maps that contain "stripes" and/or "diagonals" indicated that XOR (or XNOR) functions are present. The final equation describing the circuit can be further reduced if the XOR-type functions are factored out. There are large sets of rules that can be applied to this process, but a better approach is to simply use your brain to see possible XOR functions in K-maps and Boolean factorization to flush out the terms.

Chapter Exercises

1) Write a reduced Boolean equation in SOP form for each of the following functions.

$F1(A,B,C) = \sum(1,3,4,6)$ $F2(A,B,C) = \sum(0,2,4,6,7)$

$F3(A,B,C) = \sum(3,4,5,6)$ $F4(A,B,C) = \prod(2,4,6,7)$

$F5(A,B,C,D) = \sum(0,2,4,6,8,9,12,13)$ $F6(A,B,C,D) = \sum(0,2,5,8,10,13,15)$

$F7(A,B,C,D) = \prod(4,5,6,12,13,14,15)$ $F8(A,B,C,D) = \prod(2,3,6,7,9,11,13,15)$

2) Write an expression for F in compact minterm form:
$\bar{F}(A,B,C,D) = \prod(2,10,11,12,13,15)$

3) Write an expression for $\bar{F}$ in compact maxterm form:
$F = (A,B,C,D) = \sum(2,4,5,6,7,8,10)$

4) Write an expression for F in compact maxterm form:
$\bar{F}(A,B,C,D) = \prod(4,5,6,10,11,12)$

5) Write an expression for F in standard SOP form:
$F(U,V,W,X,Y,Z) = \sum(2,23,59)$

6) Write an expression for F in standard POS form:
$F(U,V,W,X,Y,Z) = \prod(12,33,51)$

7) Implement F using a minimum of logic devices:
$F = (A,B,C,D) = \sum(0,1,4,6,8,9,10,11,13,15)$

8) Implement F using a minimum of logic devices:

$$F = (X, Y, Z) = \sum(0,2,5)$$

9) Convert the following standard POS equation to compact maxterm form:

$$F(A,B,C,D,E) = (A + B + \overline{C} + \overline{D} + \overline{E})(A + B + \overline{C} + D + \overline{E})$$

10) Write a reduced expression for F in POS form:

$$F(A,B,C,D) = ABD + B\overline{C}\,\overline{D} + \overline{A}BD$$

11) Write the reduced SOP expression for F given the following function:

$$F(A,B,C,D) = \sum(5,10,11,13,15) + \sum m_d(0,2,3,6,8,14)$$

12) Write the reduced SOP expression for the following function:

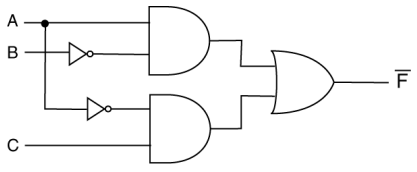
$$F(A,B,C,D) = \prod(2,6,10,11,12,14,15) \cdot \prod Md(1,5)$$

13) For the following equation:

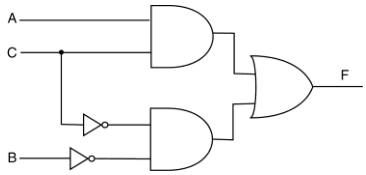
$$F(A,B,C,D) = \sum(4,5,6,7,9,11,12,14)$$

a) write the reduced SOP equation for F
b) write the reduced POS equation for F

14) Write an expression for F in compact minterm form that describes the following circuit:



15) Write an expression for F and $\overline{F}$ in reduced SOP form that describes the following circuit:



16) From the following compact minterm form:

$$F(A,B,C,D) = \sum(2,3,5,7,10,11,15)$$

a) write the reduced SOP equation for F
b) write the reduced POS equation for F

17) Convert the following expression to compact maxterm form in terms of F.

$$\overline{F}(A,B,C,D) = \sum(0,6,7,8,11,15)$$

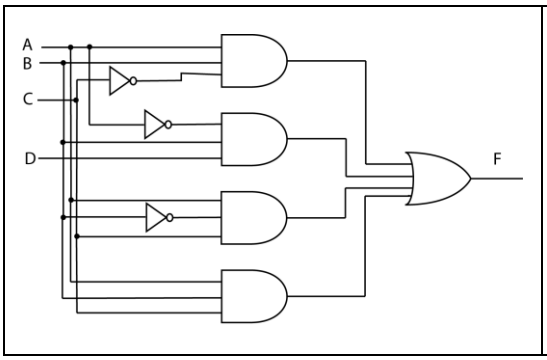
18) Write an expression for F in standard sum of products form (SOP).

$$\overline{F}(A,B,C,D) = \sum(2,3,4,5,6,7,8,9,10,11,12,13,14,15)$$

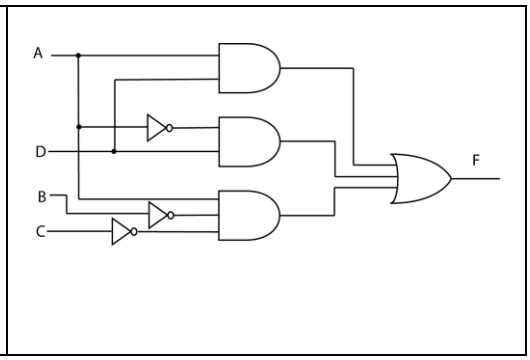
19) Write the reduced SOP expression for F given the following function:

$$F(A,B,C,D) = \sum(2,5,8,9,12,13) + \sum m_d(0,1,7,10)$$

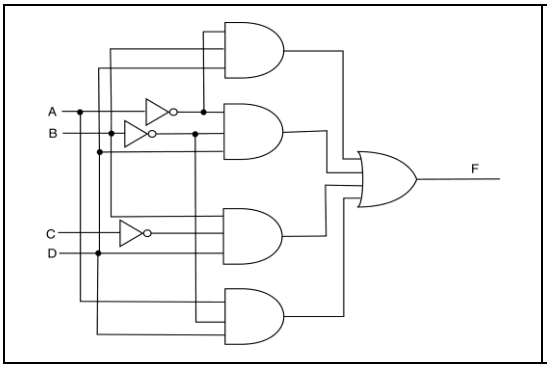
20) The equation describing the circuit below was not reduced before the circuit was implemented. Analyze the circuit and re-implement (draw the circuit) in reduced form using AND & OR gates and Inverters.



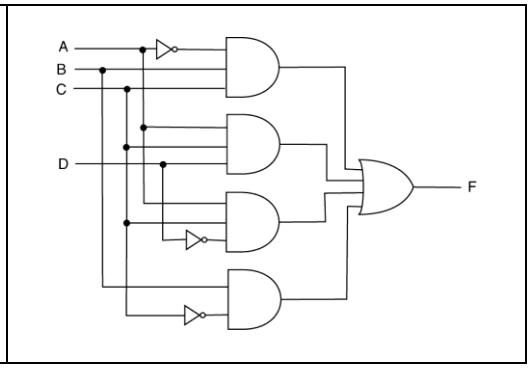
(a)



(b)

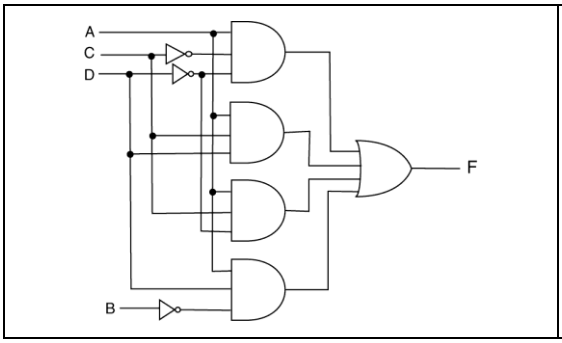


(c)

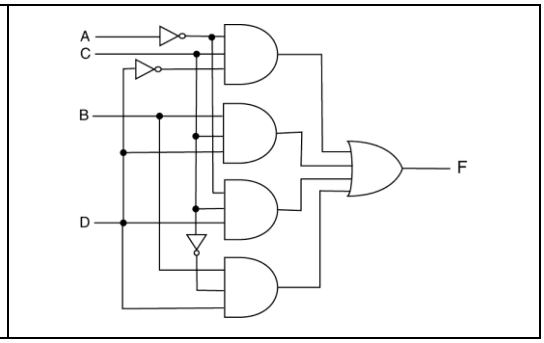


(d)

21) The equations describing the circuits below were not reduced before the circuits were implemented. Analyze the circuit and re-implement (draw the circuit) in reduced form using AND & OR gates and Inverters.



(a)



(b)

Design Problems

1) Design a circuit that has an output that indicates when the four-bit unsigned binary number on the input is a prime number. For this problem:

- assume an input value of "0000" will never occur (be sure to note this fact where appropriate)
- assume the decimal value of 1 is a prime number

Provide a block diagram, truth table, and a maximally reduced equation that models a solution for this problem.

2) Design a circuit that has an output that indicates when the three-bit unsigned binary number on the input is:

- Less than or equal to 5_{10}, and
- Greater than 1_{10}

For this problem, an input value of "000" will never occur (be sure to note this fact where appropriate). Provide a block diagram, truth table, a maximally reduced equation, and a circuit diagram that models a solution for this problem.

4) Design a circuit whose outputs represent the square of the two circuit inputs. Implement the associated output functions in reduced SOP form.

5) Design a circuit whose outputs represent the square root of the circuit's 4-bit inputs. Implement the associate output functions in reduced POS form. Round the output either up or down when necessary. Provide a truth table, a reduced Boolean equation and circuit diagram for you solution.

6) A given circuit has four inputs. Two of the inputs are considered the fractional portion of a binary number while the other two inputs are considered the integral portion of the binary number. The outputs of this circuit should represent a 2-bit binary number associated with the 4-bit input but with rounding up and down. In other words, if the input is greater or equal to 0.5, the output should represent the input rounded up. Otherwise, it output should represent the input rounded down to the nearest integer. Provide a truth table, a reduced Boolean equation and circuit diagram for you solution.

10 Chapter Ten

(Bryan Mealy 2012 ©)

10.1 Introduction

This chapter is one of the final chapters on more interesting ways to represent Boolean expressions. The approaches presented in this chapter are important because they are typically seen more often the many other approaches. You'll surely find that the forms presented in this chapter are actually quite useful. The added feature of this chapter is that you'll be doing some grunt-work with Boolean algebra; recall that this was a topic we skipped over in previous chapters.

One of the underlying notions of being able to represent functions in various forms is the notion that one form has some type of advantage over some other form. They idea here is that if I can find a functionally equivalent form that I can implement faster, requires less power to operation, cheaper, etc[1]., than that is the form I'm most likely going to use that form.[2]

Main Chapter Topics

- **CIRCUIT FORMS:** Previous chapters have presented various functionally equivalent representations of circuit. This chapter presents the theory behind generating several new forms and outlines when such forms are most useful. The new circuit forms presented in this chapter are definitely some of the most widely used representations of circuits.
- **MINIMUM COST CONCEPTS:** Being that there are many different ways to represent functions, the question arises when one representation should be used over another. This chapter outlines minimum concepts as they apply do function representations.

Why This Chapter is Important

This chapter is important because the circuit forms provide a significant amount of flexibility when it comes to representing functions. This flexibility allows you to implement function using the minimum possible cost.

10.2 Circuit Forms

[1] And also a lot of other reasons not listed here; hopefully you're getting the idea.
[2] And of course if you're into computer science, these forms can be used to help obfuscate your Boolean equation.

The term "circuit forms" is a somewhat common term in digital logic design vernacular. This term generally refers to the fact that any given digital logic function can be implemented using physically different circuits. Several of the previous chapters provided us with many different, yet functionally equivalent forms.

This section examines yet another flavor of circuit forms. For these circuit forms, the equivalent equations look distinctively different from each other, and the final circuit associated with these various forms appears distinctively different from each other. In the context of a digital system, the term functionally equivalent refers to the fact that the input/output relationship of the circuit is preserved but the implementation details are different.

There are many reasons why you would want to use one form over another. Generally speaking, one form is often desired over another; the more desired form is usually based on the notion of efficiency in that one form may require fewer gates and/or inputs than another form. This section discusses forms that are generated with successive applications of DeMorgan's theorem. This approach is somewhat standard and generates the most common forms of a circuit. In truth, there are about as many circuit forms as you could spend the time generating. In reality, there are only about four commonly used circuit forms; the good news is that you've already been working with several of these forms.

10.2.1 The Standard Circuit Forms

There are eight common (and easily derived) circuit forms; these forms are so common that we'll refer to them as the "standard circuit forms". If you examine a standard digital design textbook, you'll find that some textbooks actually list bunches of strange and wonderful circuit forms; we'll opt to stick to the standard eight types in this chapter. The nice thing about the standard forms is that they are all generated from successive applications of DeMorgan's theorem, so if you know how to use this theorem, you won't have to waste your time memorizing where the various forms come from.

Equations 1(a) and 2(a) of Table 10.1 show the compact minterm and compact maxterm forms of an arbitrary function, respectively. These two forms can be reduced using K-mapping techniques to the expressions shown in 1(b) and 2(b). These two expressions were generated from grouping the 1's of the circuit (left column) or the 0's of the circuit (right column). The resultant equations serve as the starting point to generate other forms. The following steps describe how to generate the set of eight standard forms from the two compact forms. Table 10.2 a written description of this procedure.

1(a) $F=\sum(1,4,5,9,10,11,13,14,15)$	2(a) $F=\prod(0,2,3,6,7,8,12)$
AND/OR Form	**OR/AND Form**
1(b) $F=AC+\overline{C}D+\overline{A}B\overline{C}$	2(b) $\overline{F}=\overline{A}C+A\overline{C}\,\overline{D}+\overline{A}\overline{B}\,\overline{D}$ 2(c) $\overline{\overline{F}}=\overline{\left(\overline{A}C+A\overline{C}\,\overline{D}+\overline{A}\overline{B}\,\overline{D}\right)}$ 2(d) $F=\overline{\left(\overline{A}C\right)}\cdot\overline{\left(A\overline{C}\,\overline{D}\right)}\cdot\overline{\left(\overline{A}\overline{B}\,\overline{D}\right)}$ 2(e) $F=\left(A+\overline{C}\right)\cdot\left(\overline{A}+C+D\right)\cdot\left(A+B+D\right)$
NAND/NAND Form	**NOR/NOR Form**
1(c) $\overline{\overline{F}}=\overline{\overline{AC+\overline{C}D+\overline{A}B\overline{C}}}$	2(f) $\overline{\overline{F}}=\overline{\overline{\left(A+\overline{C}\right)\cdot\left(\overline{A}+C+D\right)\cdot\left(A+B+D\right)}}$
1(d) $F=\overline{\overline{\left(AC\right)}\cdot\overline{\left(\overline{C}D\right)}\cdot\overline{\left(\overline{A}B\overline{C}\right)}}$	2(g) $F=\overline{\overline{\left(A+\overline{C}\right)}+\overline{\left(\overline{A}+C+D\right)}+\overline{\left(A+B+\overline{D}\right)}}$
OR/NAND Form	**AND/NOR Form**
1(e) $F=\overline{\left(\overline{A}+\overline{C}\right)\cdot\left(C+\overline{D}\right)\cdot\left(A+\overline{B}+C\right)}$	2(h) $F=\overline{\left(\overline{A}C\right)+\left(A\overline{C}\,\overline{D}\right)+\left(\overline{A}\overline{B}\,\overline{D}\right)}$
NOR/OR Form	**NAND/AND Form**
1(f) $F=\overline{\left(\overline{A}+\overline{C}\right)}+\overline{\left(C+\overline{D}\right)}+\overline{\left(A+\overline{B}+C\right)}$	2(i) $F=\overline{\left(\overline{A}C\right)}\cdot\overline{\left(A\overline{C}\,\overline{D}\right)}\cdot\overline{\left(\overline{A}\overline{B}\,\overline{D}\right)}$

Table 10.1: The generation of standard circuit forms by using DeMorgan's theorem.

AND/OR Form	OR/AND Form
The form in 1(b) is the AND/OR form and is referred to as the Sum of Products (SOP) form. This form is obtained from a K-map by grouping the 1's of the circuit's output and performing K-map reduction techniques. The individual groupings in the K-map form the product terms. The final function represents a logical summing of the associated product terms.	The form in 2(b) is obtained by applying K-map reduction techniques to the 0's of the circuits output. In this case, since the K-mapping was based on the 0's of the circuit, we obtain the complement of the function ($\overline{F}$). The expression is in AND/OR form but we'll massage it into a different form by writing an expression for F rather than $\overline{F}$ as is listed in 2(b) by complementing the expressions on both sides of the equal sign, which preserves the equality and produces the equation shown in 2(c). Dropping the double compliment on the left side of equality generates the equation in 2(d). An application of DeMorgan's theorem generates the expression on the right side of the equality. The equation in 2(e) shows the final OR/AND form which is also referred to as the Product of Sums (POS) form.
NAND/NAND Form	**NOR/NOR Form**
The form in 1(c) is obtained from the AND/OR form by double complimenting both sides of the equation in 1(b). Double complimenting each side of the equation preserves the equality of the expression. The double compliment on the left side of the equation 1(c) drops out. On the right side of equation 1(c), one of the compliments is used to DeMorganize the expression. The equation in 1(d) shows the NAND/NAND form of the expression; the term NAND/NAND refers to the fact that each of the individual product terms are complimented product terms (a NAND function). These individual terms are ANDed together and complimented which effectively changes it from an AND function to an NAND function.	The form in 2(f) is obtained from the OR/AND form by double complimenting both sides of the equation in 2(e). The double compliment on the left side of the equation 2(f) drops out. On the right side of equation 2(f), one of the compliments is used to DeMorganize the expression. The NOR/NOR form of the expression is shown in 2(g). This is referred to as NOR/NOR form because each of the individual sum terms are complimented (a NOR function). These individual terms are ORed together and complimented which changes it from an OR function to an NOR function.
OR/NAND Form	**AND/NOR Form**
The OR/NAND form shown in 1(e) is obtained by DeMorganizing the individual terms from 1(d) to change them from product terms to sum terms. The expression retains the overbar over the entire term.	The AND/NOR form shown in 2(h) is obtained by DeMorganizing the individual terms from in 2(g) to change them from sum terms to product terms. The expression retains the overbar over the entire term..
NOR/OR Form	**NAND/AND Form**
The NOR/OR form shown in 1(f) is obtained by DeMorganizing the entire OR/NAND form shown in 1(e). In this way, the overbar on the right side of the equals sign is distributed to the individual terms in the equation.	The NAND/AND form shown in 2(i) is obtained by DeMorganizing the AND/NOR form shown in 2(h). In this way, the overbar on the right side of the equals sign is distributed to the individual terms in the equation.

Table 10.2: Written description of the circuit forms and derivations shown in Table 10.1.

Yep, there sure are many forms out there. The good news is that the AND/OR, NAND/NAND, OR/AND, and NOR/NOR forms are definitely the most common forms. The relationship between these forms is nicer than you may be initially thinking after plodding through the math provided in Table 10.1. Let's look at the AND/OR form and it's relation to the NAND/NAND form.

Figure 10.1 shows the common AND/OR form circuit implementation. Note that in this implementation, overbars on the input signals replace the inverters in an effort to save time. The form shown in Figure 10.1 matches the equation shown in equation 1(b). Figure 10.2(a) shows the subsequent NAND/NAND circuit implementation as it appears in Equation 1(d). While the circuit implementation is correct in that only NAND gates are used in the implementation, it is somewhat misleading because it no longer resembles the AND/OR form that it originated from.

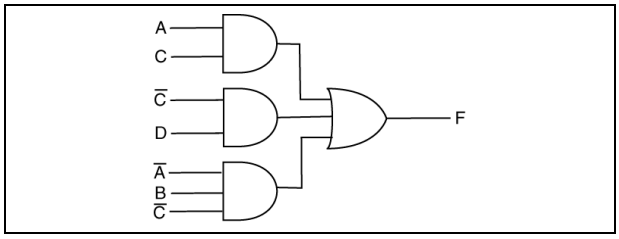


Figure 10.1: The beloved AND/OR form.

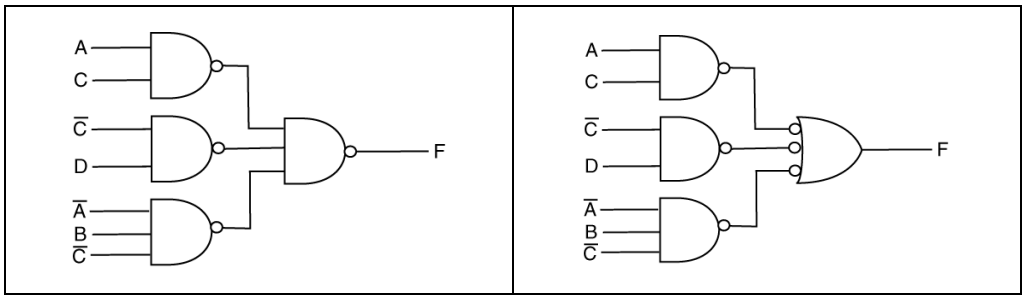
Although this is somewhat unexpected, there are two forms of NAND gates as Figure 10.2(b) indicates. We'll explain the details of this in a later chapter, so please accept it without explanation for now. By the time you get it explained to you, you'll be more able to handle some of the fine points. Since the right-most NAND gate of Figure 10.2(a) is actually implementing an OR function, you should use some type of OR-looking gate. Since this is an NAND/NAND form, the solution is to remove the right-most AND form of a NAND gate and replace it with an OR form[3] of a NAND gate as is shown in Figure 10.2(b).

Another thing that is disconcerting about the circuit of Figure 10.2(a) is that the bubbles "don't match"[4]. This is an indicator that something may be wrong. Although in the case shown in Figure 10.2(a) the implementation is truly correct, someone who is not familiar with the circuit may have doubts. In summary, you should not the similarities between the circuit of Figure 10.1 and Figure 10.2(b). Generally speaking, when you are asked to provide the circuit diagram for a function in NAND/NAND form, the best choice is to draw the circuit of Figure 10.1 and add the bubbles in the appropriate location to make the circuit appear like that of Figure 10.2(b). I like calling this the no-brainer approach to circuit forms[5]. Moreover, these are two of the most popular circuit forms. In the real world, the most widely used form is the NAND/NAND form.

[3] Don't worry about this wording for now.

[4] The "bubbles" are polarity indicators. This is a deep and often confusing subject (mixed logic) that we'll address in a later chapter. For now, just go with it and do your best to "match bubbles".

[5] In this case, the "no-brainer" thing is temporary; we'll fill in the details later. Not having brains is not necessarily a bad thing as brainlessness in the ranks of academic administrators is worn like a badge.



(a) (b)

Figure 10.2: The confusing (a) and totally clear (b) approach to NAND/NAND representations.

A similar type of argument can be made for the OR/AND and NOR/NOR circuit forms. The circuit implementation of the OR/AND form provided in Equation 2(b) is shown in Figure 10.3. Once again, the inverters have been omitted and are replaced with complemented input signals (don't try this at home). If we were to implement this circuit in the NOR/NOR form as listed in 2(f), you would end up with the circuit shown in Figure 10.4(a).

While the circuit shown in Figure 10.4(a) is technically correct, digital designers generally avoid this form because it is misleading, especially those digital designers who understand basic mixed logic principles[6]. A better NOR/NOR implementation appears in Figure 10.4(b). In this implementation, the right-most NOR gate is implemented using the AND[7] version of the NOR gate. The comforting thing here is that the NOR/NOR form implementation of Figure 10.4(b) is strikingly similar to that of Figure 10.3. Once again, if you're required to implement a function in NOR/NOR form, the circuit shown in Figure 10.4(b) is the preferred approach.

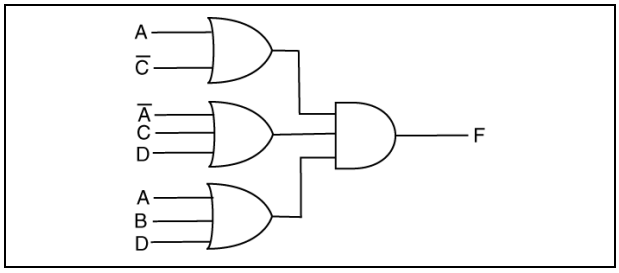
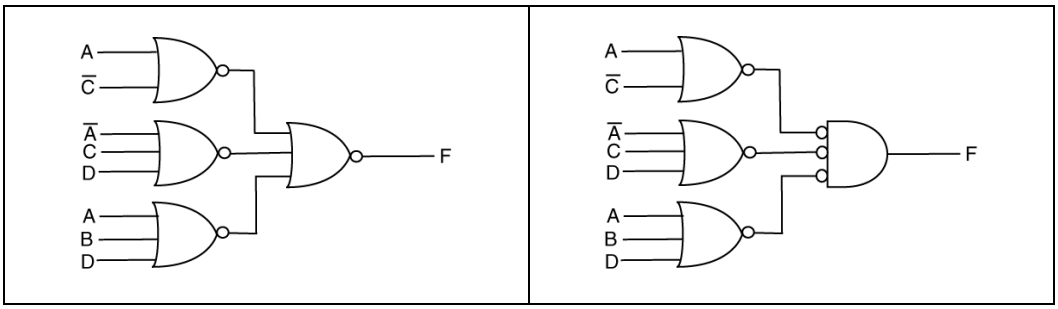


Figure 10.3: The good'ole OR/AND form.

[6] Mixed logic is an important concept that is covered in a later chapter.
[7] Once again, don't worry about this wording for now; this is another reference to mixed logic.



(a) (b)

Figure 10.4: The confusing (a) and totally clear (b) approach to NOR/NOR representations.

10.3 Minimum Cost Concepts

The desired approach to implementing circuits is to implement them at a minimum final cost. This turns out to be somewhat of an open-ended concept because minimum cost approach requires a proper definition of the word "minimum" before knowing what the minimum cost is. In reality, out there in the real world, there are about a bajillion definitions of the word "minimum" in terms of implementing a circuit. For beginning digital design courses, this definition usually refers to the number of gates used to implement a circuit and/or the number of inputs to the gates in the circuit.

The definition of "minimum" can also mean the number of integrated circuits (ICs) used to implement a circuit[8], or the number of transistors used in the ICs in the circuit etc. The definition of minimum cost is further obscured by the fact that your company may already have a bajillion ICs that are expensive but since you have nothing else planned for them, it would be cheaper to use them for your circuit because you can probably get them for a good price (namely free)[9]. It's all strange and somewhat obscure stuff. The final word on minimum cost is this: if someone tells you to apply minimum cost concepts to your design, make sure they provide you with an adequate definition of "minimum".

Up to this point, you've learned to implemented functions with many different forms. The forms primarily used are the reduced SOP and POS. When the concept of minimum cost arises, you generally examine both POS and SOP forms. But wait, it gets worse. Now that you know a bunch of other forms (such as NAND/NAND and NOR/NOR), you generally have to check all those forms also[10]. Generally speaking, unless given other specific directions, the form that uses the least amount of gates is generally the minimum cost solution. In particular, for given designs, you have the ability to use equivalent gates, which can sometimes reduce the overall device count (particularly the number of inverters used in a circuit).

The final word is this: the notion of minimum cost is primarily a relic from the past in terms of an introductory digital design textbook. The truth is that most everything is relatively cheap in the world of digital design. One thing that is not cheap is your salary. In the end, it's easier and cheaper to model a circuit using VHDL and not worry about the fine details of how it's constructed at a lower level. In most applications, if the circuit works, the world is happy. If the circuit needs to be optimal in terms of

[8] There are many ICs out there that contain different flavors of standard gates such as AND, OR, NAND gates, etc.
[9] This logic is something only a business major would understand.
[10] Though this seems somewhat excessive, it's not as strange as it seems. When you're building one circuit, saving a gate here and there is not going to make a lot of difference. However, if your circuit is going to go into production, and they're planning on building a million units of your circuit, the savings of one cent in a million circuits equates to as much money as the typical college president makes in a day.

minimum cost, then make it so. As for minimum cost concepts appearing in digital design texts, I suggest not getting to hung up with it.

Example 10-1: Minimum Cost Issues

Which of the eight standard forms would result in a minimum cost implementation in term of a) device count (gates and inverters), and, b) gate count for the following function. Assume you can use gates with any number of inputs.

$$F = \sum(1,4,5,9,10,11,13,14,15)$$

Solution: Lucky for us, this function is the same function that we used to describe the original eight forms. That means most of the work of the grunt work associated with this problem was done previously (definitely my type of problem). Going back and examining Table 10.1, you'll be able to generate the information provided in Table 10.3; it has all the info we need if we know where to look.

From Table 10.3, you can see the two best forms for the a) part of this example are OR/AND and NOR/NOR forms because they require six devices while other forms require more. For part b) all of the forms require the same number of gates; no particular form has any obvious advantage.

Form	a) Number of Gates & Inverters	b) Number of Gates only
AND/OR (SOP)	7	4
OR/AND (POS)	6	4
NAND/NAND (SOP)	7	4
NOR/NOR (POS)	6	4
OR/NAND	7	4
AND/NOR	8	4
NOR/OR	8	4
NAND/AND	8	4

Table 10.3: The whole enchilada for Error! Reference source not found..

Chapter Summary

- Circuit forms are used to implement logic functions using functionally equivalent expressions. Although there are an effectively infinite number of ways to represent a function, there are only a few standard ways. These standard ways are referred to as circuit forms and can be derived from repeated applications of DeMorgan's theorem. The most popular forms are SOP-type forms (AND/OR, NAND/NAND) and POS-type forms (OR/AND, NOR/NOR).

- Minimum cost concept pertains to the many functionally equivalent forms of circuits. When many circuit forms are possible, the circuit with the minimum cost is often the one that is implemented. Many factors can determine the minimum cost of a given function. If you are required to implement a minimum cost solution for a given function, the term "minimum cost" must first be explicitly defined.

Chapter Exercises

1) Write the following expression in OR/AND form:

$$F(A,B,C,D) = \overline{\overline{(\overline{A}+C)} + \overline{(\overline{B}+\overline{D})} + \overline{(A+\overline{C}+\overline{D})}}$$

2) Write a Boolean expression in NOR/NOR form that is equivalent to the following expression.

$$F = \overline{(\overline{A}+C+D)} + \overline{(\overline{A}+C+\overline{D})} + \overline{(\overline{A}+\overline{C}+\overline{D})}$$

3) Implement F using a minimum of logic devices (AND, OR, and inverters):

$$F = (A,B,C,D) = \sum(0,1,4,6,8,9,10,11,13,15)$$

4) Implement F using a minimum of logic devices (AND, OR, and inverters):

$$F = (X,Y,Z) = \sum(0,2,5)$$

5) Implement F using a minimum of logic devices (AND, OR, and inverters):

$$F = (A,B,C,D) = \sum(0,4,7,8) + \sum md(10,11,12,13,14,15)$$

6) Implement the following circuit (draw the circuit) in reduced form using only NAND gates.

$$F(A,B,C,D) = \sum(1,4,5,10,11,14,15)$$

7) Write a reduced equation in NAND/NAND that is equivalent to the following compact minterm form. Do not draw the circuit.

$$F(A,B,C,D) = \sum(0,2,4,8,11,15) + \sum md(10,14)$$

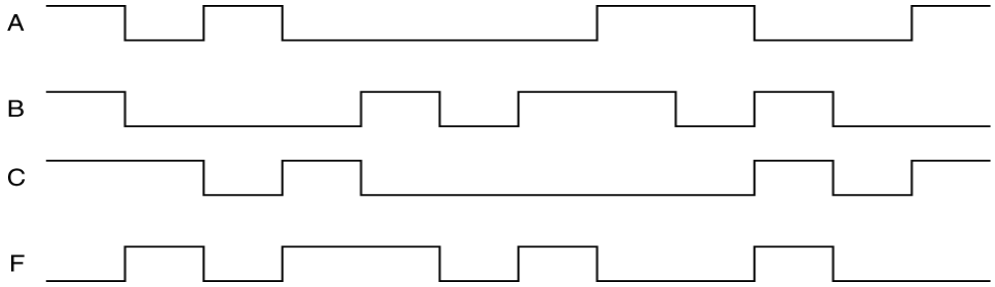
8) Generate an equation in reduced NAND/NAND form that is equivalent to the following Boolean expression. Do not draw the circuit.

$$F(A,B,C,D) = \overline{\overline{(A+C+D)} + \overline{(A+\overline{B}+C)} + \overline{(\overline{A}+\overline{B}+C)} + \overline{(\overline{A}+\overline{B}+\overline{D})}}$$

9) Generate an equation in reduced NAND/NAND form that is equivalent to the following Boolean expression. Do not draw the circuit.

$$F(A,B,C,D) = \overline{\overline{(A+C+D)} + \overline{(A+\overline{B}+C)} + \overline{(\overline{A}+\overline{B}+C)} + \overline{(\overline{A}+\overline{B}+\overline{D})}}$$

10) The following timing diagram completely defines a function F(A,B,C) that has been implemented on an 8:1 MUX. The control variables are A, B, and C (A is the most significant bit and C is the least significant bit) and the output is F. Write an expression for this function in reduced NAND/NAND form. Assume propagation delays are negligible.



Design Problems

1) Design a circuit that indicates special conditions on a 4-bit input. Consider the 4-bit input to be an unsigned binary number. This circuit has two outputs. One output indicates when the input is an even multiple of 4. The other output indicates when the input is greater than 2 and less than 11. Design this circuit any way you deem appropriate. Use nothing other than 4-input NOR gates in your final circuit.

2) Design a circuit that indicates specific number ranges on the 6-bit input. Consider the 6-bit input to be an unsigned binary number. This circuit has three outputs. One output indicates when the input is less than 32. Another output indicates when the input is between 32 and 48, not including either 32 or 48. The other output indicates when the input is greater than 47 and an even number. Design this circuit any way you deem appropriate. Use nothing other than 4-input NAND gates in your final circuit.

11 Chapter Eleven

(Bryan Mealy 2012 ©)

11.1 Introduction

There are three different approaches to performing digital design; up until now, we've only worked with one of these approaches: BFD (Brute Force Design). The approach is named based on the amount of effort required in order to complete even the simplest of circuits. The funny thing here is that we could only implement the simplest of circuits based on the basic limitations of BFD. The other name for BFD is "iterative design"; this term refers to the notion that we had to list (or iterate over) all the possible circuit inputs in order to assign outputs to the circuit. From there we employed a truth table (also a massively limited approach) in order to generate the final Boolean equation describing the circuit. BFD was limited for many reasons; in the end, its only real use was as a mechanism to introduce various digital design concepts.

In an effort to increase our efficiency as digital designers, we need to move onto other design approaches. In this chapter, we'll be dealing with our second design approach: IMD, or "iterative modular design". This approach is somewhat limited also, but it's really useful in some situations. Probably the best part about IMD is that it provides a great vehicle for presenting three different standard digital modules.

Main Chapter Topics

- **ITERATIVE MODULAR DESIGN (IMD):** This chapter introduces the notion of iterative modular design in the context of a standard digital circuit.
- **RIPPLE CARRY ADDERS**: A standard digital circuit that adds two digital values of arbitrary length.
- **COMPARATORS**: A standard digital circuit that compares two digital values of arbitrary length.
- **PARITY CHECKERS & PARITY GENERATORS**: A standard digital circuit that is commonly used for error detection in digital values.

Why This Chapter is Important

This chapter is important because it introduces the concept of "iterative modular design" (IMD). This chapter uses the IMD approach to design four standard digital circuits: the ripple carry adder (RCA), the comparator, the parity generator, and the parity checker.

11.2 Iterative Modular Design Overview

The main push behind IMD is the notion that we want to move away from the limits presented by BFD. These limits are inherent in both truth tables and Karnaugh maps. Because truth tables and K-maps were directly involved with generating an equation that described the circuit's operation, the main utility of IMD is that it decouples the digital designer from the need generate Boolean equations as part of designing digital circuits. You can argue that you can use a computer to perform Boolean reduction rather than a truth table and/or K-maps, but BFD is still a massively limited approach. The nice thing about IMD is that it does not directly require the use of Boolean equations, which have been required with current approach to digital design.

As you may guess, there are two separate aspects to IMD as implied by the name itself. The first aspect is the "modular" part of IMD. The implication and allusion here is that IMD uses previously designed modules as part of the design. In this context, a module is a black box model of something that was modeled previously at a low level and can now be used at a higher level in as a part of another design. There are many standard digital modules out there we can draw upon; the main two we've discussed so far are the half adder and full adder.

The other aspect of IMD is the "iterative" part. This means that something in digital designs based on IMD will be done repeatedly (thus "iteration"). In the context of IMD, the thing that is going to be iterated is the modular part of IMD, or the modules. In the end, IMD involves using pre-designed modules in an iterative manner in order to create circuits that do not require Boolean equations to model. Lastly, and maybe more importantly, IMD provides out first introduction to hierarchical digital design.

11.3 Ripple Carry Adders (RCA)

One of most commonly done things in digital and computer-land is arithmetic. As you may or not be aware of, math operation are often performed by computers; and since computers are primarily large digital circuits, it should be no surprise that tons of effort in digital-land is put into designing and implementing circuits that do math.

There are two classes of people out there: people who design mathematical circuits and people who use pre-design mathematical circuit. In reality, people get PhDs for designing new and more efficient circuits that perform some required calculation faster and better than other digital circuits. The reason this is so important is that computers generally spend a major portion of their computer power doing mathematical operations. The result is if you can do a certain math operation more efficiently or with a smaller digital circuit, you've saved time on the circuit level (so you can do more operations) and/or you've saved space (so you can include other circuitry to do more stuff) and probably power (so you can play games on your phone for a longer period of time before the battery needs charging).

This section looks at only one variety of mathematical circuit: the ripple carry adder, or RCA. Though you can make this circuit also do subtraction without too much effort (a topic for another chapter), this particular circuit is rather limited for reasons we'll mention later. The RCA does present a great vehicle to introduce iterative modular design.

Our first introductions to adding bits were the HA and FA. These circuits were severely limited because they could only add one bit worth of information[117]. Recall that the truth table for the FA had eight rows, which is pushing the limit of comfortability as far as truth tables and the associated Karnaugh

[117] Specifically, the result was one bit. The HA added two 1-bit values while the FA added three 1-bit values. Both devices had a "sum" and "carry" output.

maps go. The main push behind IMD is the notion that we want to move away from the limits presented by BFD by removing the need to use Boolean expressions directly in the design process.

The RCA is one of the standard digital circuits in digital design[118]. The RCA is an "n-bit adder", which is a circuit that adds two n-bit numbers and provides a result. We refer to it as an "n-bit adder" because the design can be general and thus we have no need to directly specify the value presented by "n". The RCA is built iteratively from our selection of 1-bit adders, namely the HA and FA. Without too much blather, we'll describe the RCA in the context of an example problem.

Example 11-1: The Ripple Carry Adder

Design a 4-bit Ripple Carry Adder (RCA). Each bit of this RCA should be represented by either a HA or FA.

Solution: The importance of this problem can't be overstated. First, it involves the design of a RCA, which is one of standard circuits in digital design-land. Secondly, it provides our first excursion into IMD and subsequently our first design that does not directly involve Boolean equations. Lastly, it's a circuit that is useful and instructive in many ways, as you'll see in the following discussion. In addition, worthy of note here is that we'll sort of show you most of the design and then discuss it; it would not be feasible for you to generate this design on your own at this point. Once you see the IMD technique, you'll then be able to apply it to other designs.

The starting point of this design is to define the inputs and outputs. We the RCA is a 4-bit adder, but what exactly are the inputs and outputs. The inputs include two 4-bit values; the output includes a 4-bit result and a 1-bit "carry" output. This carry output is often referred to as the "carry-out" or "co". The first step in this design is once again to draw a black box diagram of the final circuit. Figure 11.1 shows a black box diagram that conforms to this problem's specifications. In Figure 11.1, the two 4-bit input operands are represented using bundle notation, as is the 4-bit sum output. A single bit is used to represent the carry-out value and is referred to as the "co".

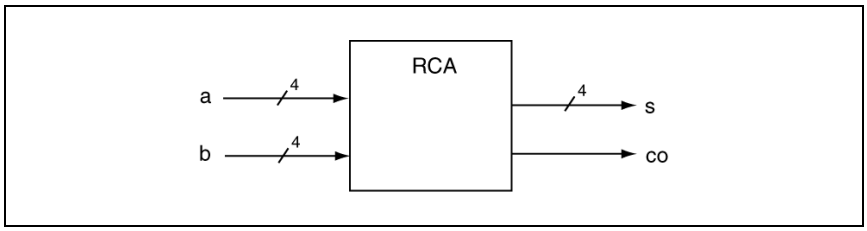


Figure 11.1: Black box diagram for the ripple carry adder.

From this, if we were to use out standard design approach, we would face creating a truth table and stuffing the output variables with values that solved the given problem. This approach presents a dilemma because, as shown in Figure 11.1, this problem contains eight inputs. Representing every possible combination of these eight inputs in a truth table would require a truth table of 256 rows. While this would be wholly possible, doing so would represent the ultimate grunt work. But it gets worse:

[118] This is not exactly true; it's true enough for now though. In reality, when you're doing digital design, you'll be using "n-bit adders" and you may not have any direct knowledge regarding how the adder is implemented.

reducing the results in an eight variable K-map would requires many years of mediation at some temple high in the Himalayas.

But let's take an approach that leverages the fact that we've already design some one-bit adders (named the HA and the FA). This problem requires that we design a 4-bit adder, but we've already designed two different types of 1-bit adders. Why not assemble the 1-bit adders in such a way as to create a 4-bit adder? Once again, this approach represents a standard digital design approach that you'll be using quite often in the future. Figure 11.2 shows the final solution for this problem. Bunches of important comments are sure to follow.

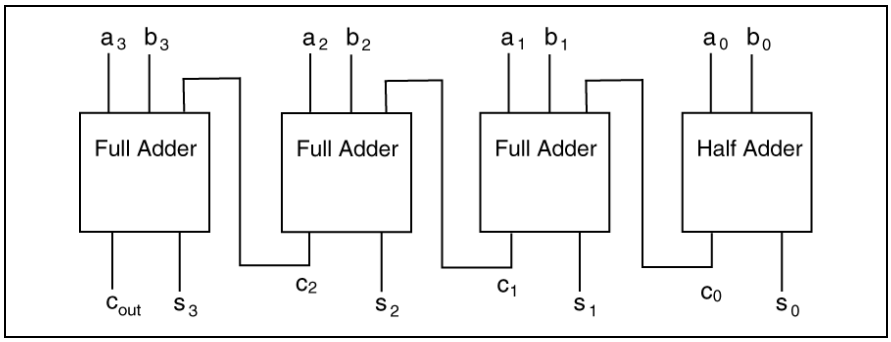


Figure 11.2: Black box diagram for a 4-bit Ripple Carry Adder.

Here are those promised important points:

- How does this circuit work? There are four 1-bit adders connected in a special manner. To ensure the correct answer on the circuit's outputs, each bit 1-bit adder must generate the "correct" values. While the **a** and **b** inputs are understood to be immediately available, the carry-outs are dependent upon the carry-ins from the previous bit location moving from right to left (except for the HA). In other words, generating the correct second-from-right sum bit is dependent upon the carry-out from the HA. In general, the correct values on each sum bits must wait until the carry-out has been generated from the lower-order 1-bit adder. The reason this circuit is referred to as a *ripple carry adder* or RCA, is that the carry must "ripple" from the lower-order adders to the higher-order adders (or right-to-left in Figure 11.2).

- Figure 11.2 uses weightings associated with each bit location. These weightings are implied by the numbering used for the inputs and sums. The higher the number "index", the higher the weighting. The s_3 output bit is referred to as the most significant bit (MSB) while the s_0 is referred to as the least significant bit (LSB). Beyond that, this adder is assumed binary in nature and uses the standard weighting associated with binary numbers.

- We completed this design without using a truth table. That being the case, we also completed this design without using any K-maps or Boolean equations. We essentially completed this design on a higher level than we completed previous designs. In other words, we completed this design exclusively using previously designed modules (namely the HA and FA). The design is modular in that we used previously designed modules; the design is iterative in that we placed a number of the previous design modules such that our original design specifications were satisfied.

- The notion of the "carry out" in many cases (and in this case) can be used as the "fifth bit". In essence, though we added two 4-bit unsigned numbers, we actually obtained a 5-bit result. There will be many times where you'll find a good use for this bit.

One of the notions we discussed regarding the RCA was the fact that it is sometimes referred to as an "n-bit adder". The reason for this is that if we suddenly wanted an 8-bit adder, we simply add four more FAs to the 4-bit RCA design. The act of "adding four more FAs" to the design is simple when you do it, but is still massively powerful and somewhat advanced technique[119]. And, as you'll see in the next example problem, it's not always the best approach to making an adder of greater width.

Example 11-2

Outline the steps you need order to design an 8-bit RCA using two 4-bit RCA circuits. State any assumptions and make any changes you may need to the 4-bit RCAs.

Solution: Although you may not have noticed it in Example 11-1, the "thing" that allowed you to increase the width[120] of your 1-bit adder was the fact that the FA was a 1-bit adder that added three different bits together and generated a 1-bit result. The key to RCA success was taking the carry-out from a bit location of lower significance and including it in the addition operation of the next bit location of higher significance. However, since the lowest-order bit, or, the LSB, only required a one-bit adder with two inputs because there would be no carry-in into that bit location.

Using a HA in the lowest order bit location saved some hardware, but that's about it. However, because of the HA, we are not able to "cascade" the RCA with other RCAs which would allow us to build RCA of greater bit-widths without too much trouble.

The solution to this example would be to substitute a FA for the HA shown in Figure 11.2. This solution omits some of the details, but Figure 11.3(a) shows the 4-bit RCA module while Figure 11.3(b) shows the black-box diagram for the solution of this example. Additionally, Figure 11.4 shows the final solution to this example with some details included.

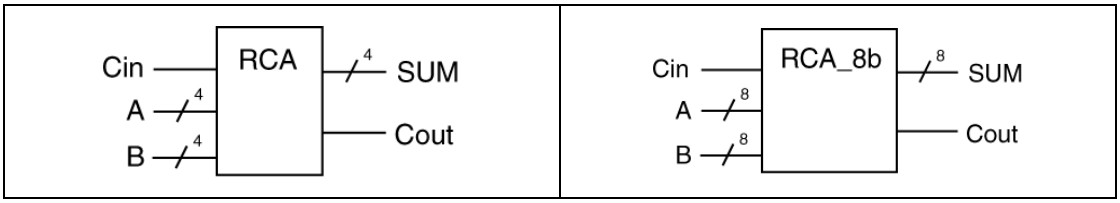


Figure 11.3: The 4-bit RCA module (a) and the 8-bit RCA module (b).

There are a few worthy things to notice in Figure 11.4. There is actually a lot of new stuff going on in this problem; these things are well worth describing.

[119] Accordingly, you now have the ability to design a 128-bit RCA if you really wanted to impress your friends.
[120] In this context, width refers to the number of bits contained in the input operands to the adder.

- There is no notion of HAs and FAs; this is because the model in Figure 11.4 is at relatively high level. At this point, we assume you know what a HA and FA is.

- It is a common assumption in digital-land to include a "carry-in" in RCAs. In this case, it is assumed that the lowest-order bit uses a FA instead of a HA.

- Parenthetical notation shows that the total number of input bits for the two operands; these are subsequently divided between the two individual 4-bit RCAs. In this case, you always need to provide some type of notation to indicate how the routing of the associated signals; you should never leave anything to guessing[121]. The same style of routing is use for the "sum" output.

- The "Cin" input to the lower-order RCA is "tied to ground". Generally speaking, it is requirement that every input in every schematic diagram is accounted for by connecting it to a signal, or assigning it a known and constant value such as '1' (power) or '0' (ground)[122].

- The reason this circuit works as an 8-bit adder in this cascade formation is that the "carry-out" form the lower-order RCA connects to the "carry-in" of the higher-order RCA. This is common in digital-land also as many ICs allow you to connect many of the same ICs together to increase the overall width (or length in some cases) of the given signal. Once again, the notion of connecting things together in this manner is referred to as cascading.

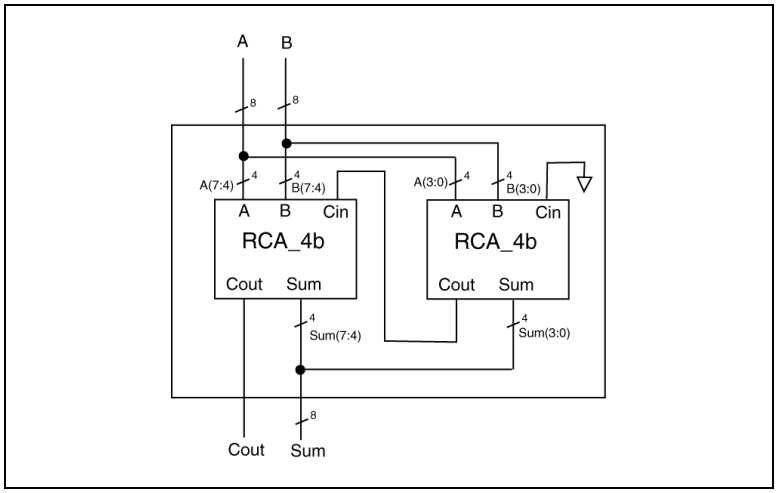
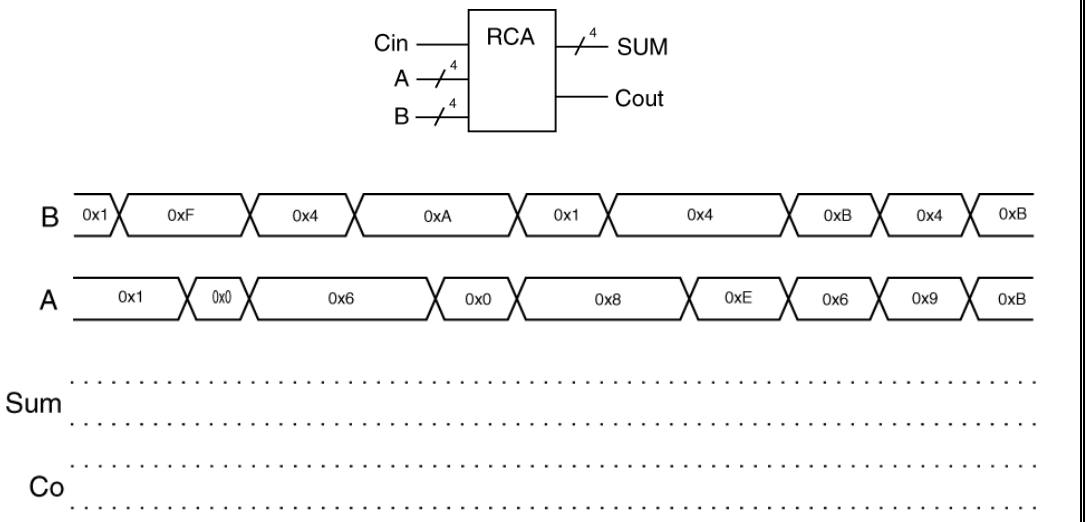


Figure 11.4: The high-level solution for Example 11-2.

[121] As will all problems you do in any context, you should always state any assumptions you make with the problem. This notion includes annotating the schematic diagram.
[122] This is super important; don't forget it.

Example 11-3

Use the block diagram of the 4-bit RCA shown below to complete the accompanying timing diagram. For this problem, assume the Cin input is always '0'.



Solution: This is a solution that you'll have to convince yourself of what is going on by examining the timing diagram in Figure 11.5. A few things to recall are that when you add two 4-bit binary numbers, you essentially end up with a 5-bit binary number with the carry-out being the most significant bi (MSB). The possible range for a 4-bit binary number is 0x0 to 0xF (equating to 0 to 15 in decimal). The solution shown in Figure 11.5 lists the bundle values in hex format; the carry-out is not a bundle.

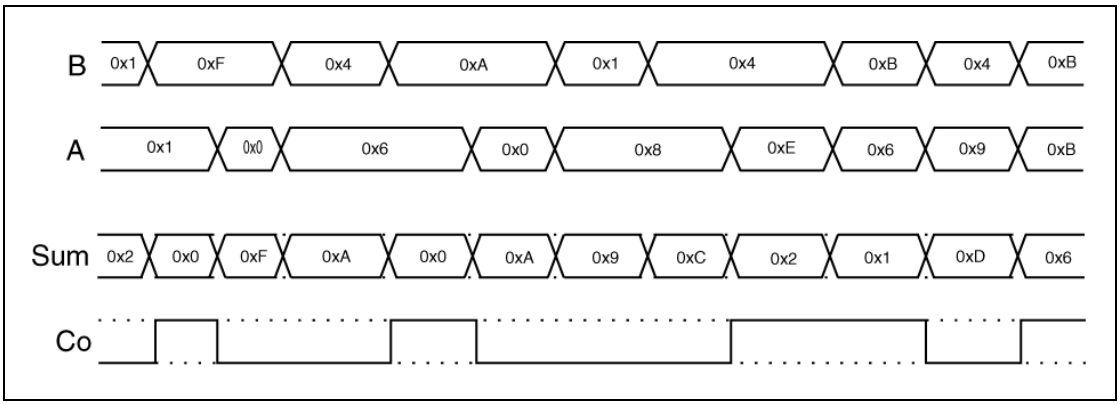


Figure 11.5: The solution to Example 11-3.

11.4 Comparators

The comparator is a commonly used device in digital land and is considered one the standard digital circuits. One of the nice things about a comparator is the fact that modeling a comparator in VHDL is

effortless while implementing the functionality in low-level logic is a giant pain it the arse. However, since the comparator is a standard digital module, you need to make sure you understand exactly what they're made of. Moreover, the derivation of the standard gate-level implementation of a comparator provides you with some useful practice dealing with exclusive OR-type functions. One of the other nice features about a comparator is the fact that it presents another chance to apply the iterative-modular design technique. We once again introduce the comparator with an example.

Example 11-4

Design a circuit that compares the values of two 2-bit inputs and indicates when the input values are equal.

Solution: Although the problem description does not state it directly, the circuit we need to design is referred to as a "2-bit comparator". For this solution, let's first design one using the BFD, or iterative-based (truth table) approach. This example states that we'll be designing a 2-bit comparator. In comparator lingo, a 2-bit comparator is a device that compares two 2-bit binary numbers; the single output of this circuit indicates when the two 2-bit inputs are equivalent. For this example, let's restrict the two 2-bit inputs to be unsigned binary numbers[123].

Step one in this design is drawing the black box; Figure 11.6 shows the result of this step. Note that in Figure 11.6, the circuit uses bundle notation to show that both the A and B signal are actually comprised of two bits each.

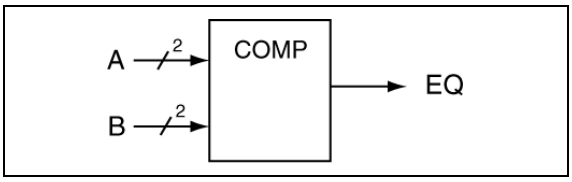


Figure 11.6: The black-box diagram for the 2-bit comparator.

Step two the design process is generating a truth table and entering the output values in such a way as to provide a solution to this problem. Since this problem has two 2-bit inputs, the truth table will have 2^4 or 16 rows. Figure 11.7 shows that we've arbitrarily listed the A inputs as the two left-most columns in the truth table. In Figure 11.7, the A1 and B1 inputs have a higher weighting (in terms of the weights of the digits) than the A0 and B0 inputs[124]. Figure 11.7 shows the completed the truth table and indicates when the two inputs are equal. In other words, the EQ output lists a '1' when the A and B inputs are equivalent. Figure 11.8 shows the associated K-map.

[123] It's OK to assume this since it was not explicitly stated in the problem and does not matter anyway.

[124] The reality is that the inputs could be placed in different columns the truth table; as long as you're consistent with the number values, all choices will lead to the same answer.

A1	A0	B1	B0	EQ
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 11.7: The truth table for the 2-bit comparator.

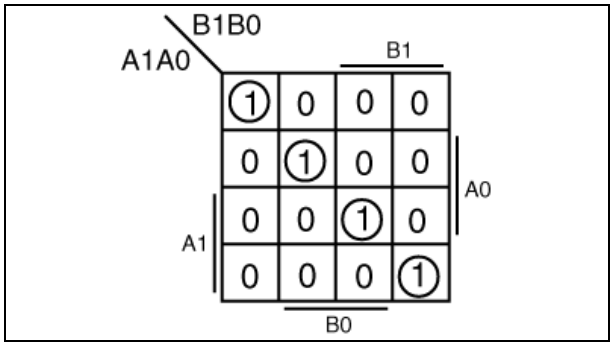


Figure 11.8: The K-map for the 2-bit comparator.

At first glance, the K-map in Figure 11.8 seems to have no opportunity for reduction since no groupings larger than one cell can be formed. The key thing to realize in this K-map is that there seems to be an opportunity to extract exclusive OR functions from the resulting equations since the groupings in Figure 11.8 seem to have diagonal components to them. Once you notice this attribute of the K-map, you can then continually factor the equation as shown in Figure 11.9.

Figure 11.9 shows an important derivation; you should make sure that you completely understand every step of it because you'll occasionally need to perform such algebraic manipulations out in digital design-land[125]. The nice thing about this derivation is that it primarily uses factoring as opposed to

[125] One thing you may want to try for this problem is to change the column-order of the independent variables and verify that you arrive at an equivalent Boolean expression.

Boolean algebra theorems[126]. The important thing here is noticing the relationship between the final equation of Figure 11.9 and the circuit implemented in Figure 11.10.

(a)	$F = (\overline{A1} \cdot \overline{A0} \cdot \overline{B1} \cdot \overline{B0}) + (\overline{A1} \cdot A0 \cdot \overline{B1} \cdot B0) + (A1 \cdot \overline{A0} \cdot B1 \cdot \overline{B0}) + (A1 \cdot A0 \cdot B1 \cdot B0)$
(b)	$F = (\overline{A1} \cdot \overline{B1})(\overline{A0} \cdot \overline{B0} + A0 \cdot B0) + (A1 \cdot B1)(\overline{A0} \cdot \overline{B0} + A0 \cdot B0)$
(c)	$F = (\overline{A1} \cdot \overline{B1})\overline{(A0 \oplus B0)} + (A1 \cdot B1)\overline{(A0 \oplus B0)}$
(d)	$F = \overline{(A0 \oplus B0)} \cdot (\overline{A1} \cdot \overline{B1} + A1 \cdot B1)$
(e)	$F = \overline{(A0 \oplus B0)} \cdot \overline{(A1 \oplus B1)}$

Figure 11.9: The ugly details of the final equation derivation for the 2-bit comparator.

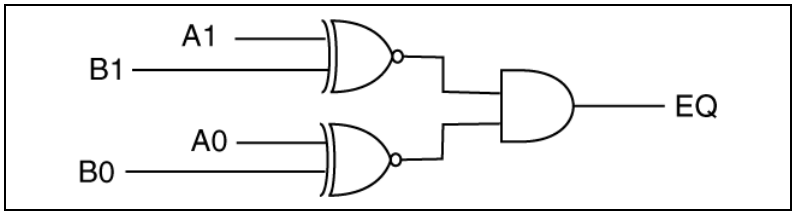


Figure 11.10: The final circuit for the 2-bit comparator as equation (e) in Figure 11.9.

Although the circuit shown in Figure 11.10 seems to be nothing special, it implicitly indicates a possibility to apply the iterative modular digital design technique. First, apply some horse-sense to understanding this circuit. What the circuit is saying is that each of the bits of the same weighting must be equal in order for the two numbers to be equal. In terms of the listed hardware, the AND gate is only satisfied when each of its inputs are a '1'. In this circuit, each of the inputs to the AND gate are an output of the individual XNOR functions. Recalling that an XNOR function is sometimes considered an equivalence gate, note that each of the bit positions being compared must be equivalent in order for the final number to be equivalent.

In the end, this problem was not too bad. Then again, it was only a 2-bit comparator; being that there were only four inputs, an iterative approach was borderline doable. But really, who has any use for a 2-bit comparator?

Example 11-5

Design a circuit that compares the values of two 4-bit inputs and indicates when the input values are equal.

Solution: For this problem, the circuit has two 4-bit inputs for a total of eight inputs. If we were to take the same approach as the previous example, we would require a truth table having eight independent

[126] Yet more proof that you can in fact be a good digital designer without scars left from applying endless Boolean algebra equations.

variables or 256 rows (2^8). Would this be possible? Yes. Would anyone really do it? No[127]. The key here is realizing that to make a 4-bit comparator, you simply need to add XNOR gates that will compare each of the added bit positions. This approach is a classic application of the iterative-modular (because you're using the same element over and over again) design approach[128]. The reality in digital design is that anytime you can apply the iterative modular approach, you'll be saving yourself a bunch of time. Figure 11.11 shows the final circuit diagram for a 4-bit comparator.

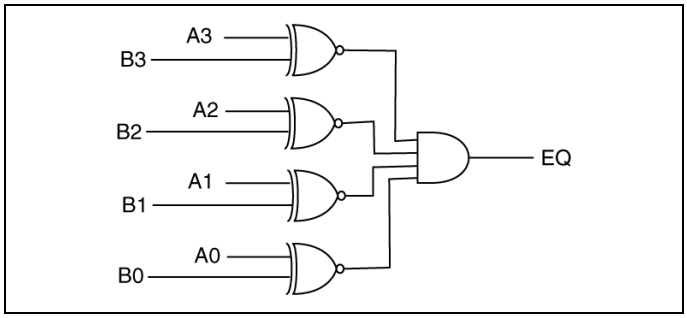
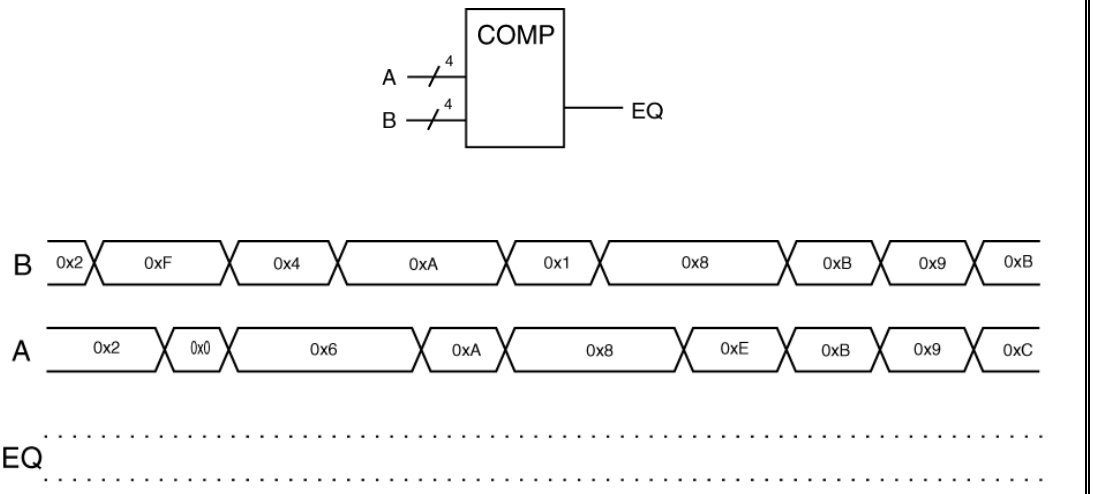


Figure 11.11: The final circuit for a 4-bit comparator.

Example 11-6

Use the black box diagram provided below to complete the accompanying timing diagram.



Solution: once again, a diagram shows the solution to this problem without a significant amount of verbal description. Check out Figure 11.12 for all the gory details.

[127] An academic administrator probably would, however, because this approach would require a lot of wasted time and effort; wasting time and effort is something academic administrators are really good at.
[128] Recall that the ripple carry adder was previously designed using the iterative-modular design approach.

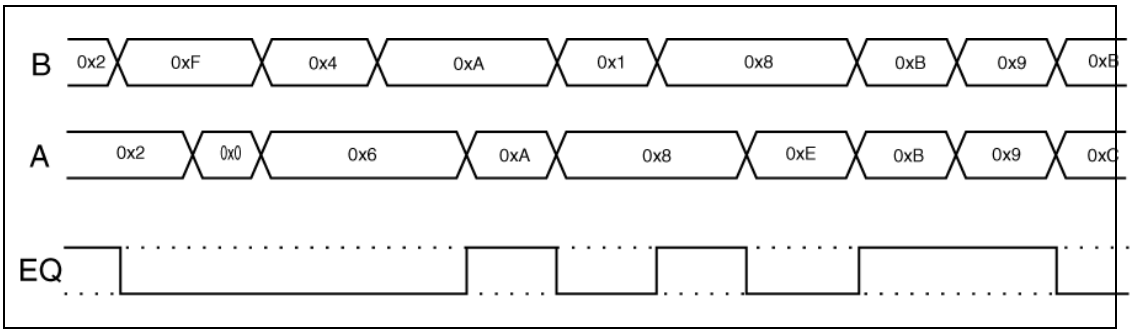


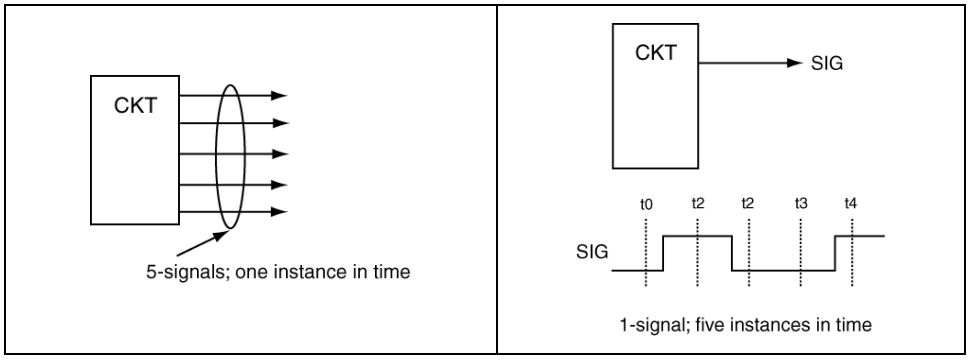
Figure 11.12: The solution to Example 11-6.

11.5 Parity Generators and Parity Checkers

Parity generators and parity checkers are two standard digital circuits that everyone who is anyone in digital-land knows about. They are also a standard digital circuit that every digital designer should understand and be able to design. The concept of parity in used quite often in digital communications, which is why both the concept itself and the circuitry that handles parity are so important. The concept of parity is relatively simple; the application of parity is slightly more complicated but is also on the doable side.

The concept of parity is generally applied to a set of bits. This set of bits can either exist at one moment in time in a *parallel* configuration or the bits can exists over several set times in a *serial* configuration. Figure 11.13(a) and Figure 11.13(b) shows an example of both parallel and serial configurations, respectively. In Figure 11.13(a), the values of the bits in question exist at one instance in time. In other words, the set of bits considered for the application of the parity concept are values of the five signals in Figure 11.13(a) circuit at the same instance in time.

The concept of parity can also be applied to a single signal over a given time span. The parity concept applies to the set of bits in Figure 11.13(b) that are the values of the SIG signal at five different instances in time. The important thing to note here is that the concept of parity applies to a set of bits: this set of bits can either be bits collected in a parallel or serial format as shown in Figure 11.13.



(a) (b)

Figure 11.13: An example of parallel signals and serial signals.

The concept of parallel and serial is probably more extensive than the concept of parity itself. Once the bits in question are gathered, parity refers to the result of a *modulo-2* addition of the bits. Although modulo-2 addition sounds intimidating, the concept is straightforward. Modulo-2 addition refers to a bit-oriented addition operation: the result of this addition is either '0' or '1'. To perform a modulo-2 addition on a set of bits, you add all the bits and your result is either '0' or '1'. In other words, if the sum of the set of bits is '0', the result of the addition was even. If the sum of bits was odd, the result was odd.

As you will find out, the XOR gate inherently performs modulo-2 addition on its two inputs; re-examine a look at the XOR gate's truth table to convince yourself of this fact. The concept of parity simply refers to the notion of whether the sum of the set of bits was odd (odd parity) or even (even parity)[129]. That's about it for the concept of parity; just keep in mind that the concept of odd and even parity has nothing to do with odd and even numbers in the event that you're considering the set of bits in question to represent some type of number.

The concept of parity is particularly useful in digital communications; Figure 11.14 shows a simple example of a communication system that uses the concept of parity. This example shows four bits that are transferred in parallel across some type of medium. The medium in question is immaterial for this problem (but could be anything). What is important in this problem is the notion that a total of four data bits are being transferred: three data bits and a parity bit. For this example, the Generator box generates the data being transferred, which again is not important for this problem.

The Parity Generator box is a circuit that imposes either an odd or an even parity to the three data lines. This parity bit is then included with the data bits being sent in the communication channel. In other words, the Parity Generator circuit assigns its output (the parity bit) to make sure that the set of data and parity bits (A, B, C, & D) are either odd or even parity, depending on how you design the circuit. Once these bits transfer across the medium, the parity better be the same as it was before the bits were transferred. If the bits are sent with even parity and arrive with odd parity, there is obviously an error generated somewhere during transmission. If the bits were originally sent with odd parity and arrives with odd parity, there is a good chance that there was not an error during transmission[130].

This circuit provides error detection for the data bits sent across the medium; in particular, this circuit provides 1-bit error detection capabilities. The circuitry on the receiving end expects either odd or even parity (as set by the circuitry); if it receives a message with the wrong parity, it indicates an error on the PR output of the Parity Checker[131].

[129] In this context, zero, or no bits that are '1', is considered even (even parity).

[130] As you would probably guess, if two bits change, the parity would still be correct but two of the bits would be incorrect and thus your entire message was garbage. The probability that two bits are erroneous is significantly less that the probability that one bit was in error which is why parity is an effect error detecting measure.

[131] Implicit in the description is the fact that the parity generator and parity checker must agree on either odd or even parity before this "system" is set up.

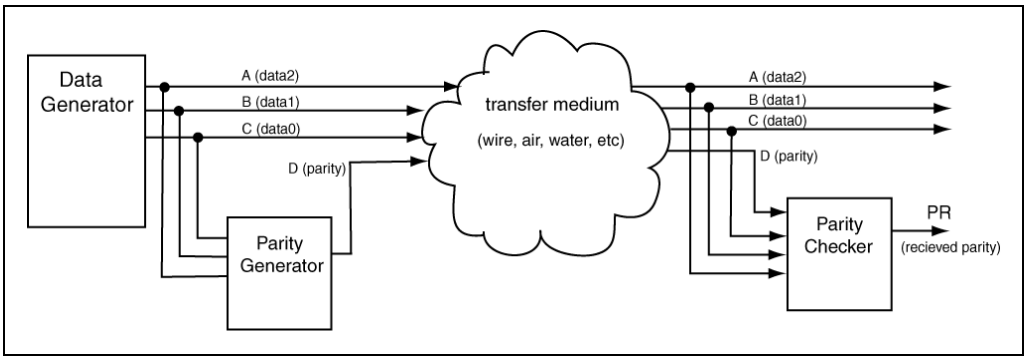


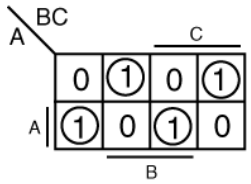
Figure 11.14: An example of parity generation and checking.

The circuitry for parity generators and parity checkers is not overly complicated; you've actually worked with similar circuitry in the case of the Full Adder. For this example, let's design the Parity Generator such that it generates even parity based on the data bits A, B, and C. For ease of representation in this circuit, we'll represent the parity bit with the variable D. What we need to do in this problem is assign D to ensure that the set of bits A, B, C, and D have even parity. The approach we'll take is the BFD approach and examine bits A, B, and C; if these bits have odd parity, the parity bit is set to '1'. In this way, if the modulo-2 sum of the data bits (A,B,C) is '1', then the parity bit will be set to '1' which makes the parity of all four bits '0' (even parity). In other words, parity from bits (A, B, C = '1') + '1' (from the parity bit D) is '0'. With a final modulo-2 addition of '0' for all the bits, the parity of bits A, B, C, and D is even.

The truth table in Figure 11.15(a) shows this concept in tabular form. In other words, the D column is assigned to ensure that modulo-2 sum of all the columns is '0' thus provided the set of bits A, B, C, and D with even parity. Figure 11.15(b) shows the K-map associated with the truth table shown in Figure 11.15(a). At first glance the truth table of Figure 11.15(a) does not appear to have any reduction possibilities, but upon further inspection you'll notice that the K-map contains bunches of diagonals. Thus, we can factor the equation generated from Figure 11.15(b) in order to extract the XOR values as shown in Table 11.1. Figure 11.16 shows the final circuit.

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Figure 11.15: The truth table for a 3-bit even parity generator (a) and the associated K-map (b).

(a)	$D = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$
(b)	$D = \overline{A}(\overline{B}C + B\overline{C}) + A(\overline{B}\overline{C} + BC)$
(c)	$D = \overline{A}(B \oplus C) + A\overline{(B \oplus C)}$
(d)	$D = A \oplus (B \oplus C)$

Table 11.1: Derivation of the even parity generating circuit.

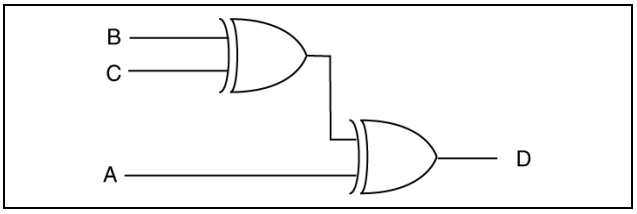


Figure 11.16: The final circuit for a 3-bit even parity generator.

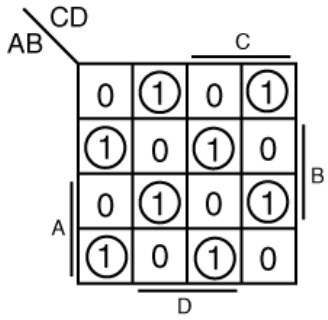
To summarize the previous process, we generated a parity bit (D) based on the three data bits (A, B, and C). For this problem, we arbitrarily produced a bit that generated even parity for the entire set of four bits. All four bits were sent across the medium.

On the receiving side of the circuit, we need to design a circuit that checks the incoming bits to ensure that they are even parity as was sent by the sending end of the circuit. This circuit essentially needs to generate the modulo-2 sum of the four received bits, which is nicely done with a truth table, of all things (more BFD). Figure 11.17(a) shows the resulting truth table. In this truth table, the PR column indicates an error if the parity of the received bits is odd. Since the bits were originally sent with even parity, the arrival of bits having an odd parity indicates that an error occurred in transmission (at least one of the four sent bits was toggled).

Another way of looking at the PR column in Figure 11.17(a) is that the PR column is assigned to generate an even parity based on all of the sent bits. Figure 11.17(b) shows the resulting K-map. Note that this truth table contains characteristics similar to the truth table of Figure 11.15(b). If you were to grind out the equations for this truth table, Figure 11.18(a) lists the final equation; you should actually grind out these equations to increase your competence level in Boolean algebra. Figure 11.18(b) shows the resulting circuit.

As a final note in this saga of parity generation and checking, you should notice a similarity between the final equation of Table 11.1 and equations in Figure 11.18(a). Note that the only difference between a 3-bit even parity generator and a 4-bit even parity generator is the addition of one more XOR term. From this similarity, you can apply the iterative-modular design (IMD) technique to easily create parity generators of more than four bits. This design technique is one of the standard design approaches in digital design-land. In addition, what about odd parity generation? It's a great exercise to examine a circuit that generates odd parity. The approach is similar to the approach take in the even parity generation. The results are definitely interesting but not overly surprising. Consider trying it.

A	B	C	D	PR
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

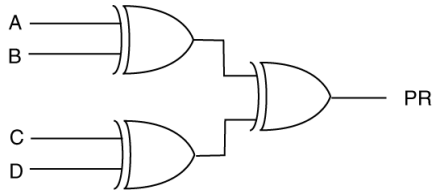


(a) (b)

Figure 11.17: The truth table (a) and K-map (b) for the 4-bit even parity generator.

$$PR = (A \oplus B) \oplus (C \oplus D)$$

$$PR = A \oplus B \oplus C \oplus D$$



(a) (b)

Figure 11.18: The equations (a) and circuit (b) for the 4-bit even parity generator.

Example 11-7

Design a circuit that generates a parity bit that indicates when four bits are even parity.

Solution: This problem is describing an even parity generator; there are two ways to look at the parity bit it generates. One way to look at the parity bit is that it indicates with a '1' when the four input bits are odd parity. Another way to look at it is that the parity bit is assigned such that the five bits (the four input bits and the parity bit) always exhibit even parity.

As always, a great place to start is by drawing a black box. Figure 11.19 shows the black box diagram associated with this problem. Note that there are four input bits; the output labeled "PR" is the parity bit.

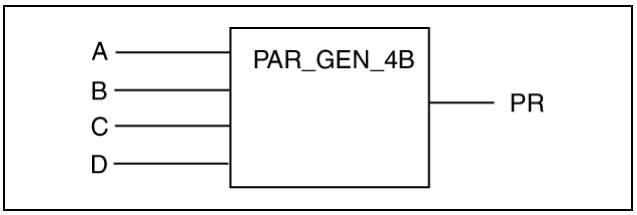
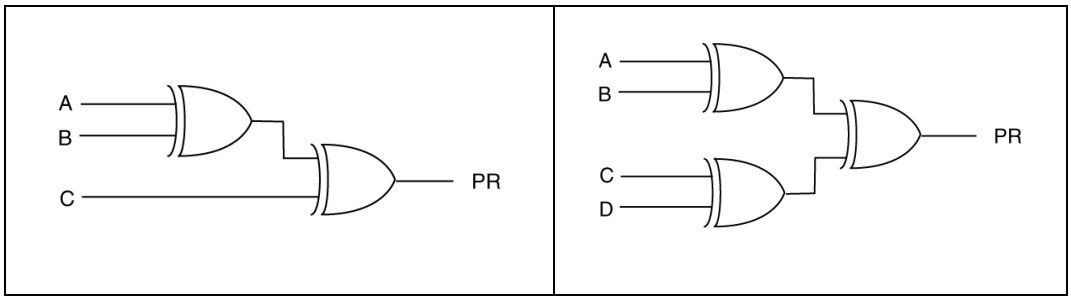


Figure 11.19: The block diagram for Example 11-7.

We could use the BFD approach to solving this problem, but we would rather use the IMD approach to save us time. Recall when we first described parity, we designed a 3-bit even parity generator. Figure 11.20(a) shows the final solution to that problem once again. In order to extend this circuit to be a 4-bit even parity generator, we add another XOR gate as shown in Figure 11.20(b). For this problem, the communications channel would now be sending five signals: A, B, C, D, & PR. The receiving ends of the channel then examines these five signals in order to verify that the received signal exhibited even parity.

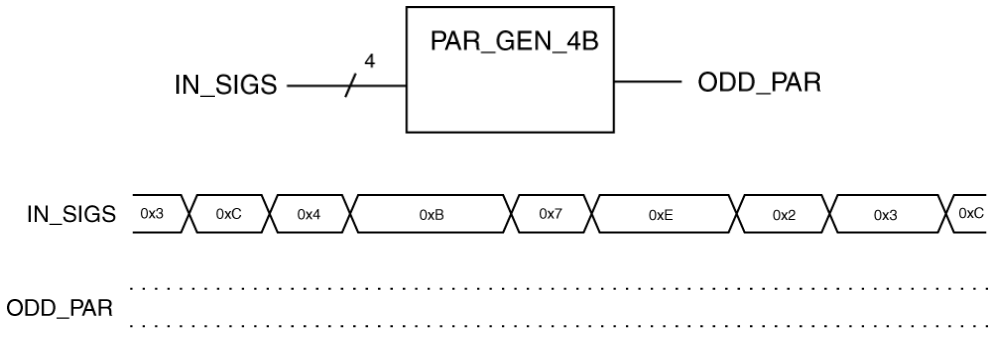


(a) (b)

Figure 11.20: The circuit solution for a 3-bit even parity generator (a) and the solution to Example 11-7, a 4-bit even parity generator.

Example 11-8

Use the black box diagram to complete the accompanying timing diagram. Consider the black box to generate odd parity based on the four input bits.



Solution: For this problem, the ODD_PAR signal generates a '1' when the sum of "1's" on the IN_SIG signal is even; otherwise, ODD_PAR generates a '0'. Figure 11.21 shows the final timing diagram.

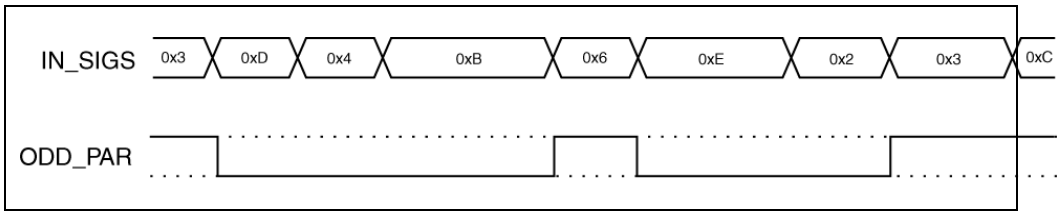


Figure 11.21: The solution to Example 11-8.

As a final note, you're correct in thinking that the idea of parity generation and parity checking is somewhat confusing. The basic concepts are straightforward; the problem is with the associated vernacular. Here is a basic overview of the confusing vernacular.

- If you're generating odd parity, your parity generator uses a '1' to indicate when the input bits have even parity. Including the '1' makes the odd parity of the signals into even parity.
- If you're generating even parity, your parity generator uses a '1' to indicate when the input bits have odd parity. Including the '1' make the odd parity of the signals into even parity.

Example 11-9

Design a circuit that adds two 4-bit digital values. If the addition operation generates a carry-out, the 4-bit sum output will be all zeros; otherwise, the 4-bit output will indicate the sum of the two 4-bit input values.

Solution: The first part of this problem is straightforward; we simply need to use a 4-bit RCA. The issue is what to do with the other part of the circuit. The best place to start from here is with a black box model as is shown in Figure 8.10.

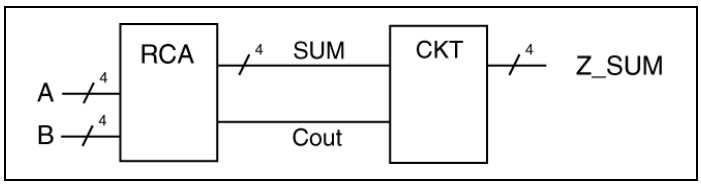


Figure 11.22: Black box diagram for this problem.

This is a typical design problem in many ways. The approach you need to take here is to think about the requirements of the black box labeled "CKT". What we need to do is pass the SUM output along if there is no carry or make all the SUM bits a logical '0' if there is a carry. What this operation describes is to pass the SUM signals along if the carry-out is '0', or, clear all of the sum bits. This operation describes a classic switch action by the carry-out. We have a gate that implements such an operation: it's called an AND gate.

Figure 11.23 shows the final solution for this problem using AND gates. Note that we needed to first invert the carry-out signal in order for it to have the correct affect on the associated AND gates. The method used to connect the AND gates insures that their output is '0' when the carry-out signal is a '0' as it is inverted before being input to the AND gates. Note that we truly did an IMD approach with the addition of the AND gates. Also, note that we indicated the expansion of the SUM bundle by using parenthetical notation on the signal contained in the bus. The numbers in the parentheses are typical of bundle expansion.

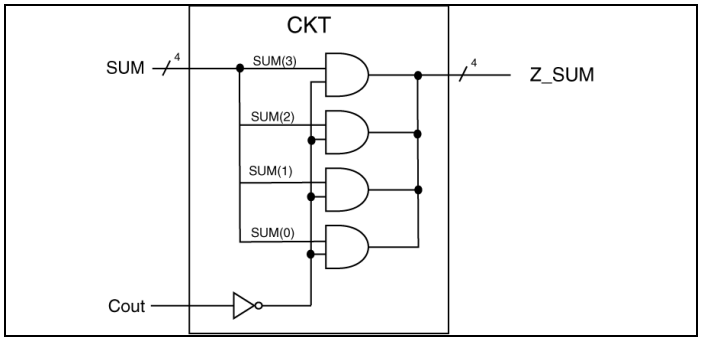


Figure 11.23: Schematic diagram for the box labeled CKT.

Chapter Summary

- Iterative Modular Design (IMD) is a more powerful design method than brute force design (BFD). What makes IMD more powerful is that it bypasses the constraints presented by the truth tables and the entire BFD approach. Special type of digital circuits are typically designed with IMD.

- Important Standard Digital Modules presented in this chapter:

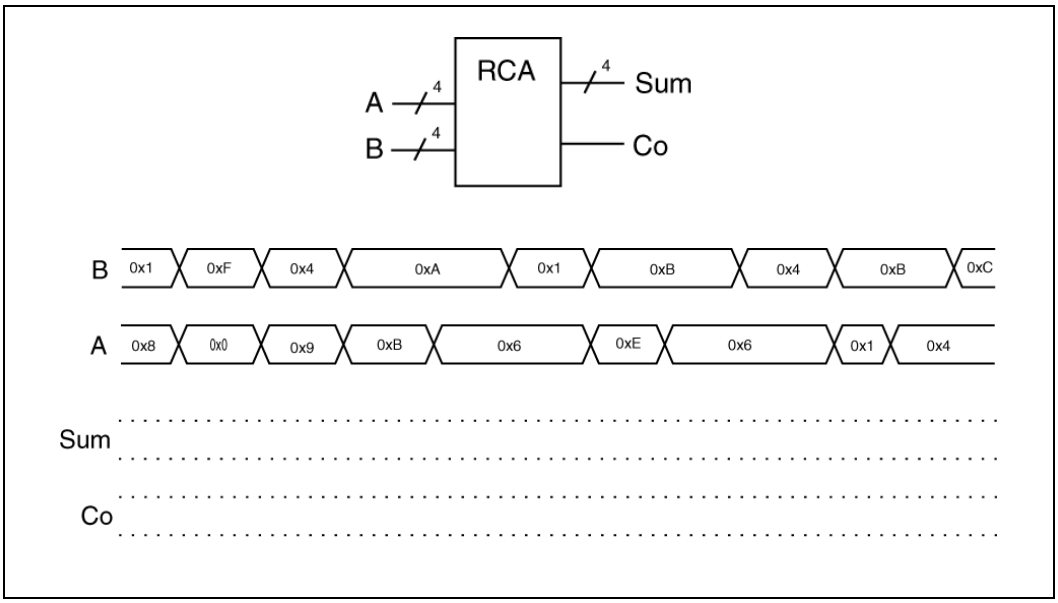
    - Ripple Carry Adder (RCA): arithmetic circuit used add digital values
    - Comparator: arithmetic circuit used to determine equality of two digital signals
    - Parity Generator: circuit used to generate parity based on a set of digital signals
    - Parity Checker: circuit used to determine the parity of a given set of digital signals.

- The notion of parity describes a characteristic of a set of signal or a sequence of signals. Parity is defined as the modulo-2 addition of the '1' bits of the signals in question. Parity can be either even or odd. Parity generators are used to generate a parity bit that ensures a group of signals exhibit even parity or odd parity. Parity checkers are essentially the same circuit as parity generators: both are implemented on the gate-level using a set of exclusive-OR type gates.

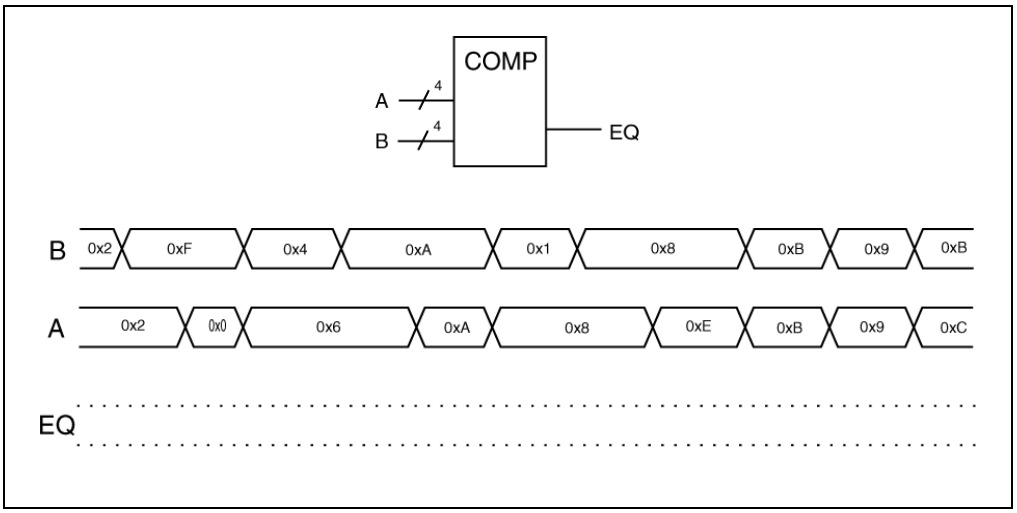
Chapter Exercises

1) Design a circuit that performs as follows: The circuit has six 10-bit unsigned binary inputs (A,B,C,D,E,F). Comparisons are made between (A,B), (C,D), and (E,F) pairs. If two and only two of these number pairs are equal, then the circuit's one output is '1'; otherwise the circuit's output is a '0'. Use only standard digital modules in your design. Minimize your use of hardware in this design. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

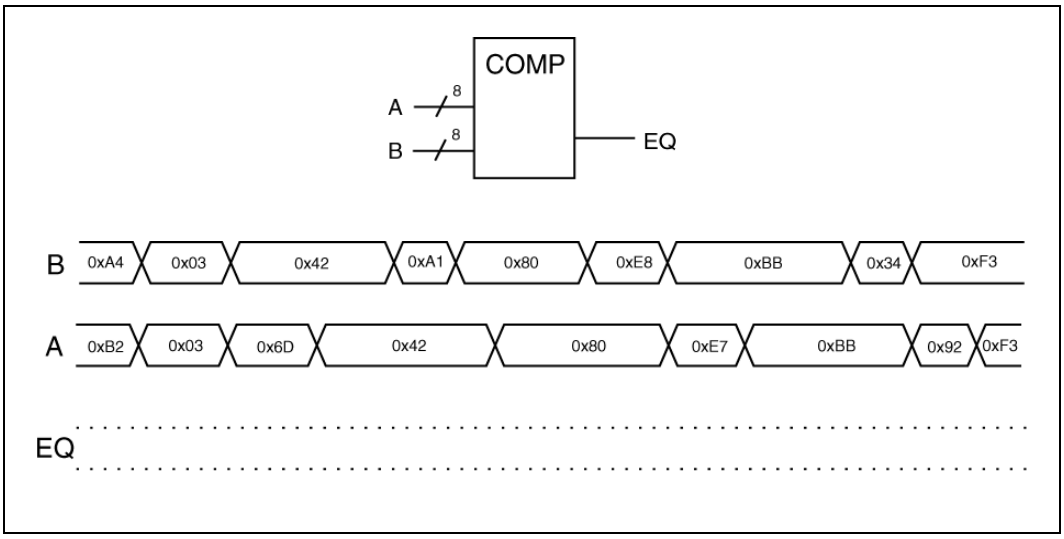
2) Complete the timing diagram shown below considering the given schematic symbol.



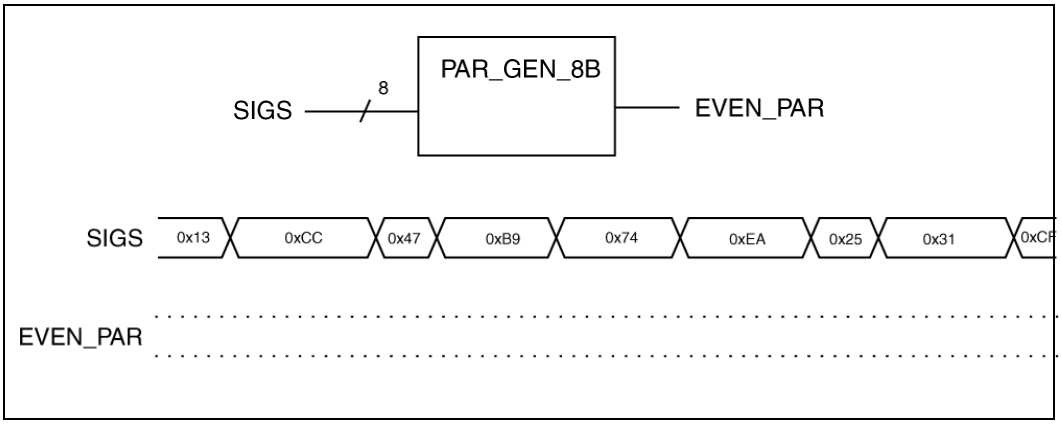
3) Complete the timing diagram shown below considering the given schematic symbol.



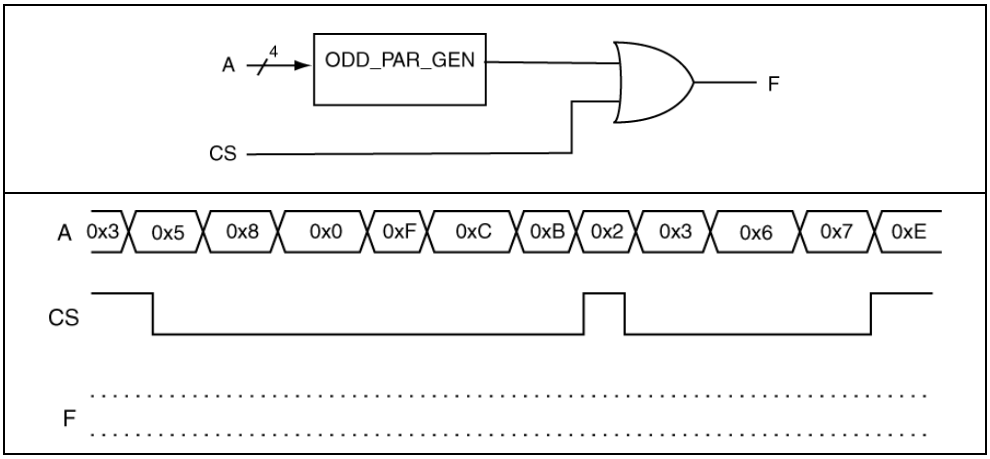
4) Complete the timing diagram shown below considering the given schematic symbol.



5) Complete the timing diagram shown below considering the given schematic symbol. Consider the circuit to generate even parity for the eight input bits.



6) Use the following circuit to complete the listed timing diagram.



Design Problems

1) Design a special 4-bit RCA with the following specifications. This circuit has an input named INV_OUT; when this input is in the '1' state, the output of the RCA is inverted from what it would normally be. HINT: Since you know all about RCAs, your solution to this example should include a dark box labeled RCA (no need to reinvent the wheel on this problem).

2) Design a 16-bit RCA using two-8-bit RCAs. For this problem, assume an 8-bit RCA was previously designed so you won't need to re-design it.

3) Design an 8-bit comparator using only standard logic gates. The output of this comparator has only one output that indicates whether the two input values are equal or not.

4) Design a 2-bit comparator in that compares two inputs, A & B; the output should indicate when A = B, A < B, and A > B. You'll need to use the IMD approach with this design.

5) Regarding the previous problem, would it be possible to use that design to generate a 6-bit comparator using the IMD approach? Explain briefly but completely.

6) Design a 3-bit odd parity generator. Specifically, this circuit indicates when the three input bits are even. Assume the 3-bit inputs are in a parallel configuration.

7) Using the design from the previous problem, design an 8-bit odd parity generator using IMD. Assume the 8-bit input is in a parallel configuration.

12 Chapter Twelve

(bryan mealy 2012 ©)

12.1 Introduction

Despite the fact that the introduction of programmable logic devices has massively changed the many aspects of digital design, it is not a topic that we'll go into in any detail. Our approach here is to use PLDs to our advantage while skipping over the low-level implementation details. In other words, we'll be dealing with PLDs at a user-level, or at a high-level of abstraction.

There was a time when this material was a much more important part of an introductory digital design course[1], but that was a long time ago. In other words, this is interesting stuff but you may never see it again despite the fact that you may be well on your way to becoming the world's greatest digital designer[2]. The chapter's topics do present a nice foundation of digital knowledge and history that you'll find useful now and then (for whatever that is worth). We make no claim for this chapter to be overly useful, as a lot of stuff is missing.

Main Chapter Topics

- **PROGRAMMABLE LOGIC DEVICES (PLDS)**: PLDs are a key element in modern digital design. This chapter presents a historical overview of PLDs and outlines some of the basic operating features.

Why This Chapter is Important

This chapter is important because it provides an overview of programmable logic devices, including a brief history and brief architectural overview. This chapter provides a context for PLDs and their use in modern digital design.

12.2 A Brief History of Digital Design.

In case you have not noticed yet, the approach taken by digital logic design these days is to abstract things to an impressively high level. While there are a lot of good things to be said about this approach, (such as improved productivity), there is one really bad thing that you should be aware of: so many of the low level details are hidden from you that you're at risk of losing touch with your basic foundation

[1] Back in the days when paper designs ruled the world and teachers had nothing better to test students on.

[2] Actually, if you are on your way to being a great digital designer, you'll probably need to delve into the details of the particular PLD you may be using. These devices do a lot; you'll eventually need to take advantage of some of their advanced features in your design. Yes, this means you may have to actually read the associated datasheet.

of digital logic. This was not the case all those years ago when computers were expensive and useful software was essentially non-existent. This chapter attempts to remind you of both how you've gotten to the digital place you find yourself today and to give you a greater appreciation of some of the tools and hardware you use on a daily basis.

If you stop and think about it for a moment, it's amazing how far digital integrated electronics have come is such a short time[3]. What is even more amazing is that as things become more developed, the speed at which changes happen seems to increase. This is especially true as general purpose computers become involved in the game with various forms of design automation and computer aided design (CAD) tools. The discussion that follows is centered around the path that got us to where we are today in the context of hardware design and programmable logic devices (PLDs).

12.2.1 Digital Design: Somewhere in the 1980's

In the 1980s, digital circuits were generally centered around discrete logic that was contained on integrated circuits (ICs). These ICs were generally referred to as SSI and MSI: small scale integration and medium scale integration. The difference between small, medium, and large scale, and very large scale integration is determined by the number of transistors in the IC)[4]. All the designs in the digital courses were primarily paper designs and there was little point in implementing them in actual hardware. A few circuits were actually implemented but students learned quickly that implementing circuits using SSI and MSI was extremely time consuming.

The SSI and MSI chips back then did not have too much functionality so even a simple digital design ended up requiring many ICs. These designs were generally done on protoboards[5], wire-wrap boards[6], or speed-wrapped boards[7] because fabricating your own printed circuit board (PCB) was too expensive, too messy, or too impossible[8]. Table 12.1 lists some of the good points and bad points of associated with this level of digital design.

Good Points:	Bad Points:
1. You could go to the local electronics store, buy the required ICs, and implement designs at a relatively low cost. 2. You could get most of the designs working using a minimal amount of test equipment (about the only thing affordable in those days was a multimeter).	1. Circuits were tedious to assemble (proto-board, PC board), hard to debug, and not easily modified. 2. Multiple ICs required lots of board space and consumed a lot of power. 3. Debugging options were limited and required special equipment.

Table 12.1: Some of the good and bad points of early digital design.

[3] Worthy of note, one of the cool things about digital design is that the advancements have been so amazing. There is nothing even remotely comparable on the analog side of electronics no matter what any analog person tries to tell you.
[4] An example of SSI would be an IC with a few NAND gates on-board. An example of MSI would be an IC that contained a counter or a multiplexor (digital devices you'll learn about later).
[5] White boards with many holes in which to stick in wires and components.
[6] Components had long posts that were used to connect to other components via thin wire.
[7] A completely flaky technology that was simpler than wire-wrapping but died off anyway.
[8] There were no PCB companies that did the work for you at a reasonable price.

12.2.2 Digital Design: The Early 1990's

Back around this time, programmable logic devices (PLDs) started appearing[9]. These were relatively simple devices such as the PALs (programmable array logic) and PLAs (programmable logic arrays). These early devices were not overly large or complex and were very understandable when presented in digital logic courses. Modern PLDs have become quite large and complicated. Table 12.2 lists some of the good points and bad points of associated with this level of digital design.

Good Points:	Bad Points:
1. Reduced the number of ICs required in a design (lower power, more board space, faster circuits). 2. More flexible: circuits could be changed quickly without major revamping (the devices were re-programmable). 3. Electronic Design Automation (EDA) tools appear and prosper.	1. Required special hardware and software to program the PLDs. 2. Required learning a "programming language" (namely a "hardware description language", or HDL). 3. Not easily tested due to the fact that so much circuitry was onboard the PLD.

Table 12.2: The good and bad points of early PLD design.

12.3 PLD Architectural Overview

The early PLDs could essentially be modeled as an array of AND logic (the AND array) connected to an array of OR logic (the OR array). The programmability of these devices is centered on the presence of connections (or lack there of) made in the circuitry. The terminology typically used is that if two lines were connected with a "fuse", the connection was made (a physical contact was made). If the connection did not have a fuse, then the connection was not made (an open-circuit was created).

Figure 12.1 shows three interesting connections commonly used in CPLD circuit descriptions. Figure 12.1(a) shows a connection between two signals that was made at the factory. Generally speaking, this connection was made at the "mask-level" it cannot be changed. Figure 12.1(b) shows a connection made with a fuse (the "X" is the symbology used to indicate that a connection exists between the two signals). The model of a fuse is that it can be broken, and thus the connection between the two signals will no longer exist. The short story is that the programmable portion of the PLD acronym is based on blowing or not blowing these fuses[10].

[9] Actually, they had been around for a while but they were now becoming affordable to real humans and student-types.

[10] In reality, the actual silicon does not contain a "fuse". Actual semiconductors use various forms of technology to obtain the functionality of a fuse. Have you ever seen a fuse box attached to an IC?

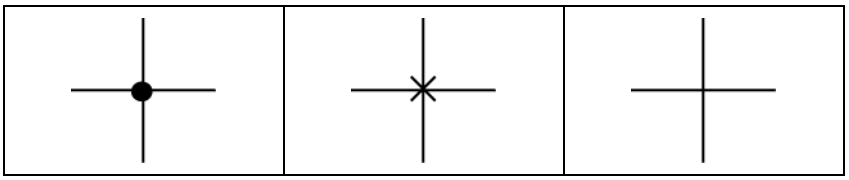


Figure 12.1: A factory made connection (a), a fuse-based connection (b), and no connection (c).

Figure 12.2 shows some other standard terminology used in PLD-land. As you'll see in some of the upcoming diagrams, the number of gate inputs can be excessive. For each of the gate inputs, there will need to be an input signal line; the diagrams can ugly real fast due to the shear number of logic devices contain on the devices. To combat this ugliness, the shortcut symbology of Figure 12.2: is used. This symbology uses one input to model all of the gate inputs; the resulting circuit diagrams are much cleaner looking.

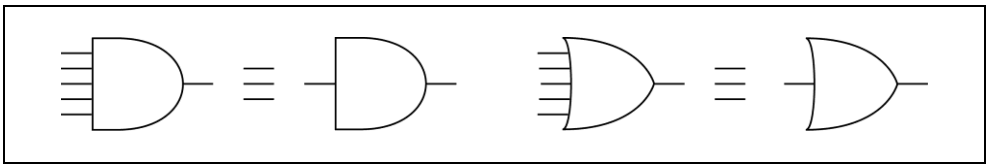


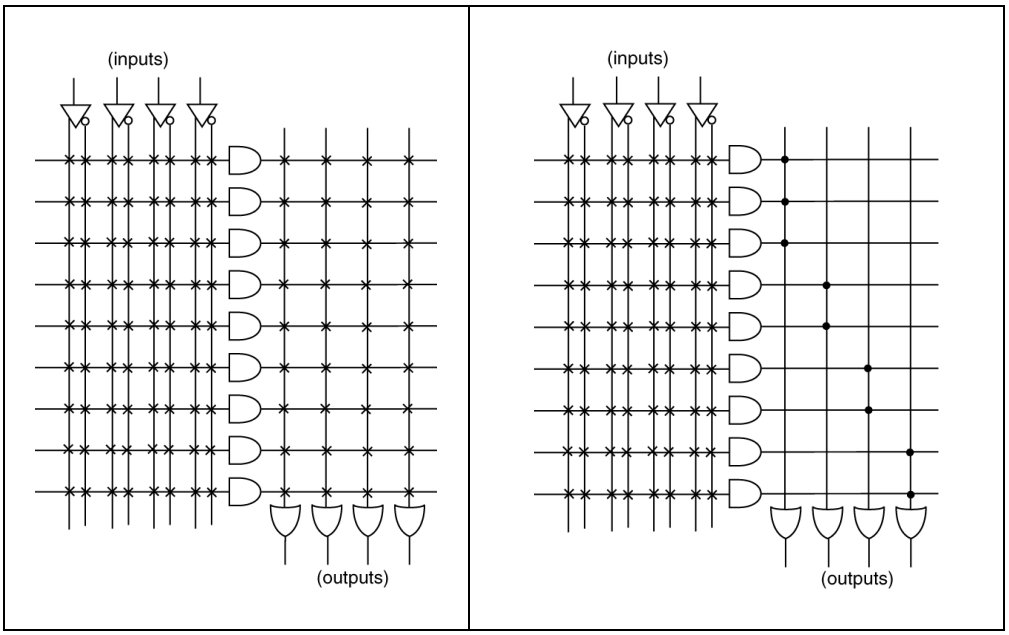
Figure 12.2: The shortcut notation generally used in describing PLDs.

Several types of PLDs can be modeled as an AND plane that connects to an OR plane. Figure 12.3(a) shows a model of such a circuit. This can most accurately be considered a model of a *programmable logic array* because both the AND plane and OR planes are programmable. In other words, programming the device is a matter of removing the unwanted fuses.

Figure 12.3(b) shows another similar device, the *programmable array logic*, or PAL. This device is noticeably similar to the PLA but the connections in the OR plane are programmed in the factory, or masked programmed, and thus cannot be reprogrammed by you the user. The important thing to note here is that both of these devices have AND planes and OR planes. As you already know, we can use AND/OR circuit forms to implement function (recall the AND/OR function forms). What these PLDs provide is a single device that can implement many different functions simultaneously which subsequently reduces the need for discrete logic ICs on a given PCB.

The reality is that these simple models for PLDs are no longer overly accurate. Yeah, some modern PLDs contain AND and OR planes, but many have other styles of internal architecture[11]. Moreover, the models shown in Figure 12.3: are very simple models. Even the simplest of AND/OR-based PLDs contain more logic than is listed in these diagrams. If this raises your curiosity, I suggest you do some product searches.

[11] In this context, architecture refers to the internal structure of the device.



(a) (b)

Figure 12.3: A PLD with all the fuses in place (all connections made) (a), and a PLD containing an OR plane that has been factory-programmed (b).

12.4 Simple PLD Function Implementations

Let's take a look at one example PLD and analyze the functions it implements. Figure 12.4 shows an example PLD. Although the functions it is implementing have no real meaning, we can still analyze it to verify we have the PLD analysis technique down. The X's in the AND plane of Figure 12.4 indicate connections are made and thus form the literal inputs to the individual AND gates. The outputs of the AND gates in the AND plane form the product terms used in the final function. Once again, the X's represent the fuses that have not been removed by programming the device. The dots in the OR plane represent connections to the OR gates below. The column of AND gates implements a bunch of product terms. The row of OR gates implement SOP functions as they sum the product terms from the AND array. Figure 12.5 lists the four functions that the PLD in Figure 12.4 implements. Don't get overly excited about these equations; they certainly do not represent anything intelligent.

Not listed in Figure 12.4 or Figure 12.5: are the output options that are typically included with PLDs. In an effort to make the devices more functional, the outputs are generally available to be fed back to serve as inputs to other functions. This somewhat more complex type of circuitry allows the devices to implement a greater number of functions as well as functions that are more complex. Not all PLDs architectures share this functionality; I mention it here just in case it may matter. The truth is that these types of PLDs can have complex architectures, which are way beyond this discussion.

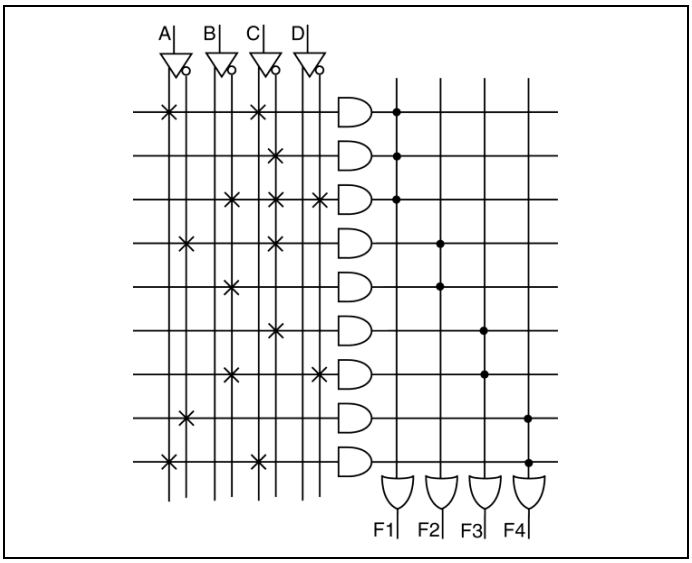


Figure 12.4: Typical model of a PAL.

$$F1 = AC + \overline{C} + \overline{B}\overline{C}\overline{D} \qquad F2 = A\overline{C} + \overline{B} \qquad F3 = \overline{C} + \overline{B}\overline{D} \qquad F4 = AC + \overline{A}$$

Figure 12.5: The Boolean functions implemented by the PAL of Figure 12.4.

12.5 Other Types of PLDs

Another major type of PLD is referred to as a *complex programmable logic devicee*, or CPLD, and the *field programmable logic device*, or FPGA. These two types of devices also have completely different architectures than the PAL shown in Figure 12.4. The CPLD does typically contain structures that can be categorized as AND/OR arrays while the FPGA generally does not. You'll study these devices in further detail in other courses; there is not too much worth saying about them now.

The idea behind the CPLD is to put a bunch of PLDs on a chip and give them the ability to talk with one another. The PLD-like part (AND/OR array) is referred to as a *logic block* and are connected to one another through programmable interconnects. It is the programmable interconnects that give the CPLD flexibility and efficiency. Figure 12.6 shows a generalized architecture of a CPLD; note that each of the logic blocks shown in Figure 12.6 can be considered a single PLD. The I/O blocks shown in Figure 12.6 are also configurable (opposed to programmable) to include pull-up resistors and other fun-type things that are generally associated with digital outputs and serve to extend the flexibility of the device.

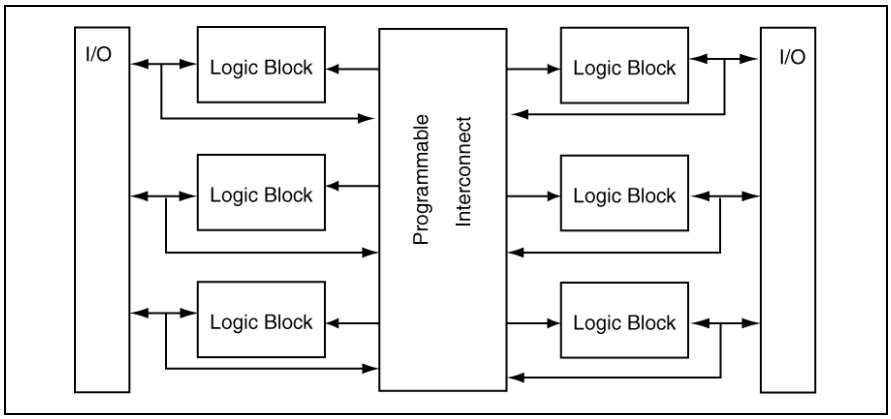


Figure 12.6: General diagram of CPLD.

The notion of configurability is different from programmability in that it allows you to select from a list of pre-defined options as opposed to creating your own options, as is the case with programmability[12]. Generally speaking, programming refers to the ability to configure various logic on the device while configurability refers to selecting various options in or associated with the device[13].

Figure 12.7 shows an expanded view of the logic blocks. As you can see, a logic block is comprised of an AND/OR array which is connected to a *macrocell*. Macrocells are an important part of CPLDs they have significant functionality and are fully programmable which makes modern CPLDs massively powerful. The macrocells provide polarity control and output level adjustment among other things. Often times, the measure of a CPLD's capacity[14] is provided in terms of macrocells.

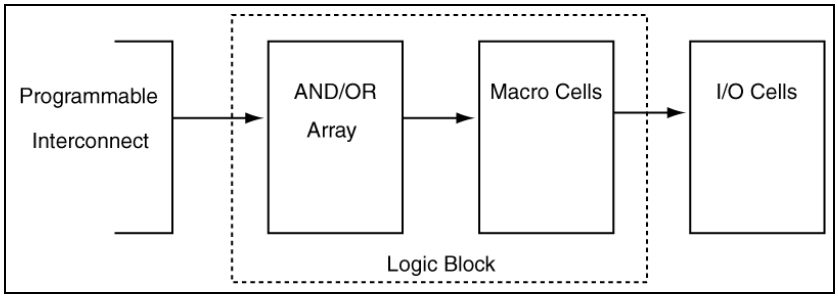


Figure 12.7: Logic Block and its relation to PIs and I/O.

In the end, designing with PLDs allowed the designer to shift focus from making sure all the ICs on your board were talking to each other to producing a circuit that satisfied the design description.

[12] A good analogy is going to a restaurant vs. going to a buffet. You get to roll your own in the buffet and create whatever you want (programmability of the logic) while in a normal restaurant, you pick between a set number of selections. What a stupid analogy.

[13] Generally speaking, you can configure the device by selecting options in the accompanying software used to select internal logic and program the device.

[14] Generally, capacity refers to how much logic is contained on a PLD. Due to several factors, not all of the logic can be used in many cases.

12.6 Final Comments on PLDs

Once again, this has been a brief introduction to PLDs and many details are missing. In truth, modern PLDs are amazingly powerful and complex devices. The internal architectures of these devices make them useful in many applications. Keep in mind that some PLD (namely FPGAs) also contain devices such as complete computers as part of their architectures. The complexity of and versatility of these devices has opened up at least two new markets for them. The point here is that PLDs have come a long way since only being able to implement a few functions; now they are able to implement large and complete digital systems.

> **Rapid Prototyping**: FPGAs are typically used quickly develop prototype systems before the "real" system is in place. This means you can test the feasibility of a design while you're waiting for a chunk of silicon. This notion also supports a fast time-to-market if you're planning on having a design using FPGAs on the final product.
>
> **Device Verification**: Because most ICs these days are first modeled in hardware description languages, the same models can first be used to configure FPGAs in order to test the models in real hardware. The idea here is to do everything humanly possible to verify your design is going to function properly before you send the device off to the fab[15]. Keep in mind, it costs a million bucks or so to have a device fabbed; in addition, even a rush fab order requires six weeks or so to complete.

The size of PLDs is always a confusing issue. While there are many ways to state the size of a PLD, not all of these ways are 100% honest. First, the act of transferring your design from a model (such as VHDL) to the actual device can be extremely complex. The development software must interpret your model in such a way as to implement its functionality on a PLD. The problem is that there are many ways to interpret models and there are many ways to place that model onto an actual PLD.

Consider this notion: you have a design that you want to go onto an FPGA and you're going to make a 1000 of these units. You want to maximize your profits by spending as little money as possible on the FPGA, so you want to choose the smallest FPGA you can and still have your design fit onto it. The notion of fitting a particular design onto an FPGA is something that many people have dedicated their lives to doing better. In addition, if you're a company that writes the development software, you want it to be able to implement designs in as small of a footprint as possible on the FPGA; the better you can do this, the more people will purchase your devices over a competitor's device. Having crappy software means that you'll have to spend more and buy a larger FPGA. Therefore, size does matter, which means the software determining size matters also[16].

The other confusing issue regarding the capacity of PLDs is the notion of routabilty. In typical PLD architectures, you have chunks of logic that talk to other chunks of logic; this conversation is done via the "routing resources" contained as part of the internal architecture of the PLD. In reality, if your device has X amount of logic resources, it will be impossible to use all of these resources in your design because of resource constraints presented by routing resources. Once again, the better the software can place and route your model onto the fabric of the PLD, the greater amount of logic you'll be able to use on your device.

[15] This is a shorthand term for "fabrication facility", a place where they create ICs from random chunks of silicon.
[16] This should make you ask the following question: In development environments that are provided free of charge, do you really think the free environment is going to do a job creating a small footprint for your design? The answer is now; typically, these companies want you to buy their expensive development software that contains the secret sauce you truly desire.

In case you're overloaded with acronyms, check out Figure 12.8. This figure shows that all of the different programmable devices discussed in this chapter are forms of PLDs. There are actually many other types out there not listed in Figure 12.8 as this is a modern digital logic textbook and not a history of technology textbook. One thing worthy of note here is that, generally speaking, every company out there has its own particular PLD architecture. This being the case, if you become an expert on one company's PLD, you won't necessarily know squat about another company's PLD.

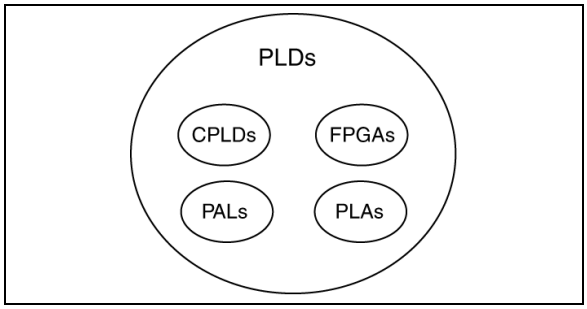


Figure 12.8: A Venn diagram showing the relationship between various PLDs.

Chapter Summary

- Programmable logic devices are a relatively new area in digital design. Although early PLDs were simple devices, modern PLDs are massively functional to the point of being the defining device of digital design. PLDs are world of their own and are massively complex at the gate level. Software tools, however, allow modern digital designers to bypass the low-level details of PLDs and implement circuits on a higher level.

- Common types of PLDs include PALs, PLAs, ROMs, CPLDs, and FPGAs. Check the list of terms for a complete description of these acronyms.

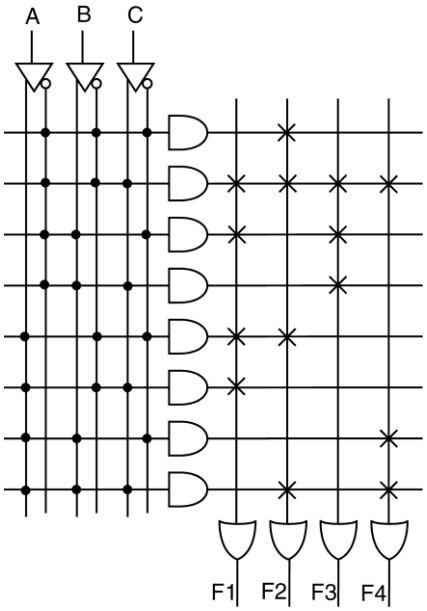
Chapter Exercises

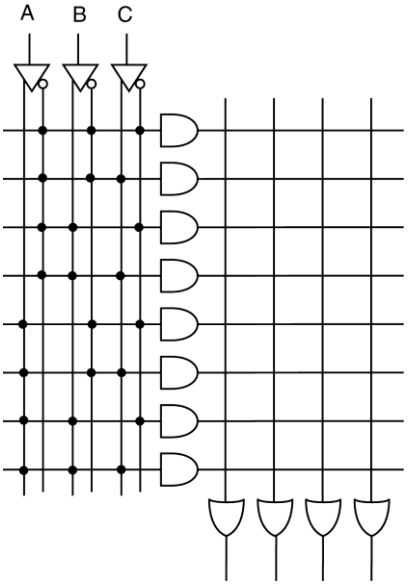
1) Is it possible to utilize all the logic resources on a typical PLD? Why or why not?

2) Would it be possible to implement a RCA on a PLD that contained only a simple AND and OR plane? Briefly discuss the issues involved.

3) List the Boolean expression implemented by the PLD on the left. Implement the following Boolean expression for the PLD on the right:

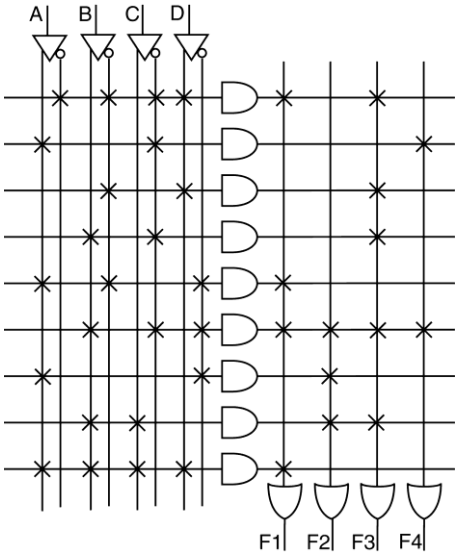
$$F5 = AB\overline{C} + \overline{A}\overline{B}C + ABC$$
$$F6 = \overline{A}B + BC + A\overline{C},$$
$$F7(A,B,C) = \sum(2,3,5,6,7)$$
$$F8(A,B,C,D) = \sum(2,3,4,5,10,11)$$

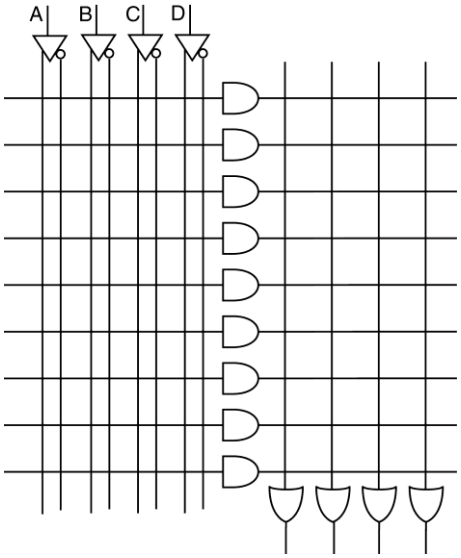




4) List the Boolean expression implemented by the PLD on the left. Implement the following Boolean expression for the PLD on the right:

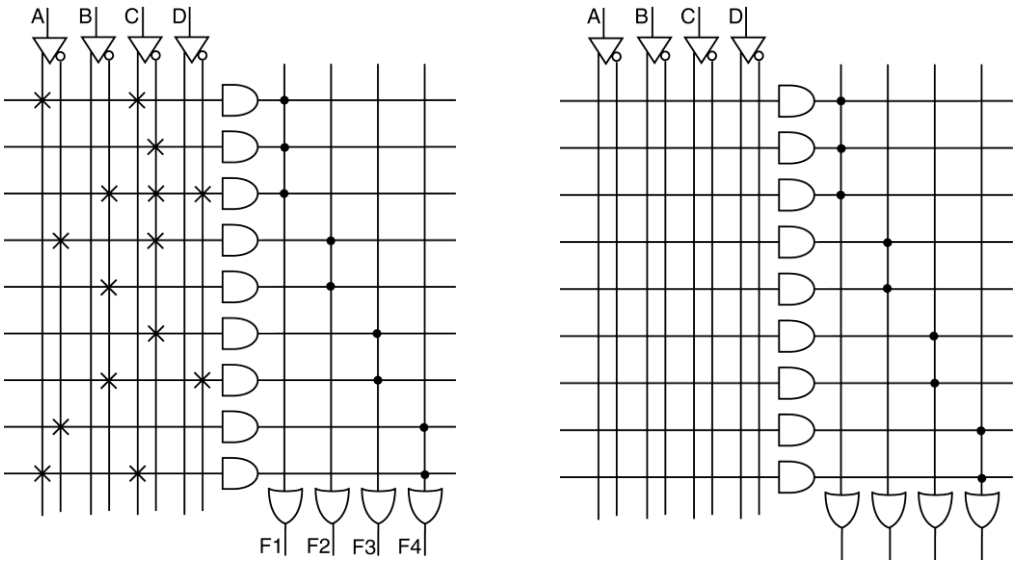
$$F5 = B\overline{D} + \overline{A}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{B}\overline{D} + \overline{A}CD$$
$$F6 = AC\overline{D} + B\overline{D} + \overline{A}C\overline{D} + \overline{A}C + \overline{A}\overline{C}$$
$$F7 = \overline{A}C + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{B}\overline{D} + \overline{A}CD + BC$$
$$F8 = B\overline{D} + \overline{A}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + BC$$





5) List the Boolean expression implemented by the PLD on the left. Implement the following Boolean expression for the PLD on the right:

$$F5 = \overline{A}\overline{B}C\overline{D} + BC$$
$$F6 = \overline{B}\overline{C} + A\overline{C}$$
$$F7 = \overline{A}\overline{C} + ABC\overline{D}$$
$$F8 = \overline{A} + \overline{D}.$$



13 Chapter Thirteen

(Bryan Mealy 2012 ©)

13.1 Chapter Overview

One of the major uses for digital logic design is creating circuits that perform mathematical-type operations. The foundation of digital knowledge that you've built up to now is sufficient to introduce the topic of binary number representations and conversions between representations in the context of arithmetic-type circuit design. The concepts presented in this chapter allow you to move closer to designing some reasonably interesting circuits and useful circuits (but not that close).

Main Chapter Topics

- **NUMBER SYSTEMS:** This chapter reviews previously presented number systems and describes hexadecimal and octal numbers.
- **CONVERSIONS BETWEEN MIXED RADII:** This chapter describes basic algorithms to convert between presented number systems.

Why This Chapter is Important

This chapter is important because a significant portion of digital design deals with numbers and their various representations. Understanding of these representations, including conversions between representations, will help all digital designers

13.2 Number Systems

A previous chapter presented a general overview of number systems. Although there are definitely many number systems out there, you'll only need to directly know two of them in order to master digital design: decimal and binary. The good news is that you probably already know decimal[148]. The less than good news is that you need to become fluent with binary since it's the only number system that digital hardware understands. The bad news about binary is that it is sometimes a pain in the ass to work with. However, the good news about working with binary is that there is some help available. The bad news is that in able to get that help, you have to learn the hexadecimal (radix=16) and octal (radix=8) number systems in order to aid you in your work with binary numbers. It's not all that bad. It's actually sort of fun. But more importantly, the skills you develop working with these number

[148] Unless of course you're aspiring to be some type of academic administrator; in that case, even decimal is challenging to you. Those who can, do; those who can't, well, they become academic administrators.

systems is something you constantly use in digital design, computer science, computer engineering, and bowling.

13.2.1 Hexadecimal Number System

The hexadecimal number system contains sixteen numbers in its associated ordered set of symbols. The first ten numbers are the same as decimal numbers, but the last six are somewhat new. Since there are no more unique decimal numbers to act as the final six numbers in the hexadecimal number system, it is customary to switch to alpha characters. In other words, we use the letters A→F to represent the hex numbers 10-15. Table 13.1 shows the hexadecimal numbers along with the associated decimal and binary numbers (in 4-bit format).

13.2.2 Octal Number System

The octal number system contains eight numbers in its ordered set of symbols. Since there are less than the ten standard numbers in the decimal system, there is no need to use alpha characters as we did for hexadecimal numbers. The thing of interest here is that once we run out of symbols in the octal number system (as we do for numbers greater than seven), we draw on that familiar juxtapositional notation where we start placing the numbers side by side. Table 13.1 shows this characteristic for several important digital radii. Convince yourself that this carry over notation is similar to the decimal number system but the carry over occurs at eight rather than at ten.

The importance of memorizing the first three columns of Table 13.1 cannot be overemphasized. The faster you have this stuff memorized, the better off you'll be in your digital design or computer science (or bowling) career in general. I would suggest you make some flash cards and use them for a few minutes per day. It'll be time well spent; you won't regret it.

(base 10) **Decimal**	(base 2) **Binary**	(base 16) **Hexadecimal**	(base 8) **Octal**
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Table 13.1: Numbers that every digital designer need to memorize really soon.

13.3 Number System Conversions

The reality is that we humans think in decimal but computers and other digital devices operate strictly in binary. This means we'll need to be able to translate between the various number systems typically associated with digital design. What you see in this section is that we humans use hex and octal numbers to simplify the representation of binary number. The use of hexadecimal is purely an aid for humans to handle long strings of 1's and 0's that digital circuitry requires. VHDL readily understands hexadecimal, which is yet another added bonus. While digital design uses hex notation quite often, octal numbers are much less useful.

13.3.1 Any Radix to Decimal Conversions

A previous chapter covered a majority of this topic for this information will partially be review. As a reminder to you, the digit positions in any number using juxtapositional notation have weights associated with them. The associated number multiplies the weights in order to generate the final number. There were few examples in an earlier set of notes for decimal and binary; below are two examples for octal and hex numbers, respectively.

Example 13-1: Octal-to-decimal conversion

Convert 372.31_8 (octal) to decimal

Solution: Example 13-1 is an example of octal-to-decimal conversion; Table 13.2 shows the solution to Example 13-1.

Decimal Value of Digit Weight	64	8	1		0.125	0.015625
Radix Exponential	8^2	8^1	8^0		8^{-1}	8^{-2}
Positional Value	3 x 64 (192)	7 x 8 (56)	2 x 1 (2)	.	3 x 0.1 (0.4)	1 x 0.015625 (0.015625)
				↑ Radix Point		
Final answer: 192 + 56 + 2 + 0.4 + 0.015625 = 250.415625						

Table 13.2: The solution to Example 13-1.

Example 13-2: Hexadecimal-to-decimal conversion

Convert $1CE.A4_{16}$ (hexadecimal) to decimal.

Solution: Table 13.3 provides the solution to Example 13-2. As you can see, the solution is strangely similar to Example 13-1.

Binary Value of Digit Weight	256	16	1		0.0625	0.003906
Radix Exponential	16^2	16^1	16^0		16^{-1}	16^{-2}
Positional Value	1 x 256 (256)	12 x 16 (192)	14 x 1 (14)	.	10 x 0. 0625 (0.625)	4 x 003906 (0.015625)
				↑ Radix Point		
Final answer: 256 + 192 + 14 + 0.0625 + 0.003906 = 462.066409						

Table 13.3: The solution to Example 13-2.

13.3.2 Decimal to Any Radix Conversion

Converting numbers from decimal to a number system of any radix can use one of many different algorithms. This section examines the most straightforward algorithm for humans. Although this approach will work for converting decimal to any base, we'll only be looking at decimal to binary conversion in this section. The best bet if you need to do conversions such as these is to use a calculator.

The decimal to binary conversion will be the conversion you use most often. In addition, since this method involves repeated division, it becomes very painful to make these conversions for anything except decimal to binary conversions. There are actually two parts to this approach; one for the integral portion and fractional portions of numbers.

As motivation for converting the integral portion of decimal number to binary, let's first convert a decimal number to a decimal number (don't worry, it actually proves a point). The approach we'll take is to divide the number multiple times by the radix value. Example 13-3 provides an overview of this division process.

Example 13-3: Decimal-to-decimal conversion

Convert 487 to decimal.

Solution: Table 13.4 shows the solution to this example. The solution comprises of repeated divisions with the top row of table being the first division.

487 ÷ 10 = 48	Remainder: 7	LSD = 7
48 ÷ 10 = 4	Remainder: 8	
4 ÷ 10 = 0	Remainder: 4	MSD = 4

Table 13.4: Decomposing an integral decimal number into a decimal number.

From Example 13-3, you can see that the repeated division by the radix value decomposes the original value into its individual weighted components. The first value that was generated by using this algorithm was the least significant digit (LSD) which was the remainder after the first division. The final value generated by this algorithm is the most significant digit (MSD). If you were to reassemble the number with the MSD on the left and the LSD on the right, you would get the original number back. Wow!

Although this example was somewhat funny because it did no actual conversion, it proves that the algorithm is valid and it will work when transferring from decimal to a number of any radix value. Example 13-4 shows an example of a decimal-to-binary conversion while an even more meaningful example appears in Example 13-5. Note that in both of these examples that we're using the terms LSB and MSB. These are common digital design terms and stand for Most Significant Bit (MSB) and Least Significant Bit (LSB). Not surprisingly, the technique is referred to as *repeated radix division* (RRD). For those brave enough to try, it truly does work for any radix.

Example 13-4: Decimal-to-binary conversion

Convert 12 to binary.

Solution: Table 13.5 shows the solution to this example in a series of steps starting with the top row of the table.

12 ÷ 2 = 6	Remainder: 0	LSB = 0
6 ÷ 2 = 3	Remainder: 0	
3 ÷ 2 = 1	Remainder: 1	
1 ÷ 2 = 0	Remainder: 1	MSB = 1

Final Answer:
$12_{10} = 1100_2$

Table 13.5: The solution to Example 13-4: Decomposing a decimal number into a binary number.

Example 13-5: Decimal-to-binary conversion.

Convert 147 to binary.

Solution: Table 13.6 shows the solution to this example in a series of eight steps starting with the top row of the table being the first step.

147 ÷ 2 = 73	Remainder: 1	LSB = 1
73 ÷ 2 = 36	Remainder: 1	
36 ÷ 2 = 18	Remainder: 0	
18 ÷ 2 = 9	Remainder: 0	
9 ÷ 2 = 4	Remainder: 1	
4 ÷ 2 = 2	Remainder: 0	
2 ÷ 2 = 1	Remainder: 0	
1 ÷ 2 = 0	Remainder: 1	MSB = 1

Final Answer:
$147_{10} = 10010011_2$

Table 13.6: The solution to Example 13-5; decomposing a yet larger integral decimal number into a yet larger binary number.

As a motivational example for converting the fractional portion of a number to some other base, let's first convert a fractional decimal number to decimal number. Yes, this too has a point. The approach we'll take is to multiply the number repeatedly by the radix value and see what the result is. In each step, we'll peel off the newly created integral portion of the number and put it aside. Example 13-6 provides an overview of this algorithm. Note from the result shown in Example 13-6 that the first integral result is the MSD of the original number. The final value we obtain is the LSD of the original number. This algorithm is referred to as *repeated radix multiplication* (RRM). Example 13-7 and Example 13-8 show two more somewhat worthy examples of this algorithm at its finest.

There are two key points about the example shown in Example 13-8. First, as opposed to the example shown in Example 13-7, the example in Example 13-8 does not appear to end. For the sake of sanity in this example, we decided to end the pain after four iterations of the algorithm. Stopping the algorithm after four iterations is arbitrary; doing four iterations was boring enough. The other key point about this example is that the answer we obtained is no longer a proper equation. In reality, since our conversion never ended as nicely as the example of Example 13-7, we must use the approximation symbol to indicate that the equality was not preserved. Also, note that all of these examples use a subscripted two to indicate that the converted number is in a binary representation. Without this subscription, we would have to interpret this number as decimal thus pissing off the digital goddesses.

Example 13-6: Decimal-to-decimal conversion (fractional)

Convert 0.243 to decimal.

Solution: Table 13.7 shows the solution to this example in a series of eight steps starting with the top row of the table.

0.243 × 10 = 2.43	remove the 2	MSD = 2
0.43 × 10 = 4.2	remove the 4	
0.3× 10 = 3.0	remove the 3	LSD = 3

Table 13.7: Solution to Example 13-6: Decomposing a fractional decimal number into a decimal number.

Example 13-7: Decimal-to-binary conversion (fractional)

Convert 0.375 to binary.

Solution: Table 13.8 shows the solution to this example in a few steps starting with the top row of the table being the first step.

$0.375 \times 2 = 0.75$	remove the 0	MSB = 0
$0.75 \times 2 = 1.50$	remove the 1	
$0.5 \times 2 = 1.0$	remove the 1	LSB = 1

$\mathbf{0.375 = 0.011_2}$

Table 13.8: Solution to Example 13-7: Decomposing a fractional decimal number into a binary number.

Example 13-8: Decimal-to-binary conversion (fractional)

Convert 0.879 to binary.

Solution: Table 13.9 shows the solution to this example in a few steps starting with the top row of the table being the first step.

$0.879 \times 2 = 1.758$	remove the 1	MSB = 1
$0.758 \times 2 = 1.516$	remove the 1	
$0.516 \times 2 = 1.032$	remove the 1	
$0.032 \times 2 = 0.064$	remove the 0	LSB = 0 (?)

$\mathbf{0.879 \approx 0.1110_2}$

Table 13.9: Solution to Example 13-8; decomposing a fractional decimal number into a binary number.

13.3.3 Binary ↔ Hex Conversions

Although binary is the official language of digital circuits, it is problematic to look at the seemingly endless strings of 1's and 0's that form binary numbers. The problem is that our minds don't easily recognize a large string of bits too well. As an example of this, consider the fact that nine bits looks a lot like eight bits at a quick glance. To get around this dilemma, we use hexadecimal and octal representations for binary numbers wherever possible. This makes the numbers much more readable; the conversion process is also quite friendly.

The key to converting between binary and hex numbers is to note that a single hex number can represent a group of four binary numbers (and vice versa). This works because both binary and hex numbers are powers of two, which allows for the individual weightings of the numbers to be powers of two also. The conversions shown in Example 13-9 and Example 13-10 highlight the relationship between the group of fours in the context of a binary-to-hexadecimal conversion and a hexadecimal-to-binary conversion, respectively.

There are a few special items to note in these examples.

- In Figure 13.1 the leading zeros in the number were omitted. Since zero has no value, you can ignore them (but don't forget they're not there).
- Zeros are added to the end of the fractional portion of the number (commonly referred to as bit-stuffing). A common mistake is to see that final '1' in the fractional portion of the number think that is equivalent to a binary '1'. This is not the case; actually the number has the weight associated with the MSB of a 4-bit binary number. In other words, the final bit is associated with a hexadecimal '8' and not '1'. Be careful to not make this error.
- Note that in these examples the numbers have the radii clearly indicated thus indicating mastery of the subject matter at hand.

Example 13-9: Binary-to-hexadecimal conversion

Convert 1100110.10101_2 to hexadecimal.

Solution: Figure 13.1 shows the solution to Example 13-9.

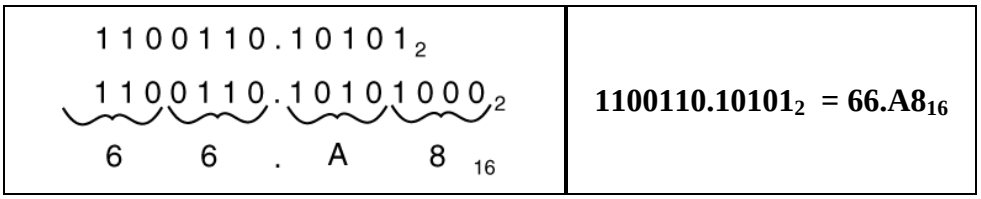


Figure 13.1: The solution to Example 13-9.

Example 13-10: Hexadecimal-to-binary conversion

Convert $D37.AC_{16}$ to binary.

Solution: Figure 13.2 shows the solution to Example 13-10.

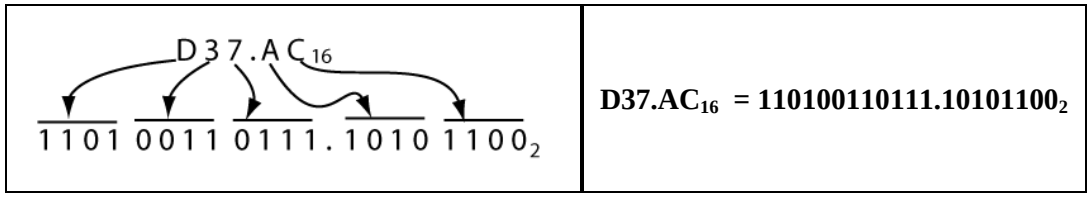


Figure 13.2: The solution to Example 13-10.

13.3.4 Binary ↔ Octal Conversions

Converting between binary and octal numbers is similar to the binary-to-hex conversion. While the binary-to-hex conversions use the "group of fours" approach, the binary-octal conversions use the "group of threes" approach. Once again, this method works because we're transferring back and forth between radii related by powers of two. The "group of threes" approach is inherent the same as the "group of fours" approach so less comment is provided there. The key here is to bit-stuff the fractional portion of the binary number being converted so that you don't make a mistake in the weighting.

Example 13-11: Binary-to-octal conversion

Convert 1101101.11011_2 to octal.

Solution: Figure 13.3 shows the solution to Example 13-11.

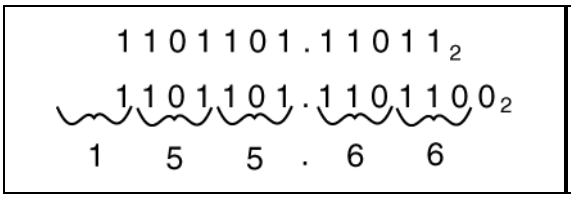


Figure 13.3: The solution to Example 13-11.

Example 13-12: Octal-to-binary conversion

Convert 241.32_8 to binary

Solution: Figure 13.4 shows the solution to Example 13-12.

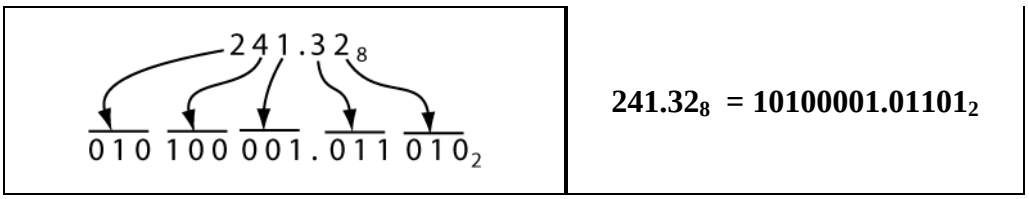


Figure 13.4: The solution to Example 13-12.

13.4 Other Useful Codes

Using binary patterns to represent numbers is a major field of study in modern engineering. Generally speaking, binary codes are generated in order to efficiently represent some given information. In that we currently live in the information age, there are literally and endless number of binary codes in use. Moreover, there's nothing to stop you from generating your own binary code just for the heck of it. Despite the fact that there are such a great number of binary codes in use today, a few highly useful codes are worth looking at. These are the binary coded decimal and unit distance codes.

13.4.1 Binary Coded Decimal Numbers (BCD)

You're about to learn several different common ways of representing numbers using binary codes. In this context, the word "code" refers to the interpretation of a set of bits. Up until this point, if you were to see a bunch of bits, you would naturally think about juxtapositional notation and the weights of the numbers, which happen to be powers of two (the radix for binary). For example, going from the radix point and moving to the left, the weights associated with each bit position are increasing powers of two. As you'll soon find out, this is only true for unsigned binary numbers in one particular format; we'll need many more representations to be fluent in digital-land.

Binary coded decimal (BCD) numbers are somewhat similar to the group of fours so we'll talk about it in this section. The goal is to have a unique set of bits to represent each of the digits in the decimal system. Since there are ten different numbers in the decimal system, we're going to need at least four bits to uniquely represent each of the digits. We could not represent the set of decimal numbers with three bits because with three bits, we only have eight different unique bit patterns which is not be sufficient to represent the ten symbols in the decimal number system. On the other hand, there is nothing stopping us from using more than four bits to represent the digits but that would end up have lots of unassigned codes. As it is, there are sixteen different bit combinations possible with four bits, which results in six of the bit combinations not used when representing the set of decimal digits[149].

Table 13.10 shows the four-bit code words and the decimal digits they represent. Note that these are the same as the group of fours approach for the first ten rows. After that, we run out of decimal digits and then have to take our shoes off.

[149] Although these six combinations are often used to represent "numbers" 10-15 hexadecimal.

Decimal	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
-	1010
-	1011
-	1100
-	1101
-	1110
-	1111

Table 13.10: The decimal digits and their associated BCD codes.

The primary role of BCD numbers is to represent decimal number on certain types of displays in devices that are able to display decimal numbers. For now, let's do a couple of examples. As you'll see, this is similar to stuff you've already done. Example 13-13 and Example 13-14 provide two examples of these conversions. One thing to note from these examples is that these representations generally show the leading zeros in the BCD numbers. In addition, these two figures have slight spaces between the bits to somewhat group them into groups of four bits, which is shown only to help you read these numbers late at night.

Example 13-13: BCD-to-decimal conversion

Convert 011001111000 (BCD) to decimal.

Solution: Figure 13.5 shows the solution to Example 13-13.

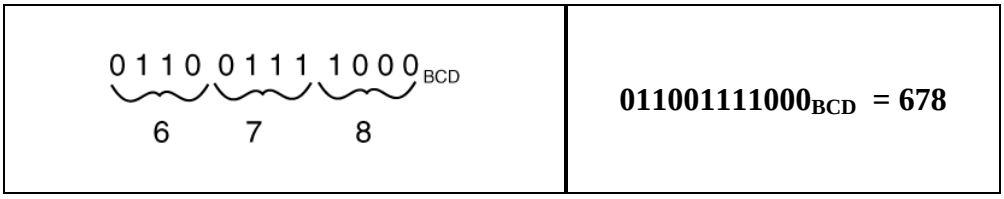
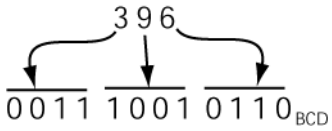


Figure 13.5: The solution to Example 13-13.

Example 13-14: Decimal-to-BCD conversion

Convert 396 to BCD.

Solution: Figure 13.6 shows the solution to Example 13-14.



$$396 = 001110010110_{BCD}$$

Figure 13.6: The solution to Example 13-14.

13.4.2 Unit Distance Codes (UDC)

The concept of "distance" in digital-land has a special and relatively simple meaning. When you see the word distance, it's usually in the context of "the distance between two code words". What this implies is that you have a given set of binary code words of equal length; the set of codes also has a specified sequence[150]. In this context, each of the code words is different from all of the other code words in the set. Since this set of code words now has order, uniqueness, and a specified constant bit-length, we can talk about the distance between two code words in the set.

An example of a code set would be the binary numbers associated with the decimal range [0,15] which could be a represented with a 4-bit binary code. Table 13.11 shows an example of a 5-bit binary code. As you can see from Table 13.11, the distance between two code words is defined as the number of bits that must be toggled (inverted) to form one code word out of a contiguous code word in the set. Table 13.11 shows a few examples of distance between code words.

Code Word A	Code Word B	Distance from Word A to Word B	Comment
00000	11111	5	5 bits must be toggled
01110	00110	1	Toggle second bit from right
00111	11100	4	Toggle outer two bits

Table 13.11: A few examples of "distances" between code words.

A unit distance code (UDC) is a set of code words where the maximum distance between any two sequential code words is one. In other words, to get from one code word to the next code word in the sequence, you only need to toggle one bit. UDCs are quite important in several areas of digital-land. You've already been exposed to UDCs but don't worry about going to the health center to have yourself checked-out or anything like that. We essential changed the cell numbering in a K-maps in order to obtain unit distance ordering on both the rows and columns which allowed the Adjacency theorem to be applied in a visual manner.

[150] Keep in mind that the standard binary count you're used to using is somewhat arbitrary.

There is actually a science to creating UDCs but we'll not go into that here. Just know where you hear the words "unit distance" that it's describing a relationship between two binary number used to represent something of importance. Also good to note here is that a special form of UDCs are *Gray Codes*. Often times when people mention Gray and Unit Distance codes, they're actually referring the unit distance property and not the special characteristics associated with Gray codes[151]. Table 13.12 lists a few examples of UDCs.

2-bit UDC	4-bit UDC	8-bit UDC
00	0001	10000001
01	0011	11000001
11	0111	11000011
10	1111	11100011
	1110	11100111
	1100	01100111
	1000	01100110
	0000	00100110
		00100100
		00000100
		00000000
		10000000

Table 13.12: Examples of 2, 3, and 8-bit UDC codes.

[151] I simply can't remember what they are right now.

Chapter Summary

- Hexadecimal (base 16) and octal (base 8) are two of the primary number systems commonly used and associated with digital design. Hexadecimal is the more popular representation.

- Conversion between numbers used in digital design is often required. The important most common conversions are decimal-to-binary, binary-to-decimal, octal-to-binary, binary-to-octal, hexadecimal-to-binary, and binary-to-hexadecimal. Each of these conversions uses special algorithms.

- Binary coded decimal (BCD) and unit distance codes (UDCs) are two of the commonly used binary codes in digital logic.

Chapter Exercises

1) Explain briefly but fully why the group of four approach works for converting number between hexadecimal and binary representations.

2) Convert $2AF6.E7_{16}$ to octal.

3) Convert 721.32_8 to hex.

4) Convert 312_{BCD} to binary.

5) Convert 789_{BCD} to hexadecimal.

6) Convert 10011110_2 to BCD.

7) Convert $B3C_{16}$ to BCD.

8) Multiply 101011011.11_2 by 8.

9) Divide 4573.234_8 by 64.

10) Divide $1AF.3D_{16}$ by 8.

11) Multiply 345.72_8 by 4.

12) What is the minimum radix value of the following number?: 145.801

13) What is the minimum radix value of the following number?: BA.123

14) Which of these two positive numbers is greater? 533.55_8 or $15B.B_{16}$

15) Which of these two positive numbers is greater? $1F3.E_{16}$ or 499.75_{10}

16) Assemble these numbers into a gray code sequence: 111, 000, 110, 011, 001, 100.

17) Can the follow set of number be made to form a gray code?:
0011, 0110, 1100, 0111, 1111, 1110, 0001.

18) What is the maximum distance between any two of the following numbers?
0011, 0110, 1100, 0111, 1111, 1110, 0001.

19) In the table below, cross out *one* code word from each column to make the code shown in the column into a unit distance code. These two columns represent two *separate* unit distance codes.

0000		00000
0010		10000
0110		10001
1110		11001
1111		11011
1100		10111
1101		10011
1001		10010
0001		00010

20) Create a 6-bit *unit distance code* that contains at least eight unique code words. The first code word should be a unit distance from the last code word (circular).

21) In the table below, add *one* code word to each column to make the code shown in the column into a unit distance code. Add the required code words only in the rows indicated with arrows. These two columns represent two *separate* unit distance codes – your answer will not necessarily be the same code word for each code.

0000		0000
0100		0001
0110		0011
0010		0111
0011		0110
	← →	
1111		1100
1110		1000
1100		
1000		

22) The table below shows five binary codes. Circle the codes that are unit distance codes.

000 001 011 111 110 100	0000 1000 0100 0010 0001	0000 0001 0011 0010 0110 0100 1100 1000	01000 01001 01011 01111 11111 01111 01110 01100 00100 00000	00000 00100 01100 01110 11111 11110 11100 11000 10000

Chapter Design Problems

1) Design a unit distance code that contains six code words. The code should be circular in nature and each code word should be five bits long.

2) Design a circuit that converts a 2-digit BCD number into a 7-bit binary number. If you use something other than a standard digital module in your design, be sure to provide a model for your creation. For this problem, you can assume the BCD number is between [0,15].

14 Chapter Fourteen

(Bryan Mealy 2012 ©)

14.1 Chapter Overview

Now that you're more comfortable with number systems and various manipulations of number in various radii, it's time to delve into the details of doing math with binary numbers. While this introduction is far from being complete, it will provide you with the background you can use as a starting point for doing more exciting math operations using digital circuits.

Main Chapter Topics

- **BINARY NUMBER REPRESENTATIONS:** This chapter presents common representations of signed binary number. These representations include sign magnitude, radix complement, and diminished radix complement.
- **BINARY ARITHMETIC:** This chapter presents the basics of binary arithmetic using signed and unsigned binary numbers. The emphasis is on fixed number lengths and detection of result validity after mathematical operations.

Why This Chapter is Important

This chapter is important because is describes the basic representations of signed and unsigned binary numbers. In addition, this chapter describes mathematical operations (addition and subtraction) on binary numbers, which form the basis of many digital circuits.

14.2 Signed Binary Number Representations

As you probably know by now, computers only have the ability to represent numbers with ones and zeros. This is all fine and good for positive numbers but is seemingly inadequate for negative numbers. There is of course no problem when you're simply writing numbers on a piece of paper because all you need to do is drop a "-" in front of the number and everyone agrees that such a number is a "negative" number. The other accepted numerical tradition is that when the number is positive, a "+" sign usually does not appear in front of the number. Sadly enough, computers generally don't rely on tradition in order to do what they do. The reality is that computers don't have an easy and efficient way to place a "-" sign front of numbers that are meant to be interpreted as negative. But alas, there is hope.

This section presents an overview of representing signed numbers using only the set of symbols associated with the binary number system (namely 1's and 0's). There are three standard methods used to represent signed numbers in binary notation; each of these representations has their good and bad points, which we'll discuss later. Once we introduce these representations, we'll concern ourselves with

the issues regarding the number ranges of these signed representation and standard mathematical operations with the most common of these representations.

14.2.1 Representing Signed Numbers in Binary Notation

There are actually an infinite number of ways to represent signed numbers using binary notation. You can make up any number of ways and they would be just as valid as any other way[152]. However, there are a few standard ways used to represent signed binary numbers, which we'll discuss in this section. In particular, there are three representations of interest: *sign magnitude (SM)*, *diminished radix complement (DRC)*, and *radix complement (RC)*. The reality is that the most widely used is RC notation but we'll be working with all three and classify the work we do with the less used notations as a wicked academic exercise.

Before we start, let's exploit the similarities between these three representations. Keep in mind that we only have the option of using ones or zeros to represent negative numbers. The easiest and most efficient approach to represent sign numbers is to use a single bit, such as a '1', to indicate that a particular number is negative. The key to this method is to agree upon a standard location for this bit. The accepted position of this bit is in the most significant bit position (highest weighting or left-most) of the given number. Once again, the most significant bit position is generally the left-most bit position; this position is commonly referred to as the MSB position, which not surprisingly stands for "most significant bit". This effectively separates the number into a bit that represents the sign and some bits that represent the magnitude.

The MSB in every signed number representation we discuss in this chapter is the *sign bit*. If the sign bit is a '1', then the number is interpreted as negative with the magnitude being represented by the magnitude bits. If the sign bit is a '0', the number is a positive number with a magnitude represented by the magnitude bits. Figure 14.1 provides a visual representation of the bit positions of the sign and magnitude bits.

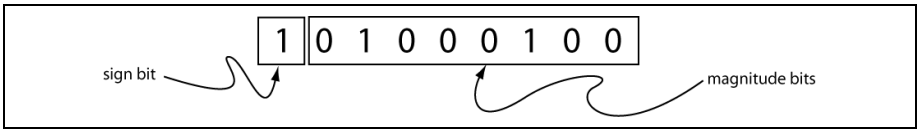


Figure 14.1: Some generic nine-bit number that is interpreted as being signed.

14.2.2 Sign Magnitude Notation (SM):

Sign Magnitude, or SM, s the most straight-forward of the three notations because SM notation closely resembles the original model of signed numbers presented in the previous paragraph. In SM notation, the sign bit indicates the sign of the number and the other bits represent the magnitude of the number. Table 14.1 listed everything you may want to know about tweaking SM numbers.

[152] Administrators do this all the time.

Operation	Procedure
Multiply number by -1	toggle (change state) the sign bit
Convert positive SM to decimal equivalent	apply binary-to-decimal conversion on magnitude bits
Convert negative SM to decimal equivalent	1) note that the number is negative 2) do binary to decimal conversion on magnitude bits 3) add in minus sign (from step 1)

Table 14.1: Standard operations on binary numbers represented in SM.

Example 14-1

Change the sign of the following binary numbers represented in SM:

a) 01100001_2
b) 110011_2

Solution: Changing the sign involves toggling the sign bit and doing nothing to the magnitude bits. Note that you don't need to know the decimal equivalents of these binary numbers in order to complete this problem.

a) 11100001_2
b) 010011_2

Example 14-2

Convert the following binary numbers represented in SM to their decimal equivalents:

a) 01100001_2
b) 110011_2

Solution: a) This number is an 8-bit positive number. The number converts directly to decimal since the sign bit is zero and thus adds nothing to the final decimal number. The answer is 97.

b) This number is a negative 6-bit binary number. The number is converted to decimal by first noting that the number is negative and then performing a binary-to-decimal conversion on the magnitude bits. The magnitude bits are 10011_2, which represent 19 in decimal. Adding the negative sign complete the solution: -19.

14.2.3 Diminished Radix Complement (DRC)

Diminished Radix Complement representations, or DRC, is best explained by the operations required to change the sign of the number. Once again, this is a straight-forward matter: toggle all the bits in the binary number (referred to as a *1's complement*). In DRC notation, the sign bit indicates the sign of the number and the other bits represent the magnitude of the number (but positive and negative numbers represent their magnitude's differently). Table 14.2 lists everything you may want to know about tweaking DRC numbers.

Operation	Procedure
Multiply number by -1	toggle all the bits (1's complement)
Convert positive DRC to decimal equivalent	do binary to decimal conversion on magnitude bits
Convert negative DRC to decimal equivalent	1) note that the number is negative 2) toggle all the bits (1's complement) 3) do binary to decimal conversion on magnitude bits 4) add in minus sign (from step 1)

Table 14.2: Standard operations on binary numbers represented in DRC.

Example 14-3

Change the sign of the following binary numbers represented in DRC:

a) 01110001_2
b) 1001101_2

Solution: Changing the sign involves toggling all the bits. This problem is doable without knowing the decimal equivalents of the binary numbers.

a) 10001110_2
b) 0110010_2

Example 14-4

Convert the following binary numbers represented in DRC to their decimal equivalents:

a) 01110001_2
b) 110011_2

Solution: a) This number is an 8-bit positive number. Conversion to decimal can be done directly using standard binary-to-decimal conversion techniques since the sign bit is zero and will add nothing to the final decimal number. The answer is 113.

b) This number is a negative 6-bit binary number. Convert it to decimal by 1) noting that the number is negative, 2) toggling all the bits, 3) doing a decimal-to-binary conversion on the resulting number, and 4) adding the negative sign.

1) Yep, its negative
2) $110011_2 \rightarrow 001100_2$
3) 001100_2 represents 12 in decimal
4) Adding the negative sign completes the solution: -12

14.2.4 Radix Complement (RC):

RC representation is once again best explained by the operations required to toggle the sign of the number. This operation is somewhat straightforward yet not as simple as the SM and DRC representations. In RC notation, the sign bit indicates the sign of the number and the other bits represent the magnitude of the number. The magnitude bits are once again interpreted differently for positive and negative numbers. For positive numbers, the magnitude bits are interpreted directly as a simple binary number. If the number is negative, the magnitude bits are considered to be in a *two's complement* representation. Table 14.3 lists everything you may want to know about tweaking RC numbers.

Operation	Procedure
Multiply number by -1	take the two's complement of the number
Convert positive RC to decimal equivalent	do binary to decimal conversion on magnitude bits
Convert negative RC to decimal equivalent	1) note that the number is negative 2) take the two's complement of the number 3) do binary to decimal conversion on magnitude bits 4) add in minus sign (from step 1)

Table 14.3: Standard operations on binary numbers represented in RC.

Finding the two's complement of a number can be done by hand in two different ways. The two's complement is defined as "one greater than the 1's complement". This means that to find the 2's complement of a binary number, you toggle all the bits (the 1's complement) and then add 1 to the result. Though this works fine, it can sometimes lead to errors since you'll possibly need to deal with a carry bit across the span of the number. There is hope though.

The easiest way to find the 2's complement of a number is to apply the following trick[153]: starting from the right-most bit in the binary number, examine each bit from right to left. When you encounter a '1', toggle every bit after the first '1' bit that is found (but don't toggle the first '1' bit). A few examples of this will drive the point home. Figure 14.2 shows just about every case you'll ever hope to run across. In Figure 14.2, NC stands for "no change" while TOG stands for "toggle".

[153] This is actually not a trick; it is more of an algorithm.

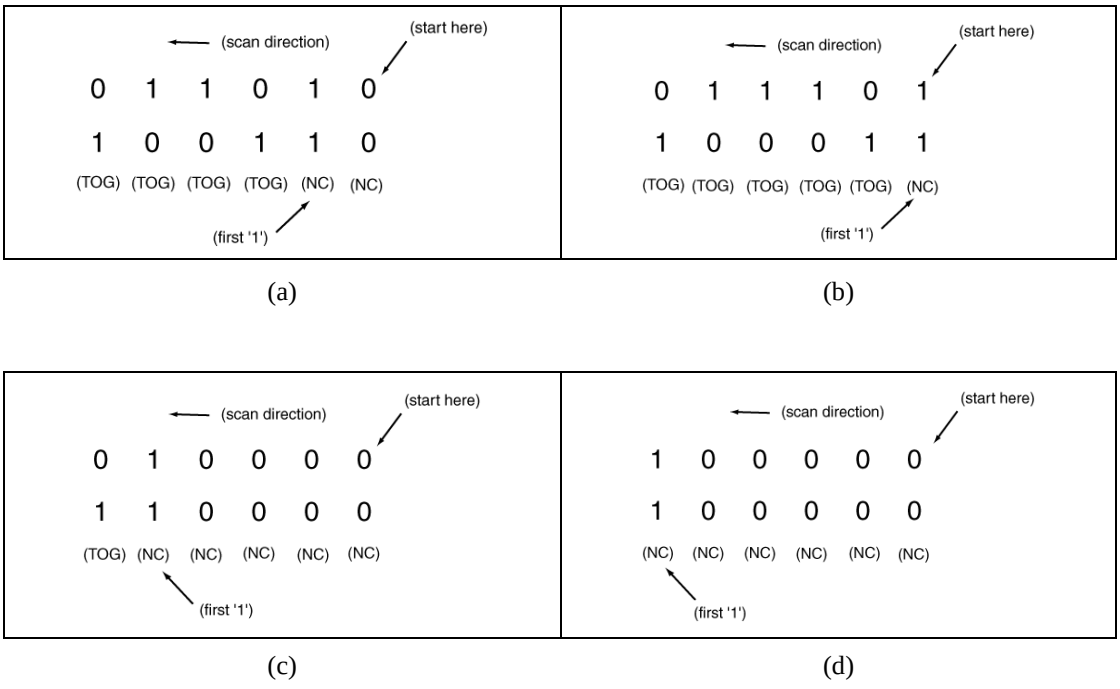


Figure 14.2: Four examples showing the 2's complement conversion algorithm.

Example 14-5

Change the sign of the following binary numbers represented in RC:

a) 00110101_2,
b) 1001101_2.

Solution: Changing the sign involves taking the two's complement of the numbers. You don't need to know the decimal equivalents of these numbers in order to complete this example.

a) 11001011_2
b) 0110011_2

Example 14-6

Convert the following binary numbers represented in RC to their decimal equivalents:

a) 00110101_2
b) 1001101_2

Solution: a) This number is an 8-bit positive number. Conversion to decimal can be done directly using standard binary to decimal conversion techniques since the sign bit is zero and will add nothing to the final decimal number. The answer is 53.

b) This number is a negative 7-bit binary number. Conversion to decimal is done by 1) noting that the number is negative, 2) taking the two's complement, and 3) doing a decimal to binary conversion on the resulting number, and 4) tacking on a negative sign to the result.

1) Yep, by golly, its negative
2) $1001101_2 \rightarrow 0110011_2$
3) 0110011_2 represents 51 in decimal
4) Adding the negative sign completes the solution: -51

14.2.5 Number Ranges in SM, DRC, and RC Notations

The reality of representing sign numbers in binary is that an extra bit (the sign bit) is used to represent the sign. It almost seems that this means one less bit can be used to represent the magnitude of the number and only one-half as many numbers can be represented by the same amount of bits[154]. This is not exactly the case. The reality is that, generally speaking, the ranges of numbers that are representable with a binary number shift downwards when a sign bit is used. The resulting range is still the same but it no longer starts at zero (as it does for an unsigned binary number); the range of a signed binary number is now centered about zero. Figure 14.3 visually shows what the last few sentences are attempting to say.

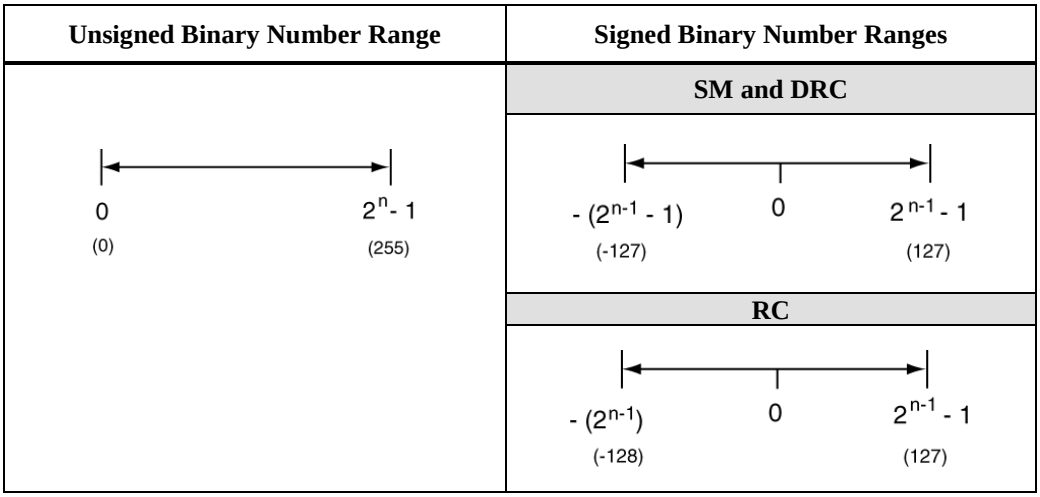


Figure 14.3: Number ranges for signed and unsigned binary numbers (n=8).

The key to understanding Figure 14.3 is that the letter *n* represents the number of bits in the binary value. The smaller numbers in parenthesis in Figure 14.3 shows the number ranges when *n=8*, which is a common bit-width in digital-land. The really important thing to notice about Figure 14.3 is the fact

[154] If this does not make sense, think about it for a minute. If there is one bit dedicated to the sign bit, doesn't that mean that there is one less bit to have a "weighting" in the number?

that with SM and DRC representations, only 2^n-1 out of the 2^n possible values for a given value of *n* are representable in those notations. However, with RC, all 2^n possible values are represented. This is why computers commonly use RC for signed binary number representations.

The section that follows is slightly painful. But… if you're able to grasp the ideas presented in this section, you'll be much better off in digital design, computer science, computer engineering, and croquet. The ideas are not that complicated once you get used to them. In addition, getting use to them will assuredly give you direct benefits down the line. Keep in mind that just about everyone is weak when it comes to the notion of 2's complement math: they get by because they rely on some other entity to mask their lack of understanding of the concepts. Don't be one of these people.

14.3 Binary Addition and Subtraction

There are a few recurring topics regarding the addition and subtraction of binary numbers. These topics are not overly complicated but they can seem somewhat strange when you first encounter them. The topic of binary arithmetic and computers is a deep subject that many people spend their entire lives studying. Generally speaking, if you can design a computer to perform efficient mathematical operations, you'll have a good computer (based on your definition of *good*). The problem is that there are a bunch of trade-offs along the way. Without doubt, you'll run into some of these topics later in your digital/computer education but they're beyond the scope of this discussion. This discussion is limited to the issues involved with addition and subtraction of signed and unsigned binary numbers.

This discussion lives in the context of how a computer would actually perform addition and subtraction. Namely, computers are comprised of a fixed set of hardware. What this means to us is that there are fixed sizes of registers[155] that can be used to perform the arithmetic operations. In other words, the precision of the arithmetic performed by the computers is limited to some pre-determined value. For example, the number ranges shown in Figure 14.3 are based on a fixed register size (and hence, word length) of eight bits.

The ramifications of a fixed register size is that your mathematical operations must stay within these limits if want the bits representing the result of your operation to be valid. The reality is that if you stay within these limits, your fixed result will be valid; if you exceed the limits of the data you're using, you're answer will be invalid. The crux of this discussion is that you'll want to know when you've exceeded these limits so you can know whether your answer is valid or not. There are two main ways to exceed these limits: 1) go over the stated number range for the size of the data you're using, or 2) go under the stated range of data you're using[156]. Keep in mind that we'll need to continue this discussion for both unsigned and signed binary data. The good thing here is that we'll limit our discussion to RC representations only.

14.3.1 Binary Subtraction

One of the many recurring themes in digital design-land is the fact that you always want to design your circuits to do what they need to do but to do it using as little hardware as possible. Mathematical operations in computers do not come free: they are done by hardware that you'll soon be learning about and designing. Hardware, or digital circuitry, takes up space and consumes power. Generally speaking, the less circuitry your design contains, the better off the world will be.

[155] A register is a piece of hardware that stores a given number of bits. Data inside of a computer is transferred around via these registers. The bit-width of these registers is fixed in a particular computer's hardware.
[156] Going over or under the stated range means the number is off the left or right side of the number line shown in Figure 14.3, respectively. Make sure you really understand this statement.

These factors play out directly in this discussion in the context of binary subtraction. Although it would not be a big deal to design a circuit that did subtraction, the cool approach is to use a circuit we've already designed to perform subtraction. This is not a big deal in digital design-land and is done quite often. The approach we'll take is to use our ripple carry adder to apply *indirect subtraction by addition*. Equation 14-1: shows the basic formula for this approach. This concept is something you've used extensively in standard mathematics; the only new thing here is that we'll be applying this concept in the context of binary subtraction.

$$N1 - N2 = N1 + (-N2)$$

Equation 14-1: Indirect subtraction by addition.

Changing the sign of a number is no big deal when dealing with RC numbers: all you need to do is take the two's complement. In other words, all you need to do in order to subtract one binary number from another is to take the two's complement of that number and add it to the other number (which is what Equation 14-1 is saying). After this addition operation, you need to examine a few items that will tell you if your result is valid or not because your result may exceed the number range you're working with.

Some useful definitions involving the addition and subtraction of two numbers are appropriate here. Consider adding two numbers A and B with a result C. The equation for this operation would look like: A + B = C. The number represented by the variable A is referred to as the *augend*, B is referred to as the *addend*, and C is referred to as the *sum*. Consider subtracting one number B from another number A with a result C. In case, A is referred to as the *minuend*, B is referred to as the *subtrahend*, and C is referred to as the *difference*. This knowledge could be actually valuable if you were to find yourself on Jeopardy but it does not get a lot of mileage outside of this discussion.

14.3.2 Addition and Subtraction on Unsigned Binary Numbers

When dealing with unsigned binary numbers, the results of your mathematical operation can either underflow or overflow the given number range. Underflow would be the result of subtracting a given binary number from a smaller binary number (the result would be negative which would violate the *unsignedness* of the number). Overflow would result when the addition of two numbers exceeds the top-end of the given range[157]. Table 14.4 and Table 14.5 list everything you need to know about the overflow and underflow of binary numbers. These two examples arbitrarily use four-bit numbers. Also, the extra bit to the left of the four-bit result is the carry bit from the given operation.

[157] An issue here is that "overflow" is often used to describe both underflow and overflow. The notion here is that you can exceed, or "overflow", the given range in either direction.

Overflow in Unsigned Binary Addition			
Description	The sum of two binary numbers exceeds the number range associated with the operation		
Indicator	The carry-out from the MSB addition is '1'.		
Example 14-7	1001 + 0011 = ?	<table><tr><td></td><td>1001</td></tr><tr><td>+</td><td>0011</td></tr><tr><td>0</td><td>1100</td></tr></table>	The carry from the MSB is 0 which indicates there was no carry. Therefore, the sum (the four-bit result) is a *valid*.
Example 14-8	1011 + 0111 = ?	<table><tr><td></td><td>1011</td></tr><tr><td>+</td><td>0111</td></tr><tr><td>1</td><td>0010</td></tr></table>	The carry out of the MSB is 1 which indicates there was a carry. Therefore, the sum (the four-bit result) is *not valid*.

Table 14.4: The low-down on unsigned overflow.

Underflow in Unsigned Binary Subtraction			
Description	The difference between two binary numbers is below the number range associated with the operation.		
Indicator	The carry-out from the MSB addition is '0'.		
Example 14-9	1001 - 0011 = ?	add the negation of 0011 (two's complement) <table><tr><td></td><td>1001</td></tr><tr><td>+</td><td>1101</td></tr><tr><td>1</td><td>0110</td></tr></table>	The carry from the MSB is '1', which indicates there was a carry. There was no underflow and the difference (the four-bit result) is a *valid*.
Example 14-10	0111 - 1100 = ?	add the negation of 1100 (two's complement) <table><tr><td></td><td>0111</td></tr><tr><td>+</td><td>0100</td></tr><tr><td>0</td><td>1011</td></tr></table>	The carry out of the MSB is '0', which indicates there was no carry. An underflow has occurred and the difference (the four-bit result) is *not valid*.

Table 14.5: The low-down on unsigned underflow.

14.3.3 Addition and Subtraction on Signed Binary Numbers

When dealing with signed binary numbers, the results of your mathematical operations can once again both underflow and overflow the given number range. The approach to dealing with operations on signed binary number is much more intuitive than dealing with unsigned binary numbers. The list below describes two concepts that you'll always need to keep in mind[158]. There is a science behind all

[158] Remember that we are now dealing with signed binary numbers, which means that the MSB in the numbers is a sign bit.

of this but understanding the basic principles will allow you to work with this type of problem without clogging your brain by memorizing this stuff.

1) **Overflow can never occur if you're adding a positive number to a negative number**. This concept affects both addition and subtraction keeping in mind that we do subtraction by negating the subtrahend and adding it. In other words, the result from the operation A - B will always be valid if both A and B are positive numbers or if A & B are both negative numbers. In either case, you're essentially adding a negative number to a positive number. In even other words, if the two numbers have different sign bits before the final addition is done[159], the answer is *guaranteed to be valid*.

2) **Overflow and underflow only occurs when you add to numbers that have the same value for sign bits but the result has a sign bit of a different value.** There is actually an easy way to check for this but we'll save the particulars of this operation until we start talking more about VHDL. The reality is that overflow and underflow can only happen in the following two scenarios:

    a. Overflow: Adding a positive number to a positive number. But due to the indirect subtraction by addition, this can include subtracting a negative number from a positive number.

    b. Underflow: Subtracting a positive number from a negative number. Also due to indirect subtraction by addition, this can include adding a negative number to a negative number.

The following examples in Table 14.6 and Table 14.7 spell out every possible scenario for both overflow and underflow in both addition and subtraction operations. Have fun.

[159] Keeping in mind that we can either add two numbers of different signs, or, we'll have to change the sign of one of the numbers when doing subtraction (indirect subtraction by addition).

Overflow in Signed Binary Addition and Subtraction			
Description	The result of an operation between two binary numbers is beyond the number range associated with the operation.		
Indicator	Two numbers of the same sign are added and the result is a number of a different sign (this is the direct addition of two numbers or the addition associated with the indirect subtraction by addition method).		

Example 14-11	0011 + 0010 = ?	`  0011`
		`+ 0010`
		`0 0101`
			The sign of addend and augend are positive and the sign of result is positive. No overflow has occurred in this operation and the result is *valid*.

Example 14-12	0100 + 1110 = ?	`  0100`
		`+ 1110`
		`1 0010`
			The sign of addend and augend are different so there can be no overflow or underflow. The result is *valid* (the carry is discarded).

Example 14-13	0110 + 0101 = ?	`  0110`
		`+ 0101`
		`0 1011`
			The sign of the addend and augend are the same (both indicate positive numbers) but are different from the sign of the result. The result is *not valid*: an overflow has occurred.

Example 14-14	0100 - 1110 = ?	add the negation of 1110
		`  0100`
		`+ 0010`
		`0 0110`
			The sign of addend and augend are positive (based on the indirect subtraction by addition) and the sign of result is positive. No overflow has occurred in this operation and the result is *valid*.

Example 14-15	0100 - 0011 = ?	add the negation of 0011
		`  0100`
		`+ 1101`
		`1 0001`
			The sign of addend and augend are different (based on the indirect subtraction by addition) so there can be no overflow or underflow. The result is valid (the carry is discarded).

Example 14-16	0100 - 1100 = ?	add the negation of 1100
		`  0100`
		`+ 0100`
		`0 1000`
			The sign of the addend and augend are the same (based on the indirect subtraction by addition) but are different from the sign of the result. The result is *not valid*: an overflow has occurred.

Table 14.6: The low-down on overflow in signed binary numbers.

Underflow in Signed Binary Addition and Subtraction			
Description	The result of an operation between two binary numbers is below the number range associated with the operation.		
Indicator	Two numbers of the same sign are added and the result is a number of a different sign (this is the direct addition of two numbers or the addition associated with the indirect subtraction by addition method).		

Example 14-17	1111 + 0010 = ?	`  1111`
		`+ 0010`
		`1 0001`
			The sign of addend and augend different so there can be no overflow or underflow. The result is *valid* (the carry is discarded).

Example 14-18	1110 + 1111 = ?	`  1110`
		`+ 1111`
		`1 1101`
			The sign of the addend and augend are the same (both indicate negative numbers) and match the sign of the result. The result is *valid* (and the carry is discarded).

Example 14-19	1100 + 1001 = ?	`  1100`
		`+ 1001`
		`1 0101`
			The sign of the addend and augend are the same (both indicate negative numbers) but are different from the sign of the result. The result is *not valid*.

Example 14-20	1110 - 1111 = ?	add the negation of 1111
		`  1110`
		`+ 0001`
		`0 1111`
			The sign of addend and augend are different (based on indirect subtraction by addition) so there can be no overflow or underflow. The result is *valid*.

Example 14-21	1100 - 0011 = ?	add the negation of 0011
		`  1100`
		`+ 1101`
		`1 1001`
			The sign of the addend and augend are the same (based on indirect subtraction by addition) and match the sign of the result. The result is *valid* (and the carry is discarded).

Example 14-22	1001 - 0110 = ?	add the negation of 1101
		`  1001`
		`+ 1010`
		`1 0011`
			The sign of the addend and augend are the same (based on indirect subtraction by addition) but are different from the sign of the result. The result is *not valid*.

Table 14.7: The low-down on underflow in signed binary numbers.

Example 14-23: SM Sign Changer

Design a circuit that compares the magnitude of two 8-bit binary numbers in signed magnitude form.

Solution: The first step in this problem is to determine the inputs and outputs of the circuit that the problem describes. The circuit has two 8-bit inputs for the two binary numbers, and, it seems only to have one output. Figure 14.4 shows the black box diagram for our interpretation of this problem. The one output of the circuit indicates whether the magnitude of the two numbers is equal or not.

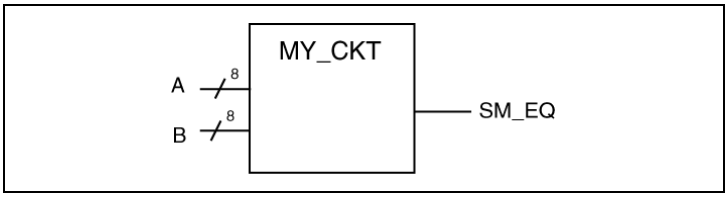


Figure 14.4: A block box diagram that supports the description of this problem.

The problem practically does itself. The problem statement mentions the word "compares", which cries out for the fact that we need a comparator in the solution. Since binary numbers in SM form use all but the sign-bit to represent the magnitude, all we need to do is compare the magnitude portion of the numbers; we do this by feeding just those inputs into a 7-bit comparator. Figure 14.5 shows the final solution to this problem.

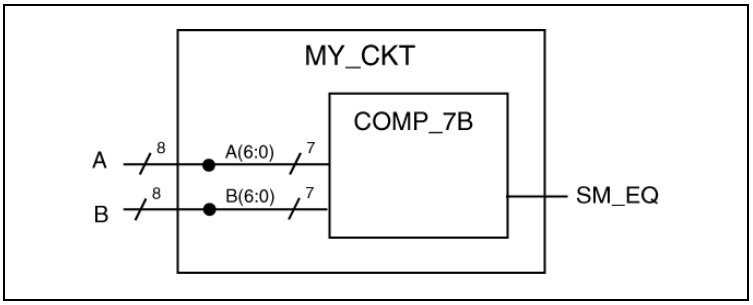


Figure 14.5: The final solution for this example.

One interesting thing worth noting in Figure 14.5 is the fact that we sort of "made up" our own terminology for this problem. Note that we put a connection dot on the bundle in an effort to indicate that we are doing something to the bundle. Next, we changed the effective width of the bundle and indicated in an arbitrary, but hopefully clear manner that only the seven lower-order bits (the magnitude bits) of the 8-bit bundle are being inputted to the comparator. Once again, this approach is arbitrary. Whenever you do something "different", you should make sure you adequately document it. Having a healthy explanation of what you're doing and writing it down is always a great approach.

Chapter Summary

- Signed binary numbers typically use a sign-bit to indicate the sign (negative or positive) of a given number. Signed binary numbers commonly use one of three representations: sign magnitude (SM), Diminished Radix Complement (DRC), or Radix Complement (RC).

- Each of the methods used to represent binary numbers have their own ranges of values that can be represented by those methods.

- Binary addition and subtraction has special meaning in the context of signed binary number representations. One of the key concerns when performing binary arithmetic operations is whether the result is valid or not. The validity of the result is based on the range of values that can be represented by a given set of bits.

- Binary subtraction is often done by using addition. This technique is referred to as the *indirect subtraction by addition* method. The accepted advantage of this approach is that the hardware used for addition can also be used for subtraction (after adding hardware that implements changing the sign of the hardware).

Chapter Exercises

1) Explain why adding two numbers of a different sign will always result in a valid number in terms of fixed hardware widths.

2) Explain the difference between the concept of overflow/underflow and the concept of carry-out.

3) Explain why "underflow" is sometimes classified as "overflow".

4) Complete the following table:

# bits	unsigned binary range	signed binary range (RC)
4		
6		
8		
10		
11		
12		
14		
15		
16		

5) Complete the following mathematical operations on the unsigned binary numbers. Indicate which results are valid based on the given number range.

a) 001100 + 000011
b) 001110 + 000111
c) 100101 + 101010
d) 001000 + 111100
e) 000100 + 101111

6) Complete the following mathematical operations on the unsigned binary numbers. Indicate which results are valid based on the given number range.

a) 001100 - 000111
b) 100101 - 001000
c) 111010 - 111100
d) 010001 - 011011
e) 010010 – 000110

7) Complete the following mathematical operations on the unsigned binary numbers. Indicate which results are valid based on the given number range.

a) 01001010 + 00010000
b) 11110000 + 00010001
c) 11100100 + 00100101
d) 01000000 + 01110000
e) 01001000 + 01111111

8) Complete the following mathematical operations on the unsigned binary numbers. Indicate which results are valid based on the given number range.

a) 01000001 - 00111100
b) 11000000 - 01001110
c) 00100101 - 10001110
d) 10000001 - 11000010
e) 11010011 - 11111100

9) Complete the following mathematical operations on the signed binary numbers (RC representation). Indicate which results are valid based on the given number range.

a) 00011 + 00111
b) 01110 + 00011
c) 01001 + 00100
d) 01010 + 00111
e) 01011 + 01001

f) 00011 - 00111
g) 01110 - 00011
h) 01001 - 00100

i) 00110 - 10100
j) 00111 - 11100
k) 01010 - 11000
l) 01010 - 11110
m) 01110 - 11001

10) Complete the following mathematical operations on the signed binary numbers (RC representation). Indicate which results are valid based on the given number range.

a) 10111 + 01000
b) 11001 + 01111
c) 11101 + 00100

d) 11010 - 01010
e) 11101 - 00100
f) 11010 - 01110
g) 10100 - 01110
h) 11111 - 01001

i) 10111 - 10111
j) 11101 - 11010
k) 11000 - 11110

11) Which of the following two signed binary (SB) number are greater? Assume the numbers are given in radix complement (RC) form. 1110 1110 0000 0010

12) Which of the following two signed binary (SB) number has a larger magnitude? Assume the numbers are given in radix complement (RC) form. 1110 1110 0001 0011

13) Which of the following three SB numbers has the largest magnitude?
a) = 1110 0001 (SM)
b) = 1001 1101 (DRC)
c) = 1001 1100 (RC)

14) The three numbers below are listed in hex but they represent 8-bit signed binary numbers in the given formats. Which of the three numbers is the most negative?
a) = B4 (SM)
b) = CC (DRC)
c) = D1 (RC)

15) Write the decimal equivalents of the number $1011\ 1100_2$ if the number is in SM, DRC and RC forms.

Chapter Design Problems

1) Design a circuit the changes the sign of an 8-bit signed binary number in sign magnitude form.

2) Design a circuit the changes the sign of an 8-bit signed binary number in diminished radix complement form.

3) Design a circuit that changes the sign of an 8-bit signed binary number in radix complement form.

4) Design a circuit generates the absolute value of an 8-bit signed binary number in sign magnitude form.

5) Design a circuit that compares the magnitude of two signed binary numbers in diminished radix form

15 Chapter Fifteen

(Bryan Mealy 2012 ©)

15.1 Chapter Overview

Digital circuits can be designed using many different approaches. In an attempt to present material on digital design, we attempted to classify these design approaches in to one of three methods. In reality, if you one day find yourself "doing digital design", you probably won't be thinking to yourself that, hey, I'm using the (file in the blank) approach to digital design; you're just going to do it. However, we're still leaning digital design so we'll continue with our attempts to classify what we're doing.

Up until now, we've used either brute force design (BFD) or iterative modular design (IMD) to design our digital circuits. This chapter outlines our final approach: Modular Design, or MD. In truth, IMD is actually a special case of MD. It's the most powerful approach to digital design, though you won't see a lot of that power in this chapter due to the fact that we've still not introduced some of the extremely important standard digital devices as of yet. This chapter presents MD in the context of the digital devices we've discussed in previous chapters, which as you'll find out as you read more chapters, is somewhat limited.

In the end, you'll start getting the feel for the notion that digital design means different things to different people; what I'm presenting here is my version of digital design that I've done my best to put into a learning context. I encourage you to form your own versions and take your digital designs techniques into the direction you want to go it. In other words, this chapter sums up what digital design is to me; I encourage everyone to figure out their own approach to digital design and run with it.

Main Chapter Topics

- **MODULAR DESIGN:** This chapter presents the basics of Modular Design (MD) starting with an overview and ending with design examples that are best solved using the MD approach.

Why This Chapter is Important

This chapter is important because modular design approach is the most powerful of all digital design approaches. It is hierarchical in nature and thus works well with VHDL structural modeling techniques. Most importantly, modular design supports the understanding of complex digital designs by directly supporting the hierarchical designs.

15.2 The Big Digital Design Overview

Now that you've learned about binary arithmetic circuits (previous chapters), you're to the point where you can start doing some mildly interesting designs. Because your knowledge and abilities has increased, it's time to introduce a more powerful design approach. However, before we do this, it seems worthy to put the new design approach into a context of what we already know.

There are three approaches to digital design; any possible digital design you do necessarily fits into one of these approaches[160]. The list below highlights the three design approaches and includes some boring explanation as well. Table 15.1 represents an even more pointless piece of drivel.

1) Brute Force Design (BFD): Also known as iterative design, this was the first design approach we worked with and was based on assigning outputs to every possible input combination via a truth table. The fast growing size of truth tables limited this approach (as the number of inputs increased, the truth tables became unwieldy).

2) Iterative Modular Design (IMD): This was the second approach to design we worked with. Most appropriately, IMD would be included as a subset of modular design, but we're opting to call it a design approach all its own. This design approach allowed us to bypass the truth table approach of BFD and enabled us to create mildly complex circuits such as the ripple carry adder (RCA), the comparator, and parity circuits.

3) Modular Design (MD): This approach is similar to what we did in the first chapter. Do you remember the black box design approach? This was actually a good example of MD. In the approach presented in the early chapter, we were most interested in drawing bunches of black boxes to model our designs. We also drew boxes within boxes within boxes which we labeled as hierarchical design[161]. Recall that we've been claiming all along that hierarchical design is massively powerful; now we'll state that modular design is also massively powerful.

Design Approach	Pros	Cons
Brute Force Design	Really straight forward	Limited by truth table size
Iterative Modular Design	Straight forward	Not applicable to all designs
Modular Design	Massively powerful	Requires a working brain[162]

Table 15.1: Matrix explaining why Modular Design can save the world.

In case you have not gotten the point yet, here it is: modern digital design consists primarily of Modular Design. You do modular design by plopping down black boxes and connecting them up in intelligent ways. The black box diagrams are of course a form of modeling. The black box models convey various levels of information regarding the digital circuit, but most importantly, someone can take your model and actually convert it to a working circuit.

The general approach to modular-based design is to collect a bag full of standard digital modules and assemble those modules in such a way as to solve digital design problems. You may not realize it, but you already have started collecting this bag of digital modules: the half-adder, the full-adder, and the ripple carry adder (RCA), the comparator, and the parity circuits are currently in that bag. The problem now is that your bag does not present a lot to work with. This chapter aims to show you new and

[160] But let me know if you think of another approach, I'll for sure add it to this list.
[161] Thus, hierarchical design is a form of modular design.
[162] Thus, you will find not viable digital designers in an academic administrative setting.

exciting ways to work with what you already have in your digital bag of tricks[163]. We'll add to the bag in later chapters.

The overall approach of MD is to abstract circuits to a higher level in order to increase our efficiency in the digital design process. The potential problem with designing at high levels is that the designer can make too many assumptions in the design process and not properly convey these assumptions to other entities. Because MD is highly model based, the issue here is to make sure the entity that is going to read your design[164], can fully comprehend what you're trying to convey with your design. That being said, there are a few rules you should always follow when doing the MD thang; and these are really not rules (they're more like guidelines, whatever those are). Here are MD modeling rules:

- **Be clear and concise:** A messy black box model or circuit diagram is a tragedy that hinders the efficient transfer of information. Strongly consider using a ruler if you're modeling by hand.

- **Don't make assumptions:** You don't really know who is going to read your models or anything about their knowledge base. Therefore, any assumption you make could quickly confound your design if the reader of your design does not know and/or understand the assumptions you made in your design.

- **Label everything:** Make sure the reader of your model does not need to make any assumptions about anything. This is a special case of not making any assumptions.

- **Provide a definition for all black boxes:** Black box modeling facilitates modern digital design. Every box you use in your model should either be clearly defined somewhere (such as at another level) or be a standard digital "box". There are many standard digital "boxes" out there. If you call out one of these boxes in your models, everyone will know what you're talking about and there is no need to define it at a lower level. At this point, we've seen relatively few of these boxes: HAs, FAs, RCAs, comparators, and the various gates we've talked about. The catch here is that you must use these boxes in the exact way there were defined; if you don't, people will not know what you're modeling. Table 15.2 shows that this point is best presented in a visual manner.

[163] They're not really tricks, they're actually standard digital circuits.
[164] It could be a person or a computer (such as the VHDL synthesizer).

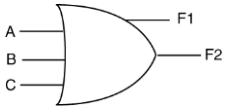
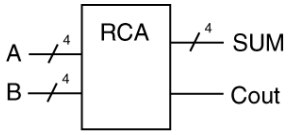
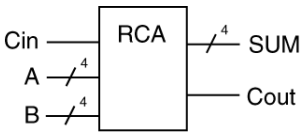
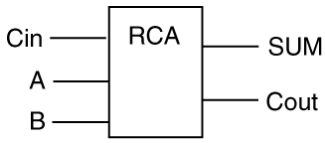
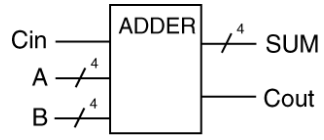
Model	Comment
	This model sort of looks like a 3-input OR gate, but having two outputs makes it non-standard. Being non-standard, it's a mystery how any of the outputs are assigned. This is a bad model. To make it valid would required that it be defined somewhere so we all know what it is.
	This is a true digital box. Since we know what an RCA is, and the inputs and outputs of the box labeled RCA match what we know about RCAs, we know exactly how it works and how to build it. This is a valid model and there is no need to define it anywhere else in your model. This RCA does not have a listed carry in input, but that is no big deal; the reader will quickly figure out that you must have used a HA in the least significant bit position for your RCA. This is a valid model.
	This is also a true digital box. If you replace the HA in a RCA with a FA, you'll have the extra carry-in input as is listed in this model. Having this input is very handy in various digital design applications. This is a valid model.
	This is labeled RCA but since we know RCAs to have multiple inputs (bundles) for the addend and augend, we're left scratching our heads. You could assume it's a RCA but you could be wrong. The SUM output has the same issue. This is an invalid model.
	This has all the correct inputs for an RCA, but since it has the ADDER label, we can't assume we know exactly what this box is doing. This is an invalid model. You could make this model valid by providing a definition for the ADDER somewhere in your design.

Table 15.2: Some good and bad example of standard digital black boxes.

This chapter has one good trait that was purportedly lacking in other chapters: More example problems and less bloviation. The explanations of MD are contained within the example problems; thus, there is less of the verbage you've grown tire of in previous chapters.

Not that rules are good things, but they can help out when first embarking on the MD approach to digital design. It's happy to note here an excellent quality regarding MD: the problems have a strange way of doing themselves based primarily on the problem description. We've outlined this approach in Figure 15.1 below and apply this approach in the design examples that follow.

1. **Read the problem**: Yes, a great start.

2. **Draw a high-level black box diagram that shows the design's interface (inputs and outputs of the high-level black box):** This is not always an easy step based on the way the problem is stated as sometimes the important information is buried in the problem description[165]. The benefit of completing this step is that it inevitably helps you understand the complete problem.

3. **Fill in the sub-design entities listed by the original design:** This can be an easy step because major clues are typically provided by the problem. For example, if a problem says something like "add" or "sum", they you know right away to include an RCA box in your design. As we learn about other standard digital devices, this becomes even more obvious.

4. **Connect the Lower-level Design Entities:** The result of the previous step is to have a bunch of black boxes in your design; this step entails connecting those black boxes to something intelligent.

5. **Provide Adequate Models for Any Non-Standard Black Boxes used in the Design:** The hope is that you can use as many standard digital design boxes as possible in your design. However, don't hesitate to create new boxes with "special" functionality that helps you solve the problem at hand. Keep in mind that these new black boxes can be boxes that contain other black boxes as a true hierarchical design approach. The overall requirement is that you list and describe every aspect of your design at the same time as keeping your design as simple as possible; the hierarchical approach supports simplicity and understanding of your design.

6. **Check your final diagram for the following**: Somewhat self-explanatory...

    a. Make sure the highest-level black box is labeled.
    b. Make sure all inputs to lower-level design entities (black boxes) are connected to something (either other signals, '1', or '0').
    c. Make sure all signals are adequately labeled.
    d. Make sure all bundle widths are labeled.
    e. Make sure all lower-level design entities (black boxes) are labeled
    f. Make sure all labels are self-commenting in nature.

Figure 15.1: The desired approach to solving modular design problems.

The final comment: you need to be creative and clever with your diagram-type solutions. You will inevitably run into situations that you have not seen listed any example you've studied. The best approach to use in this situation is to make do what you need to do in the simplest way possible. If your approach is not patently obvious, provide adequate annotation and/or description somewhere in your diagram.

Finally, the nice thing about VHDL is that it supports all three of the stated design approaches. Although we could consider VHDL-based designs a design approach on its own, we're opting to view VHDL as a tool to implement one of the three standard design approaches. In other words, VHDL supports any type of standard digital design approach; we do not consider VHDL a design approach of

[165] It's called a crappy problem description. You'll find a lot of them on exams.

its own. Moreover, when you need to solve a digital design problem, you start with a block box diagram; only total wankers start coding VHDL before they complete a black box diagram.

Example 15-1: RC Sign Changer

Design a circuit that changes the sign of an 8-bit signed binary number in radix complement form. Provide your solution in the form of a black box model. Minimize your use of hardware in your final model. If you use something other than a standard digital circuit, make sure you adequately provide an adequate description.

Solution: This is a really important and instructive circuit out in digital design-land. At this point in your design career, you may be wondering what to do. Keep in mind that the first step is always to draw a black box diagram of your solution. Note the nicely labeled model shown in Figure 15.2.

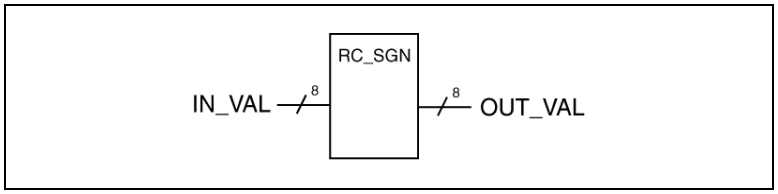


Figure 15.2: Black box diagram for RC Sign Changer.

The next step is to gather in what you know about changing the sign of binary numbers in RC notation. The standard method we learned was the visual algorithm method of starting at the right-most bit in the number and looking for the first '1' etc. Although this worked great on paper, it does not work for digital hardware[166]. What we need to do is use the other approach to changing the sign which was to take the 1's complement and add '1'. Taking the 1's complement of the input only requires an inverter for each individual bit input to the circuit. Adding '1' can be done in several ways (though we'll only use one way for this problem).

Without too much hoopla, Figure 15.3 shows the final solution for this example. Take a look at it then read and understand the comments that follow: there are a lot of important digital practices taking place in this problem that you need to know.

[166] Actually, you could use VHDL to model a circuit using this algorithm. The resulting circuit would be valid but it would be less optimal than using a more intelligent approach.

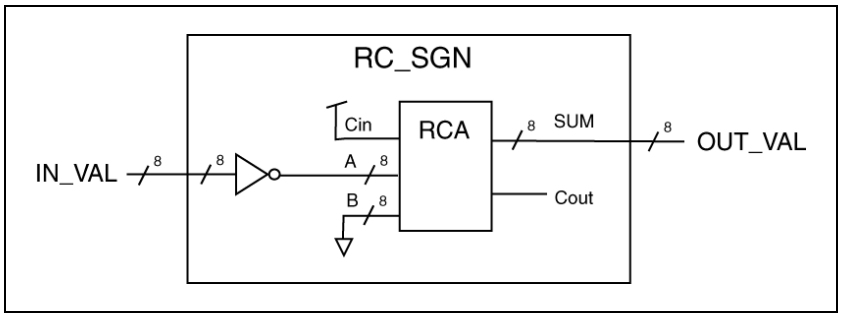


Figure 15.3: Black box diagram for RC Sign Changer.

- The box in Figure 15.3 is consistent with the box in Figure 15.2: the inputs and outputs match in both bundle size and name.

- The bundles notation in Figure 15.3 appears on both the inside of the RC_SGN box as well as the outside. Probably either listing would have been fine, but including both is fine also. If you were going to include in only one place, it would be on the outside of the box, which would make the diagram in Figure 15.3 match the diagram in Figure 15.2.

- It appears that the 8-bit bundle uses a single inverter. This is actually an accepted shorthand notation for indicating the inversion of every signal in the bundle. We could have drawn eight inverters but it would have messed up our diagram. When you use this notation, the digital world understands that there are really eight appropriately connected inverters.

- The Cin signal has a funny thing connected to it; the funny thing indicates that the Cin input to the RCA is connected to '1'. You see this notation often; sometimes you also see a "Vcc" or a "Vdd" which indicates the signal is connected to the higher voltage rail in the circuit which is generally considered to be a logical '1'.

- The B signal has a funny thing connected to it. This funny thing indicates that the B input of the RCA is connected to "ground" or a logical'0'. You also see this notation often in digital design so get used to it. Once again, this notation signifies that each of the eight individual signals in the bundle is connected to ground.

- The Cout signal of the RCA is unconnected. This is no big deal, as your design is not using it. Although you always need to connect your inputs to something, generally speaking, the outputs don't need to be connected if you're not using them. It's a good choice to include unconnected outputs but certainly not a requirement.

- Lastly, you may be wondering why this actually circuit works. The RCA as drawn in this problem uses a FA for the LSB. This means that the total equation for the RCA is: SUM = A + B + Cin. The way the circuit is connected in this problem is that the B value is always zero, the A signal is always inverted, and Cin is always '1'. The final implemented equation is therefore: SUM = (not A) + 1. This equation therefore implements a viable approach to a 2's complement, which is the 1's complement plus one.

Example 15-2: Special RC Addition Circuit

Design a circuit that adds '2' to an 8-bit signed binary number in radix complement form. This circuit has an output signal VALID that is '1' when the addition operation is valid. Minimize your use of hardware in your final model. If you use something other than a standard digital circuit, make sure you adequately provide an adequate description.

Solution: This is another important and instructive circuit in that it's does not seem trivial at first, but does in fact have a relatively simple solution. The solution starts with drawing a black box diagram of your solution. Figure 15.4 shows the nicely labeled black box model. Note that you could have drawn this diagram even if you knew nothing else about how to proceed with the solution.

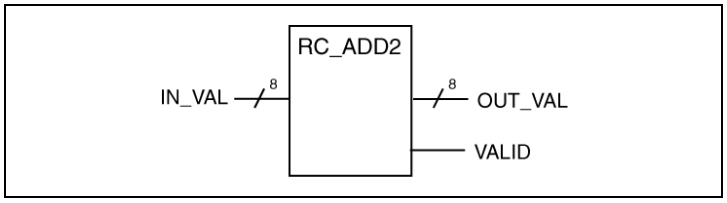


Figure 15.4: Black box model for solution.

The next step is to start speculating about what goes on the inside of the box. To do this, always look back to the original problem for clues. The first clue is that you'll be adding a number to another number which means that we're going to need an adder. The only adder we know about is a RCA so that's that well use. The next thing we'll need is some type of circuitry indication when the solution is valid or not. This is known as "control" circuitry. OK… let's put it down; check out Figure 15.5.

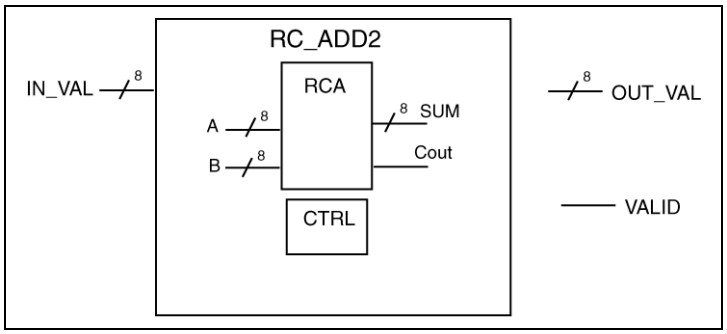


Figure 15.5: The next step in the solution.

Here are some interesting to note about Figure 15.5 that will help you move toward the solution. Figure 15.6 shows the result of listing all of these interesting things.

- The RCA is going to add two things: the IN_VAL and the number "00000010" (which is 2 in binary). Therefore, we can connect IN_VAL to one of the RCA operands and "hardwire" a binary "2" to the other operand. We've indicated this in the diagram by listing "00000010" near the bundle in question.

- The output of this circuit is going to be the result of the sum so we can connect the output of the RCA to the OUT_VAL signal.

- The CTRL circuit is going to indicate if the operation was valid or not. Although the inputs to the CTRL box are still unknown, we know the output is going to be the VALID signal.

- The big question is how are we going know if the addition operation is valid or not? The answer lies in the fact that since we're adding two signed binary numbers in RC form, the answer will only be valid if the sign of the result is the same as the sign of the two input operands. Therefore, the CTRL box will need three inputs: the sign bits of the two RCA operands and the sign-bit of the SUM operand.

- We will not need the Cout signal for our approach to this solution so we can leave it unconnected since it's an output.

The next step in the solution is to design the interior of the CTRL box. The best way to do with is with a truth table[167]. The key to filling out this table is to note that the result of the binary addition is only going to invalid when the sign bits of the operands are the same and the sign bit of the result is different. Figure 15.7 shows the resulting truth table. Note that because the sign-bit of the A input will always be '0', table entries were A=1 are listed as don't cares.

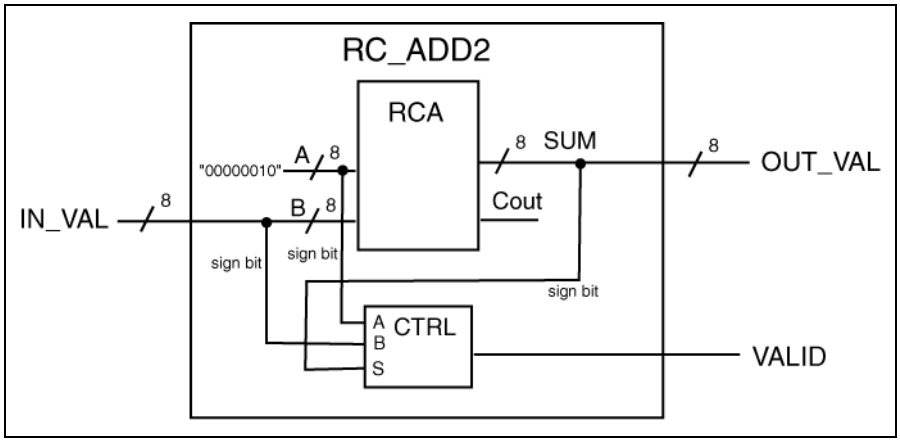


Figure 15.6: The next step in the solution.

[167] It may not be the best way, but it's a valid way since there are only three inputs.

A	B	S	VALID
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	-
1	0	1	-
1	1	0	-
1	1	1	-

Figure 15.7: The truth table modeling the CTRL box.

The equation that describes the truth table of Figure 15.7 is: $F = B + \overline{S}$. Note that we did not need the A sign-bit input after all. In the end, the fact that the sign bit of the A input to the RCA does not affect the problem makes sense, it you think about it for a few minutes.

Figure 15.8 and Figure 15.9 show the final solution. Note that there are two parts to the solution; each of these parts represents a different level of the design. Figure 15.8 represents the higher-level portion of the solution while Figure 15.9 represents the lower-level of the solution

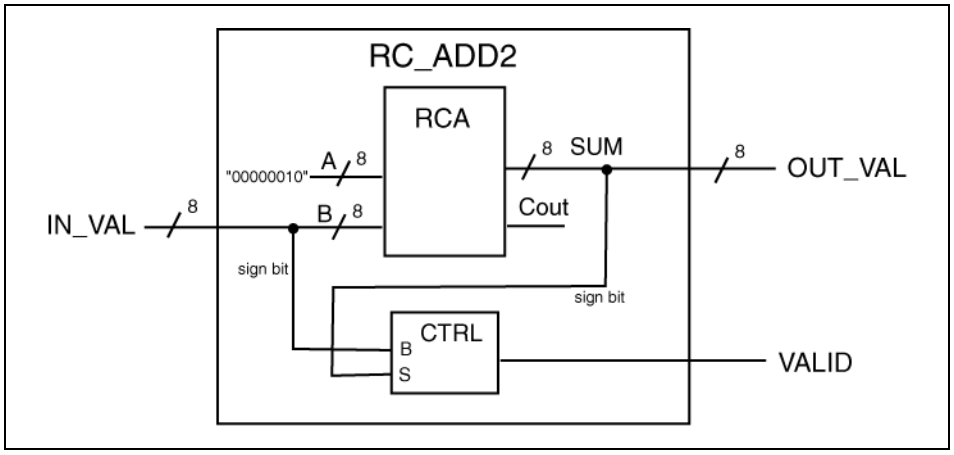


Figure 15.8: The final solution to this problem.

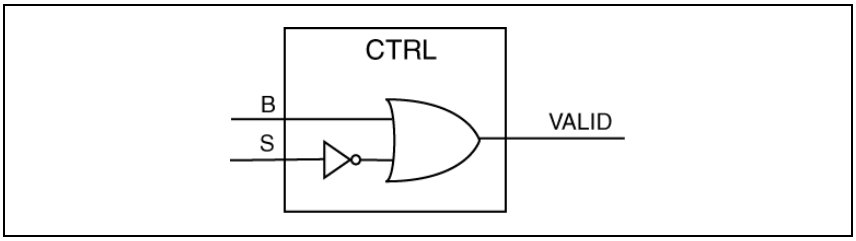


Figure 15.9: The other part of the final solution.

A few comments… yes, this is a true hierarchical design. It did not have to a hierarchical design; we could have placed the OR gate of Figure 15.9 into the black box diagram of Figure 15.8. However, this approach is more structured and thus, more clear. Also note that we never really had an idea of the final solution when we started the problem; we instead just started working towards a solution starting with what little we knew about the problem. Using this approach, we eventually ended up at the solution. This is an important concept because you'll not always have a good idea as to how the solution will appear, but you'll have a direction to go in. As you traverse that direction, you'll pick up clues to the solution. You may be well on your way or you may realize that your approach needs tossing.

For the record, another approach to this problem would be to utilize the carry-out of the RCA as follows. The only instance the 8-bit output is not valid is when the value of two is added to a large positive number. If you add two to a negative number, the magnitude always decreases. However, if you added two to a large positive number, you could exceed the range associated with the 8-bit result. In this case, the carry-out of the RCA will indicate that with a '1'.

Example 15-3: Three-Value 10-Bit Comparator

Design a circuit that compares three 10-bit values. If all three 10-bit values are equivalent, the EQ3 output of the circuit will be a '1', otherwise it will be a '0'. Use only standard comparators in this design. Use any support logic you may require but minimize the amount of hardware used in this circuit. Use the modular design approach and provide both a block-level diagram for your solution.

Solution: The main constraint in this problem is the required use of standard comparators in the solution. Other than that, you should stand back for a minute and view this problem from a wider perspective. If someone asked you to determine if three numbers were equivalent, what would you do? It's an old math thang to say, "if A = B and B = C then A = C". You mission is then to translate that probable intuitiveness to digital hardware. A good start, as always, is drawing a black box diagram as shown in Figure 15.10.

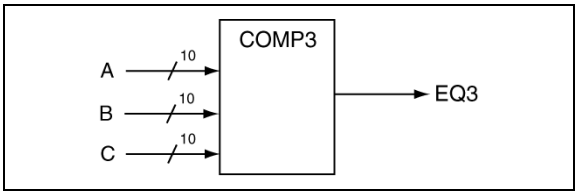


Figure 15.10: Black box diagram for Example 15-3 solution.

From here, you need to reconsider the standard comparator constraint on this problem. Since a standard comparator only compares two numbers, you'll need two comparators to determine if all three inputs are equivalent. From this point in this problem, note that the problem directly states the required extra logic in the quoted statement in the previous paragraph: the "and" indicates that this solution requires an AND gate. Figure 15.11 shows the final block diagram for this problem. Note that the block diagram directly implements the quoted statement in the previous paragraph.

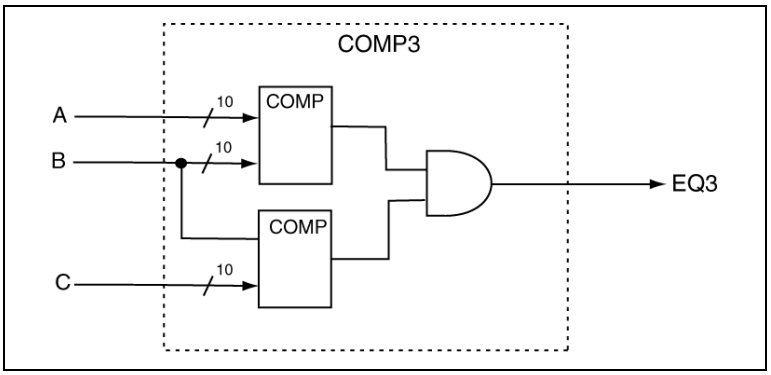


Figure 15.11: The final circuit for Example 15-3.

Example 15-4: 8-Bit Adder/Subtractor

Design a circuit that acts as both an adder and subtractor. This circuit has a control input **SUB** and two eight-bit inputs **A** and **B**. When the **SUB** input is high, the 8-bit circuit output indicates the result of **B** subtracted from input **A**. Otherwise, the output of the circuit indicates of addition of the **A** and **B**. Assume that you have no reason to worry about the carry-out from the adder. Use any support logic you may require but minimize the amount of hardware used in this circuit. Use the modular design approach and provide both a block-level diagram for your solution.

Solution: The point behind this problem is gathering up all of the *standard* circuits you've learned about up until now. These include half adders, full adders, ripple carry adders (RCAs), and comparators. You'll also need to recall all the information you learned regarding binary number and particularly signed binary numbers. In addition, as always, the first step in this solution is drawing a black-box diagram of the circuit; Figure 15.12 shows the black box diagram for this solution.

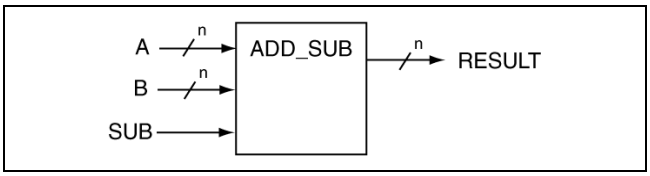


Figure 15.12: Black-box diagram of the Adder/Subtractor circuit.

Although we took a thorough approach to designing the RCA, we're going to try not to go that long and arduous design route with this problem and its requirement of doing a subtraction operation. What is going to save us is the fact we'll remember that subtraction in binary can be done by first multiplying the appropriate operand by -1 (1's complement) and then adding the result to the other operand. For this problem, that means doing a two's complement on one of the operands. So… let's first review some of the properties of the RCA.

The approach we took to designing the RCA was to have the least significant bit location as a half adder and all of the other elements as full adders. For this problem, we're going to use full adders for all of the

adder elements. Figure 15.13(a) show the RCA with a HA as the MSB and Figure 15.13(b) shows RCA with a FA in the MSB position. As you can see by comparing these circuits, they differ by the inclusion of the carry-in (Cin) for the RCA shown in Figure 15.13(b).

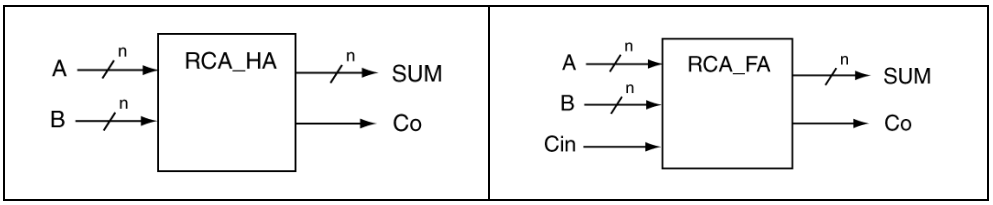


Figure 15.13: A diagram of the HA-based (a) and FA-based (b) ripple carry adder.

The key to completing this problem is noting that we'll use the *indirect subtraction by addition* approach. In other words, the subtraction is going to be done by first taking the two's complement of the operand that is being subtracted and adding the result to the other operand. As you no doubt know, the two's complement is obtained by taking a 1's complement (complementing all the bits) and adding 1. The cool thing to notice here is that Equation 15-1 shows the final result of the entire RCA in Figure 15.13(b). In other words, the SUM output of the circuit shown in Figure 15.13(b) represents the result of the addition operations shown in Equation 15-1.

$$SUM = A + B + Cin$$

Equation 15-1: What exactly the RCA is adding.

The SUB input to the circuit has two functions: 1) to select the complemented or non-complemented operand to one of the RCA's inputs, and 2) to select a '1' for the Cin input on the RCA_FA. The final circuit is thus going to look like something shown in Figure 15.14. Another way to look at this is that the value of the SUB signal is always included in the addition operation of the RCA_FA. If the SUB input is a '0', it will have no effect on the final result as the addition of zero changes nothing.

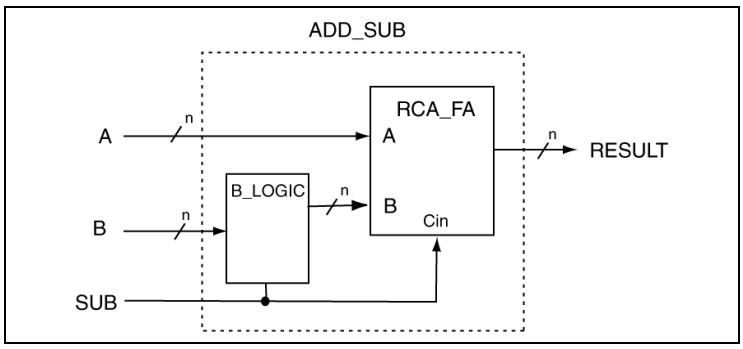


Figure 15.14: The final circuit.

The last thing you need to do here is define what is in the B_LOGIC block shown in Figure 15.14. There is actually a well-known approach to this problem. The approach is to notice that signal B sometimes needs to be inverted before it is sent to RCA_FA and sometimes it does not. When SUB is a

'1', the ADD_SUB module performs an A – B operation which means we want to invert the B signal and add '1' to the RCA_FA module via the Cin input.

The Cin input to the RCA_FA already works correctly for both addition and subtraction (see Table 15.3 for some extra details). For the B_LOGIC, we need to invert individual signals in B before they are sent to the RCA_FA when SUB is a '1'. The most straightforward way to do this is to use known properties of the XOR gate; specifically, when one input to an XOR gate is '1', the output of gate is an inversion of the other input. Similarly, when one input to a XOR gate is a '0', the other input effectively passes through the XOR gate output. Figure 15.15 shows the final circuit for the B_LOGIC block.

Figure 15.15 show a few interesting features worth noting. First, signal B is decomposed into its parts on the diagram with the assumption that B(7) is the MSB while B(0) is the MSB. The output is reassembled from its parts back into a bundle. Since the bundle has no name on the higher-level diagram of Figure 15.14, it was not named in Figure 15.15 either.

SUB value	RCA_FA operation	Comment
'0'	$SUB = A + B + 0$	A + B
'1'	$SUB = A + \overline{B} + 1$	A - B

Table 15.3: Tabular view of RCA_FA operation.

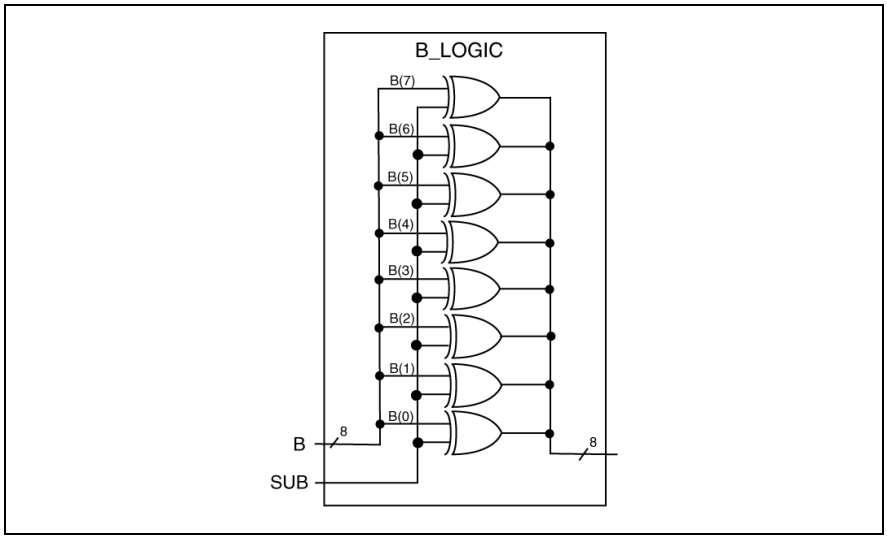


Figure 15.15: The schematic for the B_LOGIC block.

Lastly, this is a typical and well-known solution to this type of problem. Note that the schematic of Figure 15.15 required a long time to draw. Figure 15.16 shows a better approach to the final solution of this problem; this solution is better in the sense that it was easier to draw but is actually the same circuit as the previous solution. Note that Figure 15.16 uses a special shorthand notation. Although XOR gates only have two inputs, Figure 15.16 seems to indicate that the XOR gate can accept a bundle input. In actuality, the special XOR gate shown in Figure 15.16 is the same circuit as the B_LOGIC block shown in Figure 15.15.

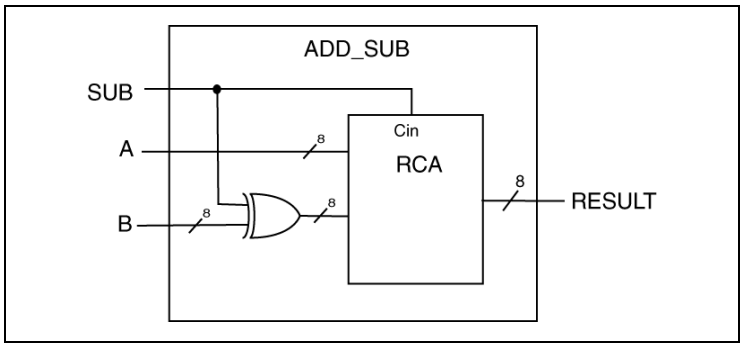
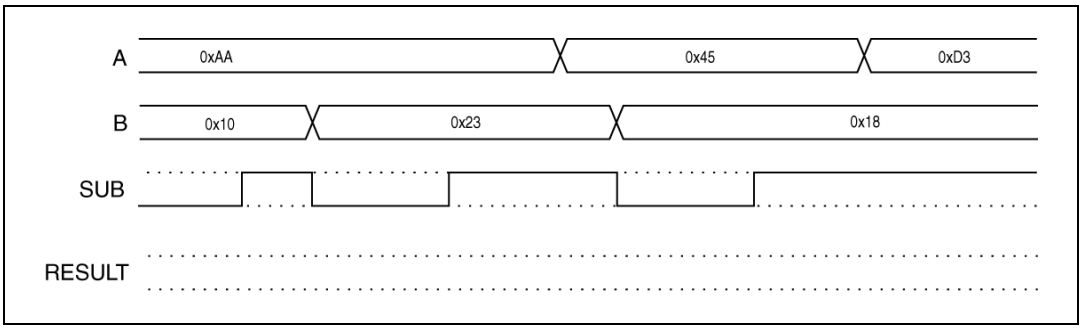


Figure 15.16: An alternate and popular approach to the final circuit.

Example 15-5: Timing Diagrams

Based on the solution to Example 15-4, complete the following timing diagram.



Solution: The problem states the value on signal B will either be subtracted from or added to signal A based on the value of signal B. Figure 15.17 shows the final solution to this problem keeping in mind that when SUB is a '0', the RESULT signal represents an addition of signal A & B.

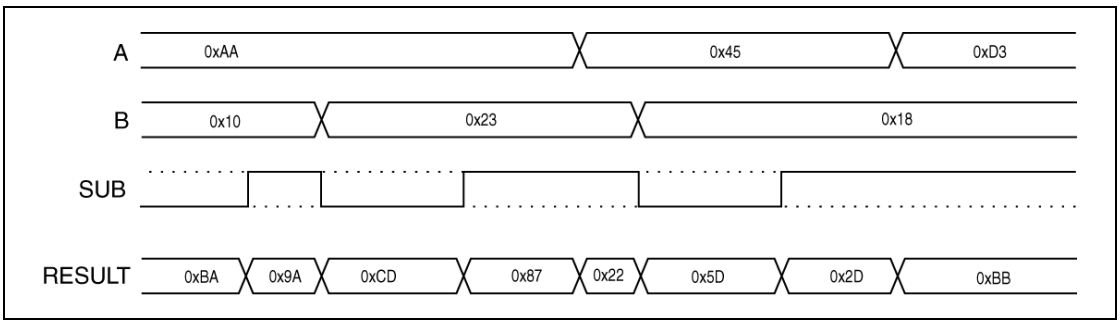


Figure 15.17: The solution to Example 15-5.

Example 15-6: Special Arithmetic Circuit

Design a circuit that has three 8-bit inputs **A**, **B**, and **C**. The single output of the circuit indicates whether the sum of **A** and **B** is equal to the sum of **B** and **C**. For this problem, assume that the addition of the two input values will never cause a carry out. Use the modular design approach for this problem and provide a circuit diagram that solves this problem. Minimize your use of hardware for this design.

Solution: Once again, you need think on a higher digital level in order to solve this problem. With the eight inputs, is would be impossible to do this any other way. So… the first step is drawing a block diagram of the final circuit as shown in Figure 15.18.

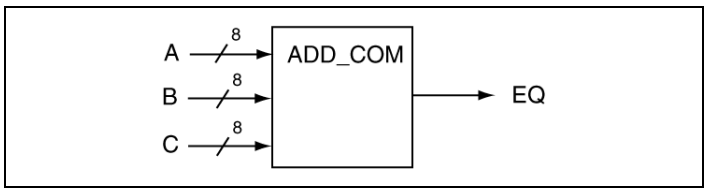


Figure 15.18: Block diagram of the final circuit.

The key to step one is decomposing the problem. From the problem statement, you can see that the final circuit is going to require two RCAs in order to perform the two required addition operations (**A** + **B** and **B** + **C**). The second clue given in the problem statement is that something needs to be compared. In this case, you'll need to check whether the results of the two addition operations are equivalent. And that's it. For this problem, you'll need two RCAs and one comparator. Note that the problem statement itself provided many of these clues.

The required connections of the three modules are based upon the requirements of the final circuit. Figure 15.19 shows the final result of Step 2) and the entire problem. Note that in this solution, the carry-in and carry-outs of the RCAs are not specified which implies they are irrelevant to this solution. Also, we did not include carry-in inputs to the RCAs so we can assume that the RCA did not have a carry-in input or the carry-input was connected to '0' and thus did not affect the problem.

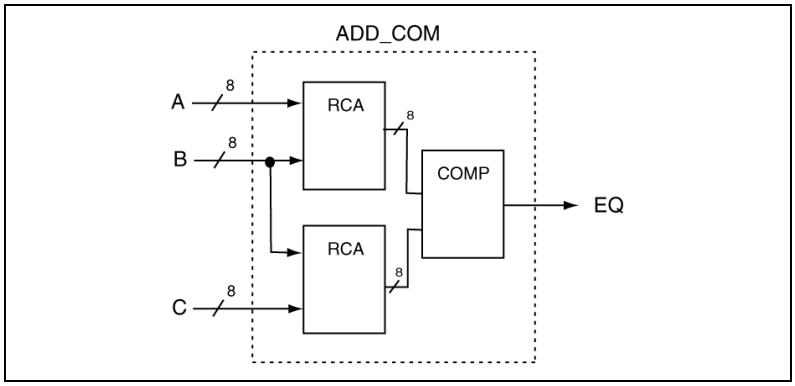
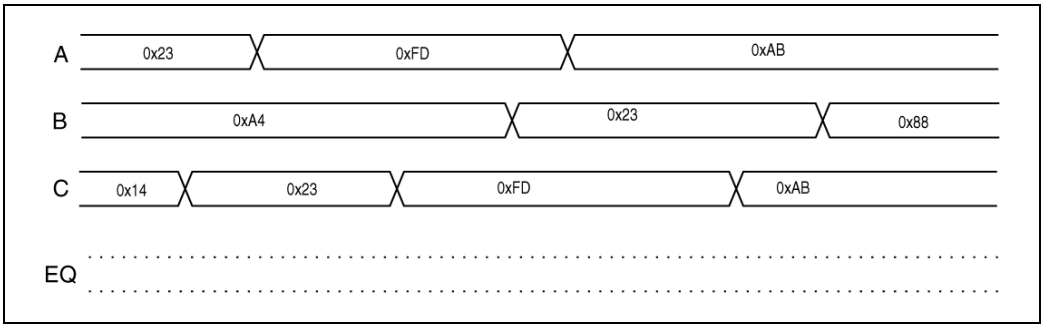


Figure 15.19: The final circuit solution for this problem.

Keep in mind that for this problem, you know how to create a RCA and a comparator (the module labeled COMP is an 8-bit comparator in this problem. The bit-width of these devices for this problem was not really a big deal; you can use iterative modular design or VHDL modeling to overcome the problems presented by the wider data paths.

Example 15-7: Special Arithmetic Circuit

Based on the solution to Example 15-6, complete the following timing diagram.



Solution: The problem states (after you read it a few times) that the EQ output is a '1' when ever A = C. Figure 15.20 shows the final solution to this example. Note that signal B does not affect the answer. Also, note that the carry-outs from the RCAs also do not affect the outputs as they do not change the value of the RCA sum outputs.

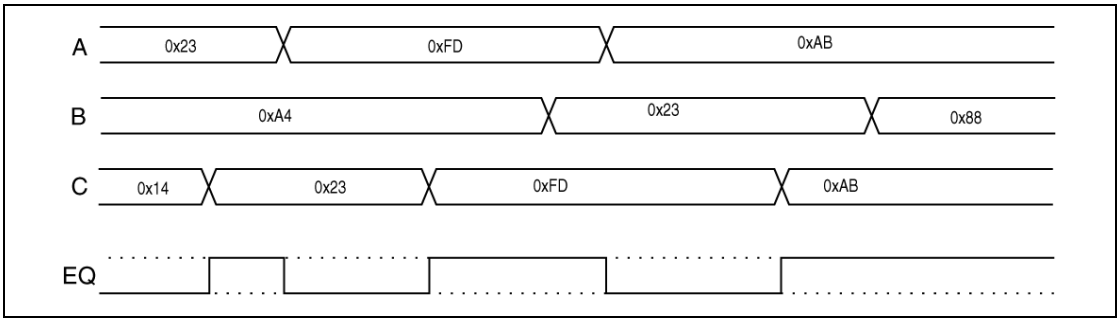


Figure 15.20: The solution to Example 15-7.

Chapter Summary

- There are three basic approaches to digital design 1) Brute Force Design (BFD), 2) Iterative Modular Design (IMD), and 3) Modular Design (MD). By far, Modular Design is the most powerful, particularly since hierarchical design is a form of MD.

- All black box diagrams should be a simple as possible. If you need to create some special notation for your solution, be sure to describe it fully.

- An overview of the approach to MD-type problems can be stated as:

    1. **Read the Problem**.
    2. **Draw a High-level Black-box Interface Diagram**.
    3. **Include the Lower-level Design Entities.**
    4. **Connect the Lower-level Design Entities.**
    5. **Provide Adequate Models for Any Non-Standard Modules.**
    6. **Check Your Final Diagram for All Important Details.**

Chapter Exercises

1) Why is IMD actually considered a subset of MD? Briefly but fully explain.

2) List several advantages to using a self-commenting style in your black box diagrams.

3) Describe how it is that MD is more powerful than BFD. Feel free to use 4-letter words in your answer.

Chapter Design Problems

1) Design a circuit the changes the sign of an 8-bit binary number in sign magnitude form. Your solution should in the form of a black box model. Minimize your use of hardware in your final model.

2) Design a circuit the changes the sign of an 8-bit binary number in diminished radix complement form. Your solution should in the form of a black box model. Minimize your use of hardware in your final model.

3) Design a circuit that provides the absolute value for an 8-bit signed binary number. Assume the number is in sign magnitude form. Your solution should in the form of a black box model. Minimize your use of hardware in your final model.

4) Design a circuit that provides the absolute value for an 8-bit signed binary number. Assume the number is in diminished radix complement form. Your solution should in the form of a black box model. Minimize your use of hardware in your final model.

5) Design a circuit that subtracts '3' from an 8-bit signed binary number. Assume the number is in radix complement form. This circuit has an output signal VALID that is '1' when the subtraction operation is valid. Your solution should in the form of a black box model. Minimize your use of hardware in your final model.

6) Design a circuit that multiplies an 8-bit signed binary number by two. Assume the number is in radix complement form. This circuit has an output signal VALID that is '1' when the operation is valid. Your solution should be in the form of a black box model. Minimize your use of hardware in your final model.

7) Design a circuit that multiplies an 8-bit signed binary number by three. Assume the number is in radix complement form. This circuit has an output signal VALID that is '1' when the operation is valid. Your solution should be provided in the form of a black box model. Minimize your use of hardware in your final model.

8) Design a circuit that translates an 8-bit number in signed magnitude form to an 8-bit number in diminished radix complement form. Provide your solution in the form of a black box model. Minimize your use of hardware in your final model.

9) Design a circuit that translates an 8-bit number in diminished radix complement form to an 8-bit number in signed magnitude form. Provide your solution in the form of a black box model. Minimize your use of hardware in your final model.

10) Design a circuit that translates an 8-bit binary number in radix complement form to an 8-bit number in diminished radix complement form. For this problem, assume the RC number will always be less than zero. Provide your solution in the form of a black box model. Minimize your use of hardware in your final model.

11) Design a circuit that adds five 10-bit unsigned binary numbers, **A**, **B**, **C**, **D**, and **E**. No matter what, the final sum should always be output, but this sum output is only a 10-bit number also. The catch is that this circuit has a "VALID" output indicates when the 10-bit output is a valid represents the actual sum of the five input values. You're only allowed to use 10-bit RCAs for this circuit. If you use anything other than a standard digital circuit, be sure to adequately describe that circuit, but do not use VHDL. Minimize your use of hardware in this design. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

12) Design a circuit that has two 8-bit inputs **A** and **B**, and one output. The output value is a '1' when the **A** input value is one greater than, equal to, or one less than the **B** input value; otherwise, the output is a '0'. Consider both inputs to be signed binary numbers in radix complement form. For this problem, assume both input values are always between 20_{10} and 120_{10}. If you use anything other than a standard digital circuit, be sure to adequately describe that circuit, but do not use VHDL. Minimize your use of hardware in this design. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

16 Chapter Sixteen

(Bryan Mealy 2012 ©)

16.1 Chapter Overview

The purpose of this VHDL presentation is to provide a guide to help develop the skills necessary to use VHDL in the context of modern introductory and intermediate level digital design courses. The skills presented with this and subsequent chapters allow budding digital designers to not only navigate modern digital design, but also give them the skills and confidence to continue with VHDL-based digital design and develop the skills required to solve more advanced digital design problems.

Main Chapter Topics

- **VHDL INTENT, PURPOSE, AND INTRODUCTION:** This chapter describes the motivation and justifications used by this text to introduce VHDL. The chapter briefly describes the primary uses of VHDL
- **VHDL DESIGN UNITS:** This chapter introduces the notion of the *entity* and the *architecture*, the basic VHDL design units.
- **VHDL MODELING INTRODUCTION:** This chapter introduces VHDL modeling in various example problems

Why This Chapter is Important

This chapter is important because it describes the principles behind VHDL. Modern digital design uses VHDL extensively to design and test digital circuits.

16.2 VDHL in Modern Digital Design

Although there are many online books and tutorials available dealing with VHDL, these sources are often troublesome for several reasons. First, much of the information regarding VHDL is either needlessly confusing or poorly written. Material with these characteristics is written from the standpoint of someone who is painfully intelligent or has forgotten that their audience may be seeing the material for the first time. Secondly, the common approach for most VHDL manuals is to introduce too much extraneous information and too many topics too early. It's much better to present most of this material later in the learning cycle. Material presented in this manner has a tendency to be confusing and easily forgotten if it is never applied. This chapter is to provide students with only what they need to know in order to get them quickly up and running in VHDL. As with all learning, once you obtained and applied some useful information, it is much easier to build on what you know as opposed to continually adding information that is not directly applicable to the subjects at hand.

Most modern introductory digital design texts suffer from similar drawbacks. One common problem with digital design texts is that they rarely integrate VHDL into the learning experience. It is blatantly obvious that in these texts, the authors had previously written the non-HDL portion of the text, and later tried to integrate the HDL concepts into existing material. Although these texts make for worthy reference material, they do not make for a good learning experience.

Another common problem with many digital design texts is that they attempt to simultaneously integrate the introduction of several HDLs (include VHDL, Verilog, and ABEL)[168]. This attribute leads to the now famous TMI syndrome (too much information) and serves no useful purpose. Lastly, worthy of comment is the fact that some texts include the HDL information only at the end of the text. The worst digital design textbook ever written uses this format[169].

The intent of this introduction to VHDL is to present topics in the context of the average student who has some knowledge of digital logic and has some skills with algorithmic programming languages such as Java or C. The information presented in this text focuses on a base knowledge of the approach and function of VHDL. With a proper introduction to the basics of VHDL combined with a logical and intelligent introduction of basic VHDL concepts, digital design students should be able to quickly and efficiently create useful VHDL models and enhance their understanding of digital systems and the modern digital design paradigm. In this way, students will be able to view VHDL as a valuable design, simulation, and test tool rather than another batch of throwaway technical knowledge encountered in some forgotten class or laboratory.

Lastly, VHDL is a powerful tool. The more you understand in the time you put into studying and working with VHDL, the more it will enhance your learning experience. The VHDL modeling paradigm is also an interesting companion to algorithmic programming. It is well worth noting that VHDL and other similar hardware design languages are used to create most of the digital integrated circuits found in the various electronic gizmos that currently overwhelm our modern lives. The concept of using software to design hardware that is controlled by software will surely cause you endless hours of contemplation.

You could be learning one of many HDLs out there. The two major HDLs are VHDL and Verilog. This text opts to use VHDL for modeling digital circuits. Using an HDL is much like using a computer language. Once you learn the basic concepts of programming, you can quickly learn a different programming language because you'll only need to learn the syntax of the new language. In computer programming, the basic programming concepts do not change and are directly applicable most any programming language. The same is true when learning an HDL. If you understand the basic concepts of the HDL modeling paradigm, you only need to learn the syntax of the new language. Keep in mind that HDLs do have special "concepts" that you must know that are different from the basic concepts of a programming language. There are only a few, however, and they're not that complicated.

One final comment: the VHDL introduction presented in this text quickly brings you down the path to understanding VHDL and writing solid VHDL code. Being able to write solid VHDL code facilitates the design and understanding of digital circuits. The ideas presented herein represent the core ideas you'll need to get up and running with VHDL. This approach to VHDL in no way presents a complete description of the VHDL language. In an effort to expedite the learning process, this approach omits some of the fine details of VHDL. Anyone who has the time and inclination should feel free to further explore the true depth of the VHDL language. There are many online VHDL references and tutorials as

[168] This is a blatant attempt to increase the sales of the text without giving thought to the trials and tribulations of the digital design students who'll be using the text.

[169] There was such a book used at Cal Poly for years. The author worked at Cal Poly and I can tell you first hand that this guy knew close to nothing about VHDL. Cal Poly students were stuck with that book for years and only one person in the EE Department actually did something about it (no one else cared about the low quality of instruction that the book provided). Sort of strange, but welcome to the politics of academia.

well as many relatively inexpensive VHDL texts available from online booksellers[170]. If you find yourself becoming curious about what you're not being told about VHDL in this text, be sure to take a look at some of these references.

16.3 VHDL Introduction

VHDL has a rich and interesting history. But since knowing this history is probably not going to help you write better VHDL models, we'll only give it a brief mention here. It is, however, worthy to state what the VHDL acronym stand for. Actually, the "V" in VHDL is short of yet another acronym: VHSIC or Very High-Speed Integrated Circuit. The HDL stands for Hardware Description Language. Obviously, the state of technical affairs these days has obviated the need for nested acronyms. The "HDL" acronym is vastly important. VHDL is a true computer language with the accompanying set of syntax and usage rules. But VHDL is different from higher-level computer languages such as C and Java because "describing hardware", or hardware modeling, is the primary use for VHDL.

The tendency for most people familiar with a higher-level computer language such as C or Java is to view VHDL as just another computer language. This is not altogether a bad approach in that such a view facilitates a quick understanding of the language syntax and structure. The common mistake made by this approach is to attempt to "program" in VHDL, as you would "program" a higher-level computer language. Higher-level computer languages execute instruction in a sequential nature; VHDL does not.

The main function of VHDL is describing digital circuits[171]. Another way to look at this is that higher-level computer languages are used to describe algorithms (inherently sequentially executed algorithms) and VHDL is used to describe hardware (inherently parallel execution). These inherent differences should encourage and inspire you to rethink how you write VHDL models. Attempts to write VHDL code with a higher-level language style generally results in VHDL code that no one understands. Moreover, the tools used to synthesize the circuits described by this type of code have a tendency to generate circuits that generally don't work correctly and have bugs that are nearly impossible to trace. In addition, if the circuit does actually work, it will most likely be inefficient because the resulting hardware is needlessly large and overly complex[172]. This problem compounds once the size and complexity of your digital circuits becomes greater. Understanding the basics of VHDL allows you to avoid these problems.

Once again, VHDL represents a modern digital design paradigm. In other words, the emphasis in modern digital design is to develop skills that are immediately applicable and easily built upon. Modern digital design is more about appropriately[173] modeling digital circuits and maintaining a quality description of the circuit as opposed to learning a bunch of throwaway knowledge that only serves to allow the instructor to write and grade pointless exams more easily[174].

16.3.1 Primary Uses of VHDL

[170] Check Ebay or www.addall.com.
[171] Probably a better way to view VHDL is as a tool to "model" digital circuit. This leads to a more appropriate vernacular to describe a pile of VHDL source code: you generate VHDL models as opposed to writing VHDL source code. The latter terminology is more of a computer science-type lingo so we'll definitely try to steer clear of using it.
[172] And often times, circuits with these characteristics simply don't work and will never work despite the fact that you may be throwing a lot of time at them in order to wishfully get them working.
[173] The word appropriately here means that you stay within the designated boundaries of the modeling system. In this case, you understand the VHDL design paradigm and you use it to your advantage rather than fight it.
[174] Just think how smart you would be if everything you learned was actually useful and important…

Although HDLs have many uses, there are four primary purposes for learning and using hardware description languages such as VHDL. These four main purposes of VHDL align perfectly with a beginning digital designer's typically uses for VHDL.

1) **Modeling Digital Circuits and Systems:** A digital circuit/system is any circuit that processes or stores digital information. Although the word *model* is one of those overly used words in engineering, in this context is simply refers to a description of something that presents a certain level of detail. One of the primary great features of modeling digital circuits with VHDL is that the rich syntax rules associated with VHDL force you to describe the circuit in an unambiguous manner. In other words, modeling a circuit using VHDL guarantees a specific operation (this assumes a correctly synthesized model; see the next item). VHDL provides everything that is necessary in order to describe the operation any digital circuit.

2) **Digital Circuit Synthesis:** With readily available software tools, you'll be able to quickly translate your VHDL models into actual functioning circuits. VHDL models are magically[175] interpreted by software tools in such a way as to create actual digital circuits in a process known as *synthesis*. This allows you to implement relatively complex circuits in a relatively short period and allows you to spend more time designing your circuits and less time actually constructing your circuits[176].

3) **Digital Circuit Simulation:** Once you've generated a VHDL model, software tools can use this model in order to simulate how an actual implementation of the circuit would operate. VHDL was used for circuit simulation before software tools were created to use the VHDL models for circuit synthesis. There are other digital circuit simulators available to model your digital designs. These simulators typically contain some type of graphical method to model circuits. The problem with this approach is that as your digital circuits become more complex, you'll find yourself explicitly connecting a bunch of lines on the computer screen which quickly becomes tedious. The more intelligent approach to digital design is to start with a system that is to be able to describe exactly how your digital circuit works without having to worry about the details of connecting massive quantities of signal lines. VHDL fulfills these promises with the notion of "testbenches" (presented in a later chapter).

4) **Hierarchical Design Support:** As was previously stated, the only possible way to understand complex digital circuits is to model them on different levels of abstraction. VHDL contains the functionality necessary to support multiple levels of abstraction. As you will see later, the approach to representing complex digital circuits with VHDL is similar to the representing large software programs.

16.3.2 The Golden Rules of VHDL

Before we go further, here are a couple of items that you should never forget when working with VHDL. Adherence to these rules will save you time and heartache in your digital careers.

- **VHLD has Limitations:** Although VHDL is flexible enough to allow the modeling of a given circuit in a virtual infinite number of ways, do not push the limits of this flexibility. Keep in mind that humans write the software tools and thus they inherently have limitations. Just because your circuit synthesizes does not mean you have created a robust model. In other words, work with the limitations of the VHDL toolset in order to generate solid VHDL

[175] It's not really magic. There is actually a well-defined science behind it.

[176] Modern engineering companies no longer pay engineers to construct circuits. So why should you have to contruct them as part of your engineering education?

models. The awards go to those who can best model their complex circuits in simple ways rather than relying on the synthesizer to do the dirty work. Moreover, if you're using a synthesizer that was provided for free, it is far less likely that this synthesizer will actually do what you're hoping it will do[177] (you'll have to pay real money for the secret sauce).

- **VHDL is a Hardware Design Language:** Although most people have probably had previously exposure to some type of higher-level computer language, these skills are only indirectly applicable to VHDL. When you're working with VHDL, you're not "programming", you are "modeling hardware"; your VHDL code should reflect this fact. If your VHDL code appears too similar to code of a higher-level computer language, it's probably bad VHDL code[178]. As you will soon discover, several structures in VHDL appear similar to constructs in higher-level languages. Regardless of this fact, VHDL does not use these structures in the same way as the higher-level language. This point relates closely to the item in the previous bullet. In other words, it is vitally important that you don't abuse the VHDL syntax.

- **Have a general concept of what your hardware should look like**. Although VHDL is vastly powerful, if you don't understand the basic digital constructs[179], you won't be able to generate solid VHDL models, and thus your circuits will be inefficient (if they actually work in the first place). Digital design is similar to higher-level language programming in that even the most complicated programming at any level are decomposable into some simple programming constructs[180]. There is a strong analogy between higher level programming languages and digital design in that even the most complex digital circuits are describable in terms of basic digital constructs[181]. If you are not able to roughly envision the digital circuit you're trying to model in terms of basic digital building blocks, you'll probably misuse VHDL, thus angering the VHDL gods.

16.4 VHDL Invariants

This section contains information regarding the "non-technical" use of VHDL. We present this information now because it provides valuable direction that is usable anytime time you use VHDL. Although your approach to VHDL modeling in general will change as you acquire more advanced modeling skills, the information in this section remains constant. Although it's rarely a good idea for people to memorize anything, you should memorize the basic information presented in this section. Making these ideas second nature should help eliminate some of the drudgery involved in learning the syntax of a new computer language while laying the foundation for creating more robust VHDL models. Don't forget about this section: come back and reread it once you generated a few VHDL models as it will make more sense at that time.

The primary purpose of the invariants listed in this section is to give the digital designer creating the VHDL models an extensive amount of control over the final model. If you use this control properly, you VHDL models, simply stated, will be better[182]. Items such as case sensitivity and white space are meaningless to the software responsible for interpreting the models, but are of utmost importance for

[177] There is a lot involved with the synthesis process. Companies invest significant time and money into this process in hopes of synthesizing meaningful digital designs. The "free" (no cost) versions of design tools out there won't necessarily represent the best tools from any given company; you have to pay actual money for those.
[178] Meaning that your code will probably not work properly after you synthesize it, or, it will be a massively inefficient design even if it does work properly.
[179] And there's really only a few of them…
[180] It's the structured programming thing all over again…
[181] Basic digital constructs include one of the relatively few "standard" lower-level digital devices. .
[182] More understandable and readable.

the humans who may be tasked with understanding the models. A VHDL model with a neat appearance is a better model in that it transfers more information to the human reader in a faster, more efficient manner. You have two choices in VHDL land: write beautiful VHDL code or write crap; there is no in-between.

16.4.1 Case Sensitivity

VHDL is not case sensitive[183]. This means that the two statements shown in Figure 16.1 have the exact same meaning (don't worry about the syntax of the statement or what the statements actually means though). Keep in mind that Figure 16.1 shows examples of VHDL case sensitivity but are not good VHDL coding practices (as you'll find out later). The main issue behind case sensitivity is to maintain consistency in your models; in other words, don't change the case of items in your code as it makes the model slightly harder to understand. Keep in mind that often times, there will be a style file provided to you (and the remainder of your group) that describes the accepted coding practices of the group[184].

```
Dout <= A and B;
```

```
doUt <= a AnD b;
```

Figure 16.1 An example of VHDL case insensitivity.

16.4.2 White Space

VHDL code is not sensitive to white space (spaces and tabs). The two statements in Figure 16.2 have the exact same meaning. Once again, Figure 16.2 is not an example of good VHDL coding style in that neither of the statements is as readable as they could be. Note that Figure 16.2 once again shows that VHDL is not case sensitive.

In addition, worthy of noting on this topic is the use of tabs in your VHDL models. There are two reasons not to use tabs in your VHDL models. First, although your code looks great in your editor, it could possibly look like crap (improper indentation) in an editor that someone else who needs to work with your code is using[185]. Second, when you generate hard-copies of your code, each printer contains an evil demon that interprets you tab characters in such a way as to mess up the appearance of your code. The result is crappy looking VHDL code and loss of friends in your social network.

```
nQ      <= In_a or In_b;
```

```
nQ <= in_a    OR        in_b;
```

Figure 16.2: An example showing VHDL's indifference to white space.

16.4.3 Comments

Comments in VHDL begin with "--" (two consecutive dashes). The VHDL synthesizer ignores anything after the two dashes and up to the end of the line in which the dashes appear. Figure 16.3

[183] Deep down in the bowels of VHDL, there are some instances of case sensitivity but you'll more likely than not run into these instances for a long time.
[184] Always ask for a copy of the style file; if there isn't one, look for a new job or better supervisor.
[185] When you're working on a large project with many people, it is inevitable that many different text editors will be used. Not all of these editors handle whitespace in the same manner.

shows two types of commenting styles. Unfortunately, there are no block-style comments (comments that span multiple lines but don't require comment marks on every line) in available in VHDL[186].

Appropriate use of comments increases both the readability and the understandability of VHDL code. The general rule is to comment any line or section of code that may not be clear to a reader of your code. The only inappropriate use of a comment is to state something that is patently obvious. It's hard to imagine code that has too few comments: don't be shy; use lots of comments. Research has shown that using lots of appropriate comments is a sign of high intelligence.

There is always an issue of where exactly to place comments. The best option is to place comments on lines before the line of code that you're commenting. The second best option is to place the comment after the VHDL code on a particular line. This approach only works for really short comments and should not be used otherwise. The third option is to place the comment after the line of code that it is describing. You should never do this: way too confusing and thus ugly. See Figure 16.4 for a few more exciting examples.

No matter how you use comments, be sure to keep the length of the comment on the given line relatively short. Long comments can go off the visible page or wrap on a hardcopy; the look really sad. Your mission, however, is to write happy VHDL code.

```
-- This next section of code is used to blah-blah
-- blah-blah blah-blah. This type of comment is the best
-- fake for block-style commenting.

PS_reg <= NS_reg;     -- Assign next_state value to present_state
```

Figure 16.3: Two typical uses of comments.

```
-- The best place for a comment is here
b_val <= a_val AND c_val;


-- Comments after the code on the line are OK, but keep them short
-- Good:
d_val <= d_val AND f_val; -- D calculation

-- BAD:
d_val <= d_val AND f_val; -- calculation of d_val based on previous results


-- Comments on the line of after the code are confusing:
d_val <= d_val AND f_val;
-- D calculation
```

Figure 16.4: Two typical uses of comments.

16.4.4 Parenthesis

VHDL is relatively lax on its requirement for using parenthesis. Like other computer languages, there are precedence rules associated with the various operators in the VHDL language. Though it is possible

[186] Many editors, however, are able to comment and uncomment large sections of code automatically. The moral of this story: ***know thy editor***. Many editors are also able to edit user specified blocks of text; a very handy feature.

to learn all these rules and write clever VHDL source code that will ensure the readers of your code will be scratching their heads, a better idea is to practice the liberal use of parenthesis to ensure the human reader of your source code understands your code. As an example, the VHDL synthesizer interprets the two statements shown in Figure 16.5 as being equivalent. Note that in Figure 16.5, the extra white space in addition to the parenthesis to makes the lower statement more clear. Don't worry about the syntax presented in Figure 16.5; we'll touch on that later.

```
if x = '0' and y = '0' or z = '1' then
    blah;  -- some useful statement
    blah;  -- some useful statement
end if;

if  ( ((x = '0') and (y = '0')) or (z = '1') )  then
    blah;  -- some useful statement
    blah;  -- some useful statement
end if;
```

Figure 16.5: An example of parenthesis use that produces clarity and happiness.

16.4.5 VHDL Statement Termination

Similar to other algorithmic computer languages, VHDL uses a semicolon to terminate statements. This fact helps when attempting to remove compile errors from VHDL code since semicolons are often mistakenly omitted during initial coding. The main challenge then becomes to know what constitutes a VHDL statement in order to know when to include semicolons. The VHDL synthesizer is not as forgiving as other languages when superfluous semicolons are place in the source code. In other words, if you forget to include a semicolon on a particular line, the synthesizer rarely flags it as an error.

16.4.6 Control Constructs: if, case, and loop Statements

As you soon will find out, the VHDL language contains **`if`**, **`case`**, and **`loop`** statements. A common source of frustration that occurs when learning VHDL is the classic dumb mistakes involving these statements. Always remember the rules stated below when writing or debugging your VHDL code and you'll save yourself a lot of time.

- Every **`if`** statement has a corresponding **`then`** component
- Each **`if`** statement is terminated with an "**`end if`**"
- If you need to use an "`else if`" construct, the VHDL version is "**`elsif`**"
- Each **`case`** statement is terminated with an "**`end case`**"
- Each **`loop`** statement has a corresponding "**`end loop`**" statement

16.4.7 Identifiers

An *identifier* refers to the name given to discern various items in VHDL. Examples of identifiers in higher-level languages include variable names and function names. Examples of identifiers in VHDL include variable names, signal names, entity names and architecture names (all of which will be

discussed soon). The list below shows the hard and soft rules (i.e., you must follow them or you should follow them, respectively) regarding VHDL identifiers. Remember, intelligent choices for identifiers make your VHDL code more readable, understandable, and more impressive to coworkers, superiors, family, and friends. People should quietly mumble to themselves "this is impressive looking code… it must be good" as they read your code. A few examples of both good and bad choices for identifier names appear in Table 16.1.

- Identifiers should be self-commenting. In other words, the text you use for identifiers should provide information as to the use and purpose of the item the identifier represents.
- Identifiers can be any length (contain many characters). Shorter names make for more readable code, but longer names present more information. It's up to the designer to choose a reasonable identifier length.
- Identifiers can only contain some combination of letters (A-Z and a-z), digits (0-9), and the underscore character ('_'); VHDL permits no other characters.
- Identifiers must start with an alphabetic character.
- Identifiers must not end with an underscore and must never have two consecutive underscores.

Valid Identifiers		Invalid Identifiers	
`data_bus_val`	descriptive name	`3Bus_val`	begins with numeric character
`WE`	classic "write enable" acronym	`DDD`	not self-commenting
`div_flag`	a real winner	`mid_$num`	contains illegal character
`port_A`	provides some info	`last__value`	contains consecutive underscores
`in_bus`	input bus (a good guess)	`start_val_`	ends with underscore
`clk`	classic system clock name	`in`	uses VHDL reserved word
		`@#$%%$`	total garbage
		`this_sucks`	true but try to avoid
		`Big_vAlUe`	valid way too ugly
		`pa`	possibly lacks meaning
		`sim-val`	illegal character (dash)

Table 16.1: Examples of desirable and undesirable identifiers.

16.4.8 Reserved Words

As with other computer languages, the VHDL language assigns special meaning to many words. These special words, usually referred to as *reserved words*, cannot be used as identifiers. A partial list of reserved words that you may be more inclined to use appears Table 16.2. A complete list of reserved words appears in the Appendix. Notably missing from Table 16.2 are standard operator names such as AND, OR, XOR, etc. (the basic gates used in digital logic).

access	exit	mod	return	while
after	file	new	signal	with
alias	for	next	shared	
all	function	null	then	
attribute	generic	of	to	
block	group	on	type	
body	in	open	until	
buffer	is	out	use	
bus	label	range	variable	
constant	loop	rem	wait	

Table 16.2: A short list of VHDL reserved words.

16.4.9 VHDL General Coding Style

Coding style refers to the appearance (as opposed to the function) of the VHDL source code. Obviously, the freedom provided by case insensitivity, indifference to white space, and lax rules on parenthesis creates a virtual coding anarchy. Generating readable code is therefore the main emphasis in VHDL coding style. Unfortunately, the level of readability of any document, particularly coding text, is subjective. Writing VHDL code is similar to writing code in other computer languages such as C and Java in that you have the ability to make the document more readable without changing the function of the document. Indenting certain portions of the code increases its readability and understandability, as does the use of self-commenting identifiers, and provided proper comments when and where necessary.

Instead of stating a bunch of rules for you to follow as to how your document should look, you should instead strive to make your source code *readable*. Listed below are a few thoughts on the notion of a readable document.

- Chances are that if your VHDL source code is readable to you, it will be readable to others who may need to peruse your document. These other people may include someone who is helping you get the code working properly, someone who is assigning a grade to your code, or someone who signs your paycheck at the end of the day. These are the people you want to please. These people are most likely massively busy and more than willing to make a subjective glance at your code; nice looking code will slant their subjectivity in your favor.

- If in doubt, you should model your VHDL source code after some other VHDL document that you find particularly organized and readable. VHDL code you initially encounter was most likely written by someone with more VHDL experience than a beginner such as yourself. Emulate the good parts of their style while on the path to creating an even more readable style of your own.

- Adopting a good coding style helps you write code without mistakes. As with other compilers you have experience with, you'll find that the VHDL compiler does a great job at knowing a document has error but a marginal job (at best) at telling you the exact location of the error or what the error is. Using a consistent coding style enables you to find errors both before compilation and/or after the compiler has noted an error.

- A properly formatted document explicitly presents information about your design that would not otherwise be readily apparent. This is particularly true in the case of indenting your code.

- Look for and/or request that someone provide you with a VHDL style-file that explicitly shows how your code should appear. Anyone who is evaluating the appearance of your VHDL code should provide you with a style-file.

16.5 Basic VHDL Design Units

The "black box" approach to any type of design implies a hierarchical approach where varying amounts of detail are available at each level of the hierarchy. In the black box approach, units of action which share a similar purpose are grouped together and abstracted to a higher level. Once this is done, the module is referred to by its inherently more simple black box representation rather than thinking about the circuitry that actually performs that functionality.

This "black box" approach has two main advantages. First, it simplifies the design from a systems standpoint. Examining a circuit diagram containing appropriately named black boxes is much more understandable than staring at a circuit containing a countless number of logic gates. Second, the black box approach allows for the reuse of previously written and working code, namely the black boxes. Previous chapter discussed other great reasons for using black box diagrams.

Not surprisingly, VHDL bases its descriptions of circuits on the black box approach. The two main parts of any hierarchical design are, 1) the black box, and 2) the stuff that goes in the black box (which of course can be other black boxes). In VHDL, the black box is specified (or referred to) by the *entity* and the stuff that goes inside the black box is specified (or referred to) as the *architecture*. For this reason, the VHDL entity and architecture are closely related. As you probably can imagine, creating the entity is relatively simple while properly describing the architecture requires the major portion of time and effort in VHDL modeling. Our approach in this chapter is to present an introduction to writing VHDL code by describing the *entity* and then moving onto some of basic details of writing the *architecture*.

16.5.1 The VHDL Entity

The *entity* is VHDL's version of the black box. The VHDL entity construct provides a method to abstract the functionality of a circuit description to a higher level. The entity essentially provides a simple "wrapper" for the underlying circuitry. This wrapper describes how the black box interfaces with the outside world. Since VHDL is describing a digital circuit, the entity simply lists the various inputs and outputs of the underlying circuitry. In VHDL terms, an *entity declaration* officially describes the black box but does not describe anything in the black box.

Figure 16.6 shows the syntax[187] of the entity declaration. Note that the *entity_name* provides a method to reference the entity. The *port clause* specifies the actual interface of the entity. Figure 16.7 shows the syntax of the port_clause. Lastly, in big-picture terms, the entity is all that the outside world needs to know in order to be able to interface with the entity.

[187] The bold font is used to describe VHDL keywords while italics are used to show names that are supplied by the writer of the VHDL code. The concept of boldness is for readability only; your VHDL synthesizer won't have a use for it.

```
entity entity_name is
   [port_clause]
end entity_name;
```

Figure 16.6: Generic form of an entity declaration.

```
port (
   port_name : mode  data_type;
   port_name : mode  data_type;
   port_name : mode  data_type
      );
```

Figure 16.7: Syntax of the port_clause.

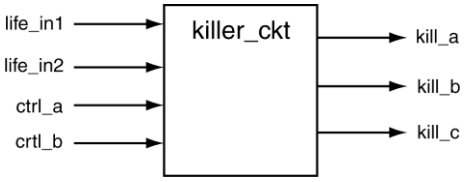
A "port" is essentially a conduit that interfaces a signal inside the box with a signal in the outside world. This signal can be either an input to the underlying circuit from the outside world or an output from the underlying circuit to the outside world. The *port clause* is nothing more than a list of the signals from the underlying circuit that are available to the outside world; this is why the entity declaration is often referred to as an interface specification. The *port_name* is an identifier used to differentiate the various signals. The *mode* specifies the direction of the signal relative to the entity and thus can enter (inputs) or exit (outputs) the black box[188]. These input and output signals are associated with the keywords **in** and **out**, respectively. The *data_type* refers to the type[189] of data associated with that port. There are many data types available in VHDL but we'll deal primarily with the **std_logic** type. We'll present information regarding the various VHDL data types in a later chapter; this will all make more sense once you start writing some actual VHDL models.

Figure 16.8 shows an example of a black box and the VHDL code that describes it. The list below describes a few things to note about the code in Figure 16.8. Most of the important things to note regard the readability and understandability of the VHDL code. The bolding of the VHDL keywords reminds you what the VHDL keywords are; most editors do not allow you to use bold text so don't expect to find bold text outside of this textbook.

- Each port name is unique and has an associated *mode* and *data type*. This is a requirement.
- The VHDL synthesizer allows several port names to be included on a single line and uses commas to delineate port names. This is not a requirement; whatever you choose to do, you should always strive for readability.
- The port names are somewhat lined up in a feeble attempt to increase readability. This again is not a requirement but you should always be striving for readability. Recall that the synthesizer ignores white space. There is no one great way to "line things up" so try to at least make it readable and for sure make it consistent (and don't use tabs).
- The VHDL code includes comments, which simulates the telling of almost intelligent things.

[188] There are actually other mode specifiers but we'll discuss them at a later time.
[189] VHDL is a strongly-typed language; there are many typing rules that you must follow.

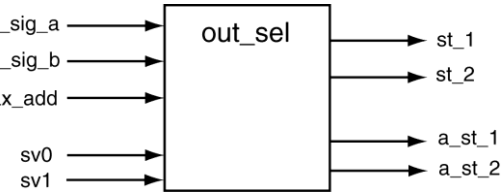
- This example provides a black box diagram of the model. Once again, drawing some type of diagram helps with any VHDL code and digital design in general. Remember… don't be a wuss; draw a diagram, and draw it before you start coding any VHDL model[190].




```
------------------------------------------------------------------
-- Here's my interface description of the killer circuit
-- It does a lot of killer things.
------------------------------------------------------------------
entity killer_ckt is
   port (     life_in1 : in std_logic;
              life_in2 : in std_logic;
        crtl_a, ctrl_b : in std_logic;
                kill_a : out std_logic;
                kill_b : out std_logic;
                kill_c : out std_logic);
end killer_ckt;
```

Figure 16.8: Example black box and associated VHDL entity declaration.

Figure 16.9 provides another example of a black box diagram and its associated entity declaration. All of the ideas noted in Figure 16.8 are equally applicable in Figure 16.9.




```
------------------------------------------------------------
-- out_sel is used to select one inputs based on the
-- conditions of sv0 and sv1 blah blah blah
------------------------------------------------------------
entity out_sel is
   port (big_sig_a, big_sig_b : in  std_logic;
                      sv0, sv1 : in  std_logic;
                      fax_add  : in  std_logic;
                    st_1, st_2 : out std_logic;
                a_st_1, a_st_2 : out std_logic);
end out_sel;
```

Figure 16.9: An example of an input/output diagram of a circuit and its associated VHDL entity.

Hopefully, you're not finding these entity specifications too challenging. In fact, they're so straight-forward, we'll throw in one last twist before we leave the realm of VHDL entities. Most the more meaningful circuits that you'll be designing, analyzing, and testing using VHDL have many similar and

[190] This is massively important; many beginning VHDL coders confuse themselves by starting their code without first drawing a black box model. Don't do this; it's a sure sign that you're wasting time.

closely related sets of inputs and outputs. These sets of signals are commonly referred to as bundles signals in computer lingo[191]. Bundles are made of more than one signal that differ in name by only a numeric reference character (or an "index"). In other words, each separate signal in the bundle name contains the bundle name plus a number to differentiate it from other signals in the bundle. Individual bundle signals are generally referred to as *elements* of the bundle.

The VHDL entity can easily describe bundles. VHDL uses a sort of new data type for bundles and a special notation to indicate when a signal is a bundle or not. Figure 16.10 shows a few examples of the new data type and associated syntax. In these examples note that the mode remains the same but the type has changed. The ***std_logic*** data type now includes the word *vector* to indicate each signal name contains more than one signal. There are ways to reference individual members of each bundle but we'll wait until later to discuss that notation.

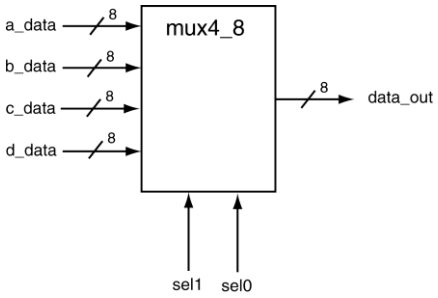
```
magic_in_bus     : in std_logic_vector(0 to 3);
big_magic_in_bus : in std_logic_vector(7 downto 0);
tragic_in_bus    : in std_logic_vector(16 downto 1);
data_bus_in_32   : in std_logic_vector(0 to 31);
mux_out_bus_16   : out std_logic_vector(0 to 15);
addr_out_bus_16  : out std_logic_vector(15 downto 0);
```

Figure 16.10: A few examples of bundled signals of varying content.

As you can see by examining Figure 16.10, there are two possible methods to describe the signals in the bundle. The argument lists shows these two methods in the parenthesis that follow the data type declaration. The signals in the bundle can be listed in one of two orders which is specified by the *to* or *downto* keyword. Producing VHDL code with greater clarity should be the deciding factor on which of these orientations to use, but strive to be consistent. Be sure not to forget the orientation of signals when you are using this notation in your VHDL model. A good practice is to adopt either the "to" or "downto" style and stick with it in all your VHDL models.

A more appropriate introduction to bundles would be to see this notation used to describe an actual black box. Figure 16.11 shows a black box followed by its entity declaration. Note that the black box uses a slash/number notation to indicate that the signal is a bundle. The slash across the signal line indicates the signal is a bundle and the associated number specifies the number of signals in the bundle. Worthy of mention regarding the black box of Figure 16.11 is that the input lines *sel1* and *sel0* could have been made into a single bundle containing two signals.

[191] These are also referred to as "buses"; but the term bundle is much better as the term "bus" has other connotations in digital-land.




```
--------------------------------------------------------------------
-- Unlike the other examples, this is actually an interface
-- for a MUX that selects one of four bus line for the output.
--------------------------------------------------------------------
entity mux4_8 is
   port (  a_data : in  std_logic_vector(0 to 7);
           b_data : in  std_logic_vector(0 to 7);
           c_data : in  std_logic_vector(0 to 7);
           d_data : in  std_logic_vector(0 to 7);
        sel1,sel0 : in  std_logic;
         data_out : out std_logic_vector(0 to 7));
end mux4_8;
```

Figure 16.11: A black box example containing bundles and its associated entity declaration.

16.5.2 The VHDL Architecture

The VHDL entity declaration describes the interface or the external representation of the circuit. The *architecture* describes what the circuit actually does. In other words, the VHDL architecture describes the internal implementation of the associated entity. As you can probably imagine, describing the external interface to a circuit is generally much easier than describing the operation of the circuit.

The most challenging part about learning VHDL is becoming familiar with the myriad of possible ways that VHDL can model a circuit. At this point in your digital design career, we'll hide a bulk of the details from you in an effort to make you more comfortable with the basic syntax. The approach we'll take is to present some simple examples that utilize simple VHDL operators. Once we introduce more details regarding digital logic and digital design, we'll introduce more details regarding VHDL. Don't feel bad that you really don't know what you're doing because you won't realize that you don't really know what you're doing until after you've been doing it for awhile[192].

16.5.3 The Architecture Body

The term "architecture body" is the name given to the thing that defines the input/output relationship to the ports listed in the VHDL entity. Because the input/output relationship for a given digital circuit can be complex, the VHDL designer can include a myriad of information in the architecture body. For now, we'll only present the basics and leave the more challenging stuff until later. Figure 16.12 shows the generic form of the architecture body.

[192] I have no idea what this means; I put it in here because it sounded incredibly stupid.

```
architecture arch_identifier of entity_name is
       {declarative region}
begin
       {statement region}
end arch_identifier;
```

Figure 16.12: The architecture body beautiful in all its generic glory.

There are a few key features in the architecture body commenting on. Don't allow the sheer number of items in the list below intimidate you; designing simple architecture bodies becomes second nature once you do it once or twice.

- The bold-face typing lists the VHDL keywords. The italicized text represented items that the VHDL designer needs to provide. The VHDL designer also provides the stuff in the braced delineated items; later bullets cover these items.

- The *arch_identifier* is a label that you must supply. Section 16.4.7 outlined the rules governing identifiers; try to make your identifier names give a hint as to the function of the architecture. In other words, all identifiers should be self-commenting.

- The *entity_name* is the name associated with the entity that a given architecture is describing. Be sure to apply self-comment when choosing names for your entities.

- The *declarative region* contains items that are used by the architecture but don't directly describe the operation of the circuit. You're actually able to put many different items in this area which emphasizes the versatility of VHDL in describing circuit operation; we'll cover some of the more useful items in a later chapter.

- The *statement region* contains VHDL statements that directly describe the operation of the circuit. We'll do a few simple example follow this boring description.

- Each architecture body contains one *begin* statement that terminates the *declarative region* and an associated *end* statement that terminates the architecture body.

16.6 Simple VHDL Models: entity and architecture

You now have enough information in order to model digital circuits using VHDL. The key to VHDL modeling is that every VHDL model has an entity/architecture pair. The examples used in this section are complete but they are simple enough such that the *declarative region* of the architecture is blank. For the following examples, we'll use the architectures to directly model Boolean expression that were presented in the previous chapter[193]. The *statement region* is filled with basic expressions that model the logic that implements the required circuit functionality.

VHDL uses various *operators* to implement standard logic gates. As you may know from higher-level computer languages, "operators" are used to implement "operations"[194]. VHDL likewise contains many useful operators. Fortunately, the only operators we'll need to get started are the operators associated with these logic functions. Table 16.3 shows an overview of these operators.

[193] The implication here is that there are always other ways to generate your models; we'll talk about those later.
[194] This sentence is vying for Sentence of the Year Award.

Logic Function	Logic Symbol	Logic Example	VHDL Operator	VHDL Example
AND	•	$A \cdot B$	AND	A **AND** B
OR	+	$A + B$	OR	A **OR** B
NAND	n/a	$\overline{A \cdot B}$	NAND	A **NAND** B
NOR	n/a	$\overline{A + B}$	NOR	A **NOR** B
XOR	$\oplus$	$A \oplus B$	XOR	A **XOR** B
XNOR	$\odot$	$\overline{A \oplus B}$	XNOR	A **XNOR** B
Compliment	$\overline{\quad}$	$\overline{A}$	NOT	(**NOT** A)

Table 16.3: A few of the logic operators used in VHDL.

Example 16-1

Provide the VHDL code that models the following Boolean equation:

$$F(B2,B1,B0) = B2 \cdot \overline{B1} \cdot B0 + B2 \cdot B1 \cdot \overline{B0} + B2 \cdot B1 \cdot B0$$

Solution: The place to start with any digital design problem is with a black box diagram. The black box diagram is particularly helpful in VHDL modeling because does a great job of modeling the entity. Figure 16.13 shows the black box diagram while Figure 16.14 provides the entire solution to Example 16-1.

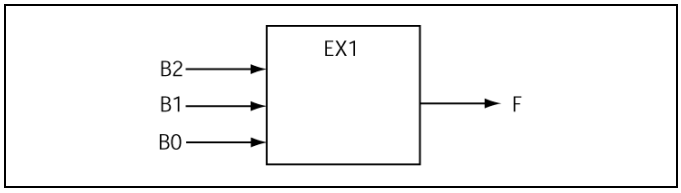


Figure 16.13: The black box diagram for Example 16-1.

```
entity ex1 is
   port (B2,B1,B0 : in std_logic;
                F : out std_logic);
end ex1;

architecture ex1_a of ex1 is
begin

   -- implement Boolean expression
   F <= (B2 AND (not B1) AND B0) OR
        (B2 AND B1 AND (not B0) OR
        (B2 AND B1 AND B0);

end ex1_a;
```

Figure 16.14: The full solution (VHDL model) for Example 16-1.

Although the solution in Figure 16.14 is relatively short, there are some important points to note:

- The entity name (ex1) associates the architecture with a specific entity. In this example, this architecture described the "ex1_a" version of the "ex1" circuit. In the real world, you may have multiple versions of the same circuit for various reasons.

- The declarative region of the architecture is blank. This model is relatively simple and we therefore don't need to put anything inside the declarative region.

- The solution includes a trivial comment in order to remind you of the importance of commenting your code.

- The individual product terms in the solution are placed on three separate lines and lined up nicely which increases the readability of the solution (a good use of white space). If it were not written this way, it would be really hard to read and may risk running off the page.

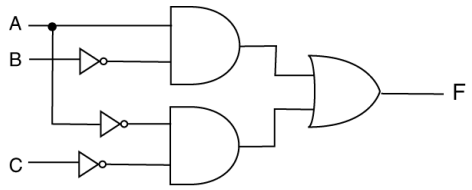
- The statement region of the architecture contains one statement. This one statement uses the *signal assignment operator* ("<=") to assign the result of the logic operations to the output. In other words, the value of the signal is determined by performing the logic operations on the right side of the signal assignment operator; the signal on the right side of the signal assignment operator is assigned the result.

- The model extensively uses parenthesis. You could write the statement without using parenthesis, but it would be confusing. Worst of all, it would require you to know the VHDL operator precedence rules (which I've don't know because I prefer to use parenthesis).

- If you really don't want to use parenthesis, the world will keep spinning. But, my general rule of VHDL is when using the "not" operator, always put the expression that is being inverted into parenthesis. If you don't, you'll generate errors that will be hard to find.

Example 16-2

Provide a VHDL model that is equivalent to the following circuit model:



Solution: Be sure to note the similarities between this example and Example 16-1. Figure 16.15 shows the black box diagram for this example; Figure 16.16 provides the full solution.

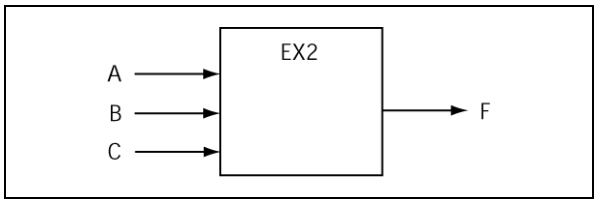


Figure 16.15: The black box diagram for Example 16-2.

```
entity ex2 is
   port (A,B,C : in std_logic;
               F : out std_logic);
end ex2;

architecture ex2_a of ex2 is
begin

   -- implement Boolean expression associated with ckt model
   F <= (A AND (not B)) OR ((not A) AND (not C));

end ex2_a;
```

Figure 16.16: The VHDL model for Example 16-2.

One good thing to note here is that the entity model name matches the model name provided by the black box model in Figure 16.15. This is one of those things that should always be done but in actuality one of those things that is rarely done. The problem is that most people rarely start their designs out by drawing a black box model; they usually draw the black box model after the completion of the VHDL model. I strongly encourage you to make drawing the black box model the first step in every design you embark on.

Chapter Summary

- VHDL is an integral part of modern digital design. Any introductory digital design text that does not integrate VHDL and basic logic concepts will make a good Kleenex substitute in a pinch.

- The main uses of VHDL include: 1) modeling digital circuits in an unambiguous manner, 2) simulating digital circuits, 3) generating actual hardware from VHDL models, and 4) hierarchical design support.

- VHDL has several apparent limits; these limitations are avoidable with proper use of VHDL.

    1. VHDL synthesis tools have their limits. Strive to keep your circuits simple, particularly by utilizing hierarchical design.

    2. VHDL is a "hardware design language" and not a computer programming language. Don't use VHDL constructs that are similar to programming constructs in a "software" manner.

    3. Have a general concept of what your hardware should look like and strive for circuit descriptions that are utilize standard digital circuits.

- The entity declaration describes the inputs and outputs to a circuit. This set of signals is often referred to as the interface to the circuit since these signals are what the circuitry external to the entity uses to interact with the given circuit.

- Signals described in the entity declaration include a mode specifier and a type. The mode specifier can be either an *in* or an *out* while the type is either a *std_logic* or *std_logic_vector*.

- The word *bundle* is preferred over the word *bus* when dealing with multiple signals that share a similar purpose. The word *bus* has other connotations that are not consistent with the bundle definition.

- Multiple signals that share a similar purpose should be declared as a bundle using a *std_logic_vector* type (or vector types). Bundled signals such as these are always easier to work with in VHDL as compared to scalar types such as *std_logic*.

- VHDL models generally comprise of an *entity* and *architecture*. The entity describes the interface of the circuit (the inputs and outputs to the associated black box) while the architecture describes the operational characteristics of the circuit.

- Basic VHDL operators include logic operators (AND, OR, and NOT) and signal assignment operators ("<=").

Chapter Exercises

1) Why is it important never to use tab characters in your text used for coding purposes? Briefly explain.

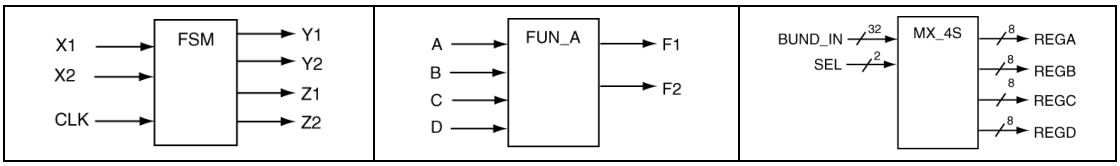
2) What is referred to by the word *bundle*?

3) Why is the word bundle more appropriate to use than the word *bus*?

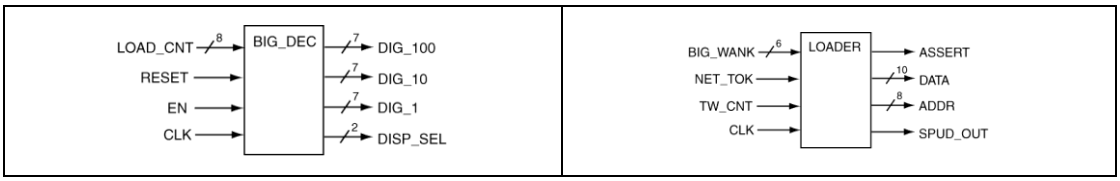
4) What is a common method of representing bundles in black box diagrams?

5) Why is it considered a good approach to always draw a black box diagram when using VHDL to model digital circuits?

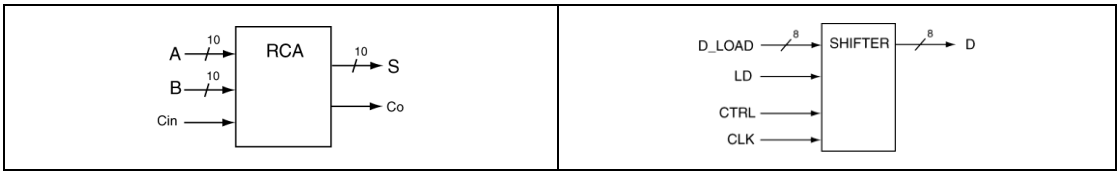
6) Write VHDL entity declarations that describe the following black box diagrams:



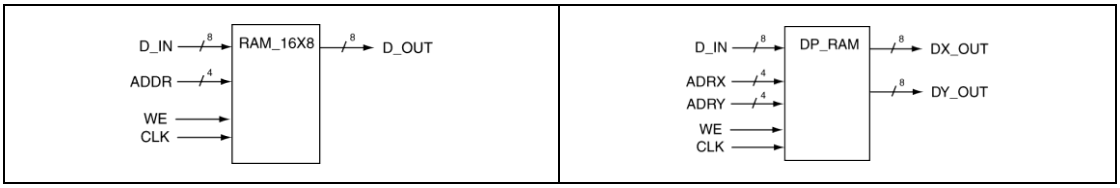
(a) (b) (c)



(d) (e)



(f) (g)



(h) (i)

7) Provide black box diagrams that are defined by the following VHDL entity declarations:

```
entity ckt_a is
   port ( in_a : in std_logic;
          in_b : in std_logic;
          in_c : in std_logic
         out_f : out std_logic);
end ckt_a;
```

(a)

```
entity ckt_b is
   port (    LDA,LDB : in std_logic;
             ENA,ENB : in std_logic;
         CTRLA,CTRLB : in std_logic;
           OUTA,OUTB : out std_logic);
end ckt_b;
```

(b)

```
entity ckt_c is
   port (   bun_a,bun_b_bun_c : in std_logic_vector(7 downto 0);
                  lda,ldb,ldc : in std_logic;
          reg_a, reg_b, reg_c : out std_logic_vector(7 downto 0);
end ckt_c;
```

(c)

```
entity ckt_d is
   port ( big_bunny : in std_logic_vector(31 downto 0);
                mx : in std_logic_vector(1 downto 0);
          byte_out : out std_logic_vector(7 downto 0));
end ckt_d;
```

(d)

```
entity ckt_e is
   port ( RAM_CS,RAM_WE,RAM_OE : in std_logic;
             SEL_OP1, SEL_OP2 : in std_logic_vector(3 downto 0);
                  RAM_DATA_IN : in std_logic_vector(7 downto 0);
                  RAM_ADDR_IN : in std_locic_vector(9 downto 0);
                 RAM_DATA_OUT : out std_logic_vector(7 downto 0));
end ckt_e;
```

(e)

```
entity ckt_f is
   port ( rss_bytes, rss_sux, rss_dogface : in std_logic;
             worthless, way_bad, go_away : in std_logic_vector(23 downto 0);
                big_joke, insecure, lazy : out std_logic_vector(32 downto 0);
                                     SMD : out std_logic_vector(7 downto 0));
end ckt_f;
```

(f)

8) The following two entity declarations contain two of the most common syntax errors made in VHDL. What are they?

```
entity ckt_a1 is
   port ( J,K : in std_logic;
          CLK : in std_logic
            Q : out std_logic;)
end ckt_a1;
```

(a)

```
entity ckt_d is
   port ( mr_fluffy : in std_logic_vector(15 downto 0);
            mux_ctrl : in std_logic_vector(3 downto 0);
            byte_out : out std_logic_vector(3 downto 0);
end ckt_d;
```

(b)

9) Provide VHDL models that implement the following Boolean expressions.

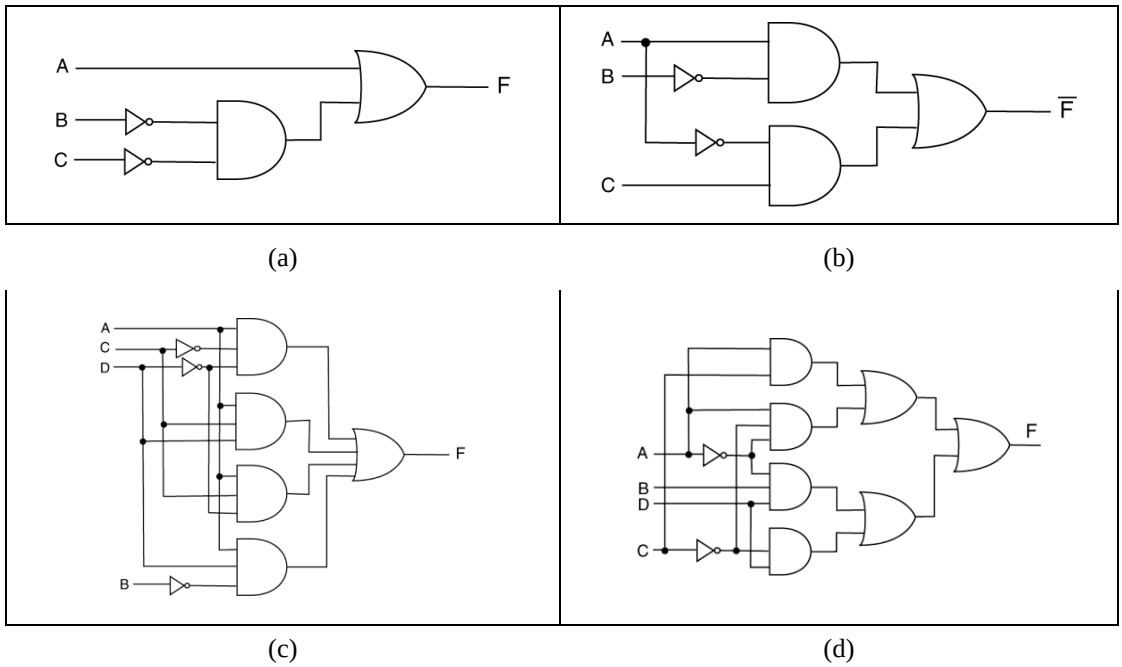
a) $F(A,B,C) = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}$

b) $F(X,Y,Z) = \overline{X} \cdot \overline{Y} \cdot Z + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$

c) $F(A,B,C) = (A+B+C) \cdot (A+\overline{B}+C) \cdot (A+\overline{B}+\overline{C}) \cdot (\overline{A}+\overline{B}+C)$

d) $F(X,Y,Z) = (\overline{X}+\overline{Y}+Z) \cdot (\overline{X}+Y+\overline{Z}) \cdot (X+\overline{Y}+Z) \cdot (\overline{X}+\overline{Y}+\overline{Z})$

10) Provide VHDL models that implement the following circuit models.



11) Provide a VHDL model that implements a half adder (HA). Use only one architecture for your solution.

12) Write an equivalent equation for F in reduced SOP form using the following VHDL model. Do not draw the circuit.

```
entity ex1_model is
    Port ( A,B,C : in std_logic;
           F : out std_logic);
end ex1_model;

architecture ex1_model of ex1_model is
begin

   F <= ( A OR (not B) OR C) AND
        ( A OR (not B) OR (not C)) AND
        ( (not A) OR B OR (not C)) AND
        ( (not A) OR (not B) OR (not C))  ;

end ex1_model;
```

13) Write an equivalent equation for F in reduced POS form using the following VHDL. Do not draw the circuit.

```
entity exam1_model is
    Port ( A,B,C : in std_logic;
           F : out std_logic);
end exam1_model;

architecture exam1_model of exam1_model is
begin

   F <= ((not A) AND (not B) AND (not C)) OR
        ((not A) AND B AND (not C)) OR
        (A AND (not B) AND (not C)) OR
        (A AND (not B) AND C);

end exam1_model;
```

Design Problems

1) Provide a VHDL model that could be used to synthesize a circuit that performed as follows. If the two 2-bit inputs to a circuit are equivalent, the circuit outputs a the value of "11". Otherwise, the circuit outputs a value of "00".

2) Provide a VHDL model that could be used to synthesize a circuit that performed as follows. The four inputs are equivalent, the circuit outputs a '1'. The circuit also outputs a '1' when the circuit's inputs considered as a 4-bit unsigned binary value are odd and less than 9. Otherwise, the circuit outputs a '0'.

17 Chapter Seventeen

(Bryan Mealy 2012 ©)

17.1 Chapter Overview

One of the underlying themes in digital design is the use of modularity. You've already seen the power of the modular approach to digital design in our discussion of modular design (MD). The modularity theme is also one of the major themes of computer program design. What you'll find as you work through this text is that there are simply not that many different types of basic digital circuits out there. In other words, once you learn the few basic digital circuit types out there and become fluent with their use, you'll be well on your way to be coming a great digital designer. This means that you can subdivide even the most complex digital circuit into a set of the relatively few standard digital circuits[195]. However, if you take that previous statement and look at it in the opposite way, you can create complex digital circuit designs by connecting a set of standard digital circuits in an intelligent way.

It should be no surprise that VHDL fully supports all aspects of modular design. The modular design approach to modeling in VHDL is referred to as "structural modeling". Structural modeling supports all of the intuitive aspects of modular digital design; the only issue here is learning a new set of syntax associated with structural design.

Main Chapter Topics

- **VHDL STRUCTURAL MODELING OVERVIEW:** This chapter presents an overview and the motivation behind VHDL structural models.
- **VHDL STRUCTURAL MODELING MECHANICS:** This chapter shows some of the syntax and mechanics involved in using structural modeling. The examples presented in this chapter are basic but form a complete foundation of VHDL structural model.

Why This Chapter is Important

This chapter is important because it describes VHDL structural modeling, which is a modeling approach that supports hierarchical design with VHDL.

17.2 Modular Digital Design

The best approach to becoming an efficient digital designer is to incorporate *modular digital design* techniques wherever possible. The modular design approach bypasses some of the basic limitations of truth table-based design. Modular digital design is done at a higher-level than truth table-based iterative

[195] The "structured programming" approach to modular digital design.

designs[196]. While the main components in truth table-based designs were the independent variables, the main components of modular digital design are pre-designed circuits (the modules). The modular design approach is powerful because it generally models circuits using pre-designed (and tested) modules connected by a combination of signals and possibly something that is generally referred to as *glue logic*[197]. The modular design approach increases your efficiency as a digital designer by allowing you to assemble previously designed modules into new and different circuits. You should never find yourself reinventing the wheel in your digital designs[198].

One of the many great attributes about VHDL is its support of modular design and module reuse. Although we're not at the point of designing overly meaningful circuits at the modular digital design level, we are, however ready to understand the VHDL-based support mechanism of modular digital design. Although the examples in this chapter seem rather simple, they represent the basis of implementing and understanding even the most complex digital circuits. The VHDL mechanism used to support modular digital design is referred to as *structural modeling*. VHDL structural modeling supports modular digital design by directly supporting the hierarchical design partitioning. It is not a stretch to state that any complex digital model that does not employ structural modeling techniques represents a substandard use of VHDL. Try not to be substandard; there are too many people like that already[199].

17.3 VHDL Structural Modeling

The design of complex digital circuits using VHDL should closely resemble the structure of complex computer programs. Many of the techniques and practices used to construct large and well-structured computer programs written in higher-level languages can and should be applied when using VHDL to describe digital designs[200]. The common structure we are referring to is the ever so popular *modular* approach to coding. The term structural modeling is the terminology that VHDL uses for the modular design. The VHDL modular design approach directly supports hierarchical design, which is essential when attempting to understand complex digital designs.

The benefits of modular design to VHDL are similar to the benefits that modular design or object oriented design provides for higher-level computer languages. Modular designs promote understandability by packing low-level functionality into modules. These modules are easily reused in other designs thus saving the designer time by removing the need to reinvent and retest the wheel.

The hierarchical design approach extends beyond code written on the file level. VHDL modules can be placed in appropriately named files and libraries in the same way as higher-level languages. Moreover, there are often design libraries out there that contain useful modules that are only accessible using a structural modeling approach[201]. Having access to these libraries and being fluent in their use serves to increase your efficiency as a designer and allows you to design better circuits.

---

[196] Recall the cases of the half and full adders vs. the ripple carry adder. The HA and FA were designed with a purely iterative approach while the RCA was design with an iterative modular approach. In other words, the HA and FA used truth tables as a starting point of the design while the RCA used a modular approach.

[197] This term is analogous to the term glue code used in computer science. Many computer programs can be modeled as a set of modules that are made to interact with each other with the additions of some extra code referred to as *glue code*. Similarly in digital design, glue logic is some extra logic that is required to assure the correct interaction of pre-design hardware modules.

[198] Actually, you should always be looking to reuse previously designed modules in your designs.

[199] And most of them end up being academic administrators (needless to say)…

[200] Simply stated, VHDL is designed to support this form of coding efficiency.

[201] This means that you can use the module but you can't see the model the VHDL model that generated the module.

Finally, after all the commentary regarding complex designs, we present a few simple examples. Though the structural modeling approach is most appropriate for complex digital designs, the examples presented in this section are rather simplistic in nature. These examples show the essential details and associated syntax of VHDL structural modeling. It is up to the designer to conjure up digital designs where a structural modeling approach would be more appropriate.

We are still relatively early into the world of digital design so you'll probably be thinking that structural modeling is unnecessary. However, don't be fooled; you'll soon be to the point where the only way you can intelligently model your circuits using VHDL is to use structural modeling. This chapter is essentially showing you how to design at a higher-level by showing you how to reuse modules in your current design.

17.3.1 VHDL and Programming Languages: Exploiting the Similarities

The main mechanism for modularity in higher-level languages such as C is the *function*. In other less useful computer languages, the use of the methods, procedures, and subroutines accomplish a similar modularity. Figure 17.1 lists the general approach used to support modularity in C programs.

1. Name the function interface you plan on writing (the function declaration)
2. Code what the function will do (the function definition or function body)
3. Let the program know it exists and is available to be called (the proto-type)
4. Call the function from the main portion of the code.

Figure 17.1: The modular approach to computer programming.

The general approach used to support modularity in VHDL is similar to the modular approach in C programming; Figure 17.2 outlines this approach. Table 17.1 lists the similarities between the C and the VHDL approach to modularity in a handier format.

1. Name the module you plan to describe (the entity)
2. Describe what the module will do (the architecture)
3. Let the program know the module exists and can be used (component declaration)
4. Use the module in your code (component instantiation, or mapping).

Figure 17.2: The modular approach to VHDL models.

"C" programming language	VHDL
Describe function interface	the *entity*
Describe what the function does (coding)	the *architecture*
Provide a function prototype to main program	*component declaration*
Call the function from main program	component instantiation (mapping)

Table 17.1: Similarities between modules in "C" and VHDL.

A well-structured computer program has the form of functions calling functions (calling functions, etc). Computer programs modeled in this manner are considered to have different levels with the levels based on the depth of the associated function calls. A VHDL model can be viewed in a similar manner. A properly constructed complex VHDL model generally is viewed as modules using modules (using modules, etc)[202]. The different levels associated with a VHDL models are associated with the depth of structural modeling used in the design. For example, a three-level computer program has at least one function that calls one other function; this function subsequently calls one other function. Similarly, a VHDL model contains a base module that contains at least one module; this one module then in turn uses one other module.

As with the modular approach to computer programming, there is nothing forcing you to use the modular approach in VHDL. But with both computer programming and VHDL modeling, *flat designs*, or the non-modular approach, are considered poor practice[203] in that they typically obfuscate the purpose of the program or design, respectively. Besides that, they really piss me off because although they are technically correct, they take too long to understand and they take even long to debug when they contain errors.

17.4 Structural Modeling Design Overview

A simple example best explains VHDL structural modeling. As you'll see, VHDL structural modeling concepts are primarily syntactical in nature: VHDL structural models do nothing other than allow you to implement black box designs that hierarchical in nature. The biggest challenging using VHDL for modeling circuits is designing the modules; accessing modules using structural modeling is a no-brainer once you get past the associated syntax. The examples that follow primarily instruct you on using and understanding the special VHDL syntax associated with structural modeling.

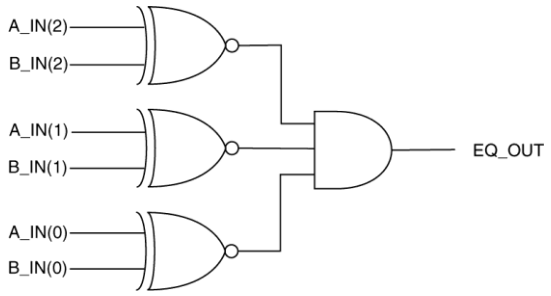
The use of structural modeling supports digital design, but it is by no means considered to be digital design. VHDL structural modeling simply allows you to implement your modular designs, which should inherently be black box models. The following example introduces structural modeling by implementing a simple circuit model with a circuit that you've already worked with. This example in reality is not an example of good VHDL modeling in that the circuit can be more easily modeled using other more straightforward not-yet-mentioned VHDL modeling techniques.

[202] Note the similarity to functions calling functions calling functions, etc.

[203] This is almost too general of a statement. Simple designs should definitely use flat models while complex designs should always use hierarchical models. Between these two statements is a lot of gray area so the overriding factor in using VHDL to describe circuits is to make your models as clear and concise as possible.

Example 17-1

Implement the following comparator circuit using a two-level VHDL structural model.



Solution: As you probably have realized, you can model this circuit in VHDL using a single statement (based on a single Boolean equation). Using one VHDL statement to model this circuit would be an example of a flat design (only one level). Since the problem states that the solution must use two levels, we must model this solution using a hierarchical approach and thus use structural modeling to implement our solution. The approach of this solution is to model each of the discrete gates as individual modules or "systems". For this example, these "systems" are actually simple gates but the interfacing requirements of the VHDL structural approach are the same regardless of whether the circuit elements are simple gates or complex digital subsystems.

The first step in this solution is to redraw the circuit shown in the problem statement. Figure 17.3 shows the original circuit model redrawn with some extra information added. The extra information provided in Figure 17.3 relates to the VHDL structural implementation. First, the dashed line represents the boundary of the top-level VHDL entity i.e., signals that cross this boundary must appear in the entity declaration for this implementation. Second, each of the internal signals of Figure 17.3 includes a label (or name). In this case, signals that do not cross the dashed entity boundary are considered internal signals. Assigning names to the internal signals is a requirement for VHDL structural implementations as this provides a mechanism to subsequently assigned signals to the various sub-modules on the interior of the design (somewhere in the architecture).

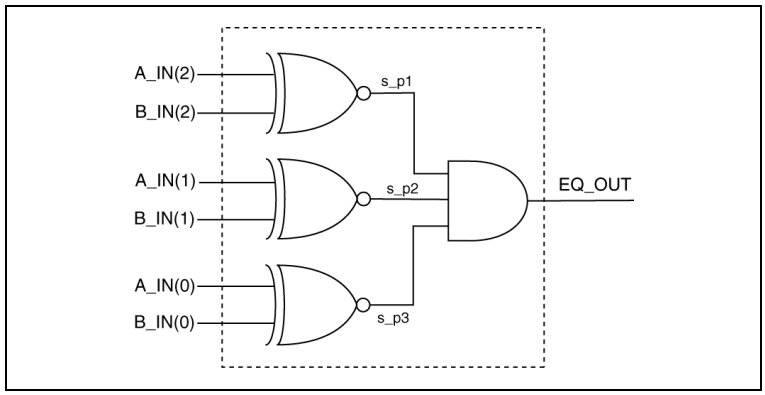


Figure 17.3: A redrawn version of the original circuit model.

The first VHDL-based portion of the solution is to provide entity and architecture implementations for the individual gates shown in Figure 17.3. We need to provided as least one definition for both the XNOR gate and one definition for the 3-input AND gate. We only need to provide one definition of the

XNOR gate despite the fact that Figure 17.3 shows three XNOR gates. The modular VHDL approach allows us to reuse circuit definitions and we freely take advantage of this feature. Figure 17.4 shows the VHDL models for the XNOR and AND gates; these implementations present no new VHDL details. The new information how VHDL uses the circuit elements listed in Figure 17.4 as modules in a circuit.

```
-------------------------------------------------------------
-- Descriptions of XNOR function
-------------------------------------------------------------
entity big_xnor is
    Port ( A,B : in std_logic;
              F : out std_logic);
end big_xnor;

architecture ckt1 of big_xnor is
begin
      F <= not ( (A and (not B)) or ( (not A) and B) );
end ckt1;


-------------------------------------------------------------
-- Description of 3-input AND function
-------------------------------------------------------------
entity big_and3 is
    Port ( A,B,C : in std_logic;
                F : out std_logic);
end big_and3;

architecture ckt1 of big_and3 is
begin
      F <= ( A and B and C );
end ckt1;
```

Figure 17.4: Entity and Architecture definitions for discrete gates.

Now that we've defined the lower-level modules, we're ready to implement the higher-level module. Figure 17.5 lists the basic procedures used for implementing a structural VHDL design. These steps assume that the entity declarations for the interior modules already exist (for this example, the XNOR and 3-input AND gate are considered the interior modules) and can be located by the VHDL synthesizer. Keep in mind that the following rules are not the best approach to "knowing" something, but are an adequate approach to learning something. After you do a few VHDL structural models, you'll not need to refer back to these rules (you'll find that there's really not that much to it).

1. Generate the top-level entity declaration
2. Declare the lower-level design units used in design
3. Declare required internal signals used to connect the design units
4. Instantiate and Map design units

Figure 17.5: The basic steps required in a VHDL structural model.

Step One: The first step in a structural implementation is identical to the standard approach we've used for the implementing other VHDL circuits: the entity. The entity declaration is derived directly from dashed box in Figure 17.3 and is shown in Figure 17.6. In other words, signals that intersect the dashed lines are signals that the entity uses to interface with the outside world and therefore must be included

in the entity declaration. Note that in Figure 17.3, the parenthetical operators associated with the input signal names imply that the circuit uses bundle notation. Figure 17.6 shows that subsequent entity declaration uses bundle notation. In actuality, a more appropriate start to this problem would be drawing a true higher-level black box such as the one shown in Figure 17.7.

```
-----------------------------------------------------
-- Interface description of circuit
-----------------------------------------------------
entity my_ckt is
    Port (   A_IN : in std_logic_vector(2 downto 0);
             B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_ckt;
```

Figure 17.6: Entity declaration for Example 17-1.

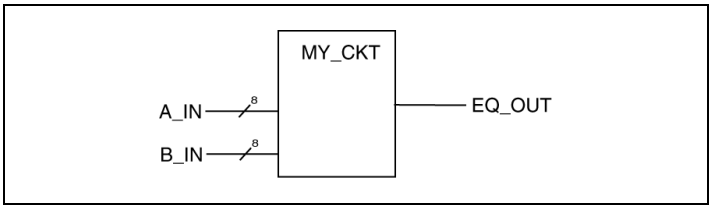


Figure 17.7: The proper higher-level black box model for Example 17-1.

Step Two: The next step is to declare the design units that the top-level circuit uses. In VHDL lingo, declaration refers to the act of making particular design units available for use in a given design. Note that the act of declaring a design unit, by definition, transforms your circuit into a hierarchical design. The declaration of a design unit makes the unit available for later placement in the overall design hierarchy. These design units are essentially modules that are used and/or defined in the lower levels of the design. For our design, we need to declare two separate design units: an XOR gate and a 3-input AND gate; Figure 17.4 defines both of these gates.

There are two factors involved in declaring a design unit: 1) how to do it, and, 2) where to place it. A component declaration is essentially a modification of the original entity declaration for the individual modules. The difference between an entity declaration and a component declaration is that the word *component* replaces the word *entity*; the word *component* must also follow the *end* keyword to terminate the declaration. The best way to do this is by cutting, pasting, and modifying the original entity declaration[204]. The resulting component declaration goes into the declarative region of the architecture declaration. Figure 17.8 shows the two entity declarations and their associated component. Figure 17.9 shows the component declarations as they appear in working VHDL code.

[204] Sometimes the original entity declaration is not available. In these cases, you'll be using a component from a design library and the component declaration is provided either directly or by "including" the particular design library.

entity big_xnor is
  Port (A,B : **in std_logic**;
  F : **out std_logic**);
end big_xnor;
	component big_xnor
	  Port (A,B : **in std_logic**;
	  F : **out std_logic**);
	end component;

entity big_and3 **is**
  Port (A,B,C : **in std_logic**;
  F : **out std_logic**);
end big_and3;
	component big_and3
	  Port (A,B,C : **in std_logic**;
	  F : **out std_logic**);
	end component;

Figure 17.8: A comparision of entity and component declarations.

Step Three: The next step is to declare internal signals used by your model. The required internal signals for this design are the signals that do not intersect the dashed lines shown in Figure 17.3 These three signals are analogous to local variables used in higher-level programming languages in that they must be declared before they are used in the design. These signals effectively provide an interface between the various design units that are present in the final.

For this design, three signals are required and used as both the outputs of the XOR gates and inputs to the AND gate. Internal signal declarations such as these appear with the component declarations in the declarative region of the architecture. Note Figure 17.9 that the declaration of intermediate signals is similar to the signal declaration contained in the entity body. The only difference is that the intermediate signal declaration does not contain the mode specifier. As you quickly find out, the use of intermediate signals in this manner is an extensively used feature in VHDL. The signal declarations are included as part of the final solution shown in Figure 17.9. Note that the signals contain the "s_" prefix which is a useful form of self-commenting that indicates the identifier is associated with a signal.

Step Four: The final step is to create "instances" of the required modules and map these instances to other components described in the architecture body. Technically speaking, as the word "instance" implies, the appearance of instances of design units is the main part of the component *instantiation* process. In some texts, the process of instantiation includes what we've referred to as component declaration but we've opted to keep these ideas separate. The approach presented here is to have the *declaration* refer to the component declarations in the declarative region of the architecture body, while *instantiation* refers to the creation of individual instances of the component in the statement region of the architecture body. The component mapping process is therefore included in our definition of component instantiation.

The mapping process provides the interface requirements for the individual components in the design. This mapping step associates external connections from each of the components to signals in the next step upwards in the design hierarchy. Each of the signals associated with individual components "maps" to either an internal or an external signal in the higher-level design. In other words, each signal associated with a component either must map to other components or to a signal in the entity associated with the next higher level in the design. Each of the individual mappings includes a unique name for the particular instance as well as the name associated with the original entity. The actual mapping information follows the VHDL keywords of: **`port map`**. Figure 17.9 shows all of information appearing in the final solution.

One key thing to note in the instantiation process is the inclusion of labels for all the instantiated design units. The design unit instantiation should always include labels in order to the understandability of your VHDL model. In other words, the proper choice of labels increases the self-commenting nature of your design; all intelligent uses of VHDL use this technique.

```
entity my_ckt is
    Port ( A_IN, B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_ckt;

architecture ckt1 of my_ckt is

   -- XNOR gate --------------------
   component big_xnor
      Port ( A,B : in std_logic;
                F : out std_logic);
   end component;

   -- 3-input AND gate -------------
   component big_and3
      Port ( A,B,C : in std_logic;
                   F : out std_logic);
   end component;

   -- intermediate signal declaration
   signal s_p1, s_p2, s_p3 : std_logic;

begin
   xnor1: big_xnor
   port map (A => A_IN(2),
             B => B_IN(2),
             F => s_p1);

   xnor2: big_xnor
   port map (A => A_IN(1),
             B => B_IN(1),
             F => s_p2);

   xnor3: big_xnor
   port map (A => A_IN(0),
             B => B_IN(0),
             F => s_p3);

   and1: big_and3
   port map (A => s_p1,
             B => s_p2,
             C => s_p3,
             F => EQ_OUT);
end ckt1;
```

Figure 17.9: VHDL code for the top of the design hierarchy for the 3-bit comparator.

It is worthy to note that the solution shown in Figure 17.9 is not the only approach to use for the mapping process. Figure 17.9 shows an approach that uses what is referred to a *direct mapping* of components. In this manner, each of the signals in the interface of the instantiated design units are listed and are directly associated with the signals they connect to in the higher-level design by use of the direct mapping operator "**=>**". This approach has several potential advantages: it is explicit, complete, orderly, and allows the signals to be listed in any order. The only possible downside of this approach is that it uses up a lot of space in your VHDL source code, a practice that rarely presents problems.

The other approach to mapping is to use *implied mapping*. In this approach, connections between external signals from the design units are associated with signals in the higher-level unit by order of their appearance in the mapping statement. This differs from direct mapping because only signals from the higher-level design appear in the mapping statement as opposed to direct mapping where signal names from the both levels are explicitly mapped. The ordering of the signals as they appear in the component declaration determines the association between signals in the design units and the higher-level design. This approach uses less space in the source code but requires placing signals using a

particular ordering. Figure 17.10 shows an alternate but equivalent architecture for Example 17-1 using implied mapping.

In reality, implied mapping provides no real advantage other than to save paper if you have to print out your VHDL models. I highly suggest you never use implied mapping your VHDL models, as it tends to create bugs that are terrifically hard to find. Direct mapping is clear and concise and provides a certain degree of self-commenting to your models. Therefore, use direct mapping.

```
architecture ckt2 of my_ckt is

   component big_xnor is
      Port ( A,B : in std_logic;
               F : out std_logic);
   end component;

   component big_and3 is
      Port ( A,B,C : in std_logic;
                 F : out std_logic);
   end component;

   signal s_p1, s_p2, s_p3 : std_logic;

begin
   xnor1: big_xnor port map (A_IN(2),B_IN(2),s_p1);
   xnor2: big_xnor port map (A_IN(1),B_IN(1),s_p2);
   xnor3: big_xnor port map (A_IN(0),B_IN(0),s_p3);
   and1:  big_and3 port map (s_p1,s_p2,s_p3,EQ_OUT);
end ckt2;
```

Figure 17.10: Alternative architecture for Example 17-1 using implied mapping.

Because this design was relatively simple, it was able to bypass one of the interesting issues that arise when using structural modeling. Often when dealing with structural designs, different levels of the design will often contain the same signal name. The question arises as to whether the synthesizer is able to differentiate between the signal names across the hierarchy. VHDL synthesizers, like compilers for higher-level languages, are able to handle such instances.

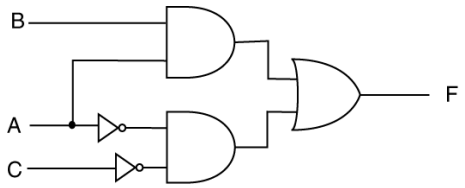
Signals with identical names at different levels are mapped according to what appears in the component instantiation statement. Probably the most common occurrence of this is with clock signals. In this case, a component instantiation such as the one shown in Figure 17.11 is both valid and commonly seen in designs containing a system clock. Name collision does not occur because the signal name on the left side of the direct mapping operator ("=>") is internal to the component while the signal on the right side is resides in the next level up in the hierarchy. Please avoid the temptation to rename one of these signals to make the model clearer (it will only make the model more oogly and everyone who reads your code will know you're a total wanker).

```
   x5: some_component port map (CLK => CLK,
                                CS => CS);
```

Figure 17.11: An example of the same signal name crossing hierarchical boundaries.

Example 17-2

Implement the following circuit using both a flat VHDL model and a two-level VHDL structural model.



Solution: The solution to this example is similar to the solution to the previous example. There are two points we want to make with this solution beyond which were not present in the previous solution. Figure 17.12 shows the black-box diagram for this problem and nicely supports the VHDL entity declaration provided in Figure 17.13. Keep in mind that he same high-level black box diagram is used for both the flat and hierarchical version of the solution.

Figure 17.14 shows the flat version of the solution. Note that the flat version model is short and to the point and represents the approach you should take when you're required to model relatively simple circuits such as this one. However, since this chapter introduces VHDL structural models, we'll continue modeling circuits in a less optimal manner[205].

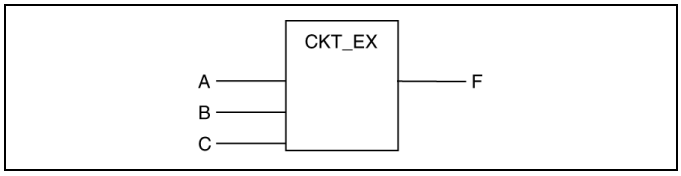


Figure 17.12: The entity declaration for Example 17-2.

```
entity ckt_ex is
    Port ( A,B,C : in std_logic;
                F : out std_logic);
end ckt_ex;
```

Figure 17.13: The entity declaration for Example 17-2.

```
architecture ckt1 of ckt_ex is
begin
   F <= (A AND B) OR ((not A) AND (not C));
end;
```

Figure 17.14: The architecture for the flat version of Example 17-2.

[205] In all likelihood, the development tools would interpret these circuits as being functionally equivalent and generate the exact same hardware. This fact emphasizes that writing clear and concise VHDL models is much more important than attempting to "optimize" (whatever that means) the length of your VHDL code.

Figure 17.15 shows the solution for the hierarchical version of Example 17-2. Figure 17.15(b) shows the VHDL models of the 2-input AND and OR gates while Figure 17.15(a) provides the remainder of the solution. In other words, Figure 17.15(b) provides the lower-level of the solution while Figure 17.15(b) provides the higher-level, or top-level, of the solution. Listed below are a few interesting points regarding the solution.

- Both architectures share the same entity declaration. In real life, there are often occasions where the same circuit has different models. For example, one version may be faster but larger and more power consuming than another version.

- This design used intermediate signals in several areas. The signal names include the "s_" pre-fix notation to quickly indicate to the reader that they intermediate signals. This helps the human reader of this solution instantly realize the difference between intermediate signals and signal appearing on the entity declarations. Adding such a prefix is good coding practice so you should strongly consider adopting this coding style.

- The model uses two separate statements to model the two inverters in the design. Although you could have modeled an inverter on a lower-level (as was done with the AND and OR gates), it is clearer to use the two statements as shown in Figure 17.15(a).

```
-- model for two-level solution
architecture ckt2 of ckt_ex is

   -- 2-input AND gate declaration -------
   component and_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   -- 2-input OR gate declaration --------
   component or_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   -- intermediate signal declaration
   signal s_a1, s_a2 : std_logic;
   signal s_inv1, s_inv2 : std_logic;

begin

   -- AND gate instantiation
   and1: and_2
   port map (A => A,
             B => B,
             F => s_a1);

   -- AND gate instantiation
   and2: and_2
   port map (A => s_inv1,
             B => s_inv2,
             F => s_a2;

   -- OR gate instantiation
   or1: or_2
   port map (A => s_a1,
             B => s_a2,
             F => F);

   -- inverters for A and C inputs
   s_inv1 <= not A;
   s_inv2 <= not C;

end ckt2;
```

(a)

```
-- model for 2-input AND gate
entity and_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end and_2;

architecture and_2 of and_2 is
begin
   F <= A AND B;
end and_2;



-- model for 2-input OR gate
entity or_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end or_2;

architecture or_2 of or_2 is
begin
   F <= A OR B;
end or_2;
```

(b)

Figure 17.15: The two-level solution for Example 17-2.

Finally, Figure 17.16 shows yet another solution for Example 17-2. This solution shows that the inverters can be modeled by placing the NOT operator into port mapping section of the instantiation. This approach is acceptable but does not work for any other operator other than the NOT operator. Attempting to place Boolean expressions in the port mapping clauses is not permissible and angers the VHDL goddesses.

```
-- another model for two-level solution
architecture ckt3 of ckt_ex is

   -- 2-input AND gate declaration -------
   component and_2 is
      Port ( A,B : in std_logic;
               F : out std_logic);
   end component;

   -- 2-input OR gate declaration --------
   component or_2 is
      Port ( A,B : in std_logic;
               F : out std_logic);
   end component;

   -- intermediate signal declaration
   signal s_a1, s_a2 : std_logic;

begin

   -- AND gate instantiation
   and1: and_2
   port map (A => A,
             B => B,
             F => s_a1);

   -- AND gate instantiation
   and2: and_2
   port map (A => (not A),
             B => (not B),
             F => s_a2;

   -- OR gate instantiation
   or1: or_2
   port map (A => s_a1,
             B => s_a2,
             F => F);

end ckt3;
```

Figure 17.16: Yet another solution for Example 17-2.

17.5 Practical Considerations for Structural Modeling

While reading about structural modeling is all good and fine, you don't really learn it until you actually implement a few models. It does initially seem like there is a lot of syntactical stuff to remember, but once you really start doing it and do it a few times, it becomes second nature. Structural modeling is not something you'll ever worry about once you get into it as it quickly becomes a no-brainer.You'll soon be more concerned with ensuring your digital design actually do what they're supposed to be doing rather than worrying about syntax issues.

Once you get deeper into VHDL, you'll be using structural modeling almost everywhere. However, as a beginner, there are a few things to note in order to get you going. Similar to higher-level language programming, there is a world full of previously written VHDL models that you can easily incorporate into your design using structural modeling. However, since you're probably learning VHDL now, your designs will be relatively simple and you won't be accessing these libraries as of yet.

When it comes to entity objects, you have two choices in the context of structural modeling. Either you can place all of the entities in the same physical VHDL file or you can use the VHDL development

environment you're using to make your particular design aware of entity objects that your design uses as components.

Figure 17.17 shows an example of the approach that places everything required by the structural model in the same file. The following verbage list a few more things to note about Figure 17.17

- The font is really small… this is because I was trying to fit it all on one page. Sorry about that. The comments disappeared for the same reason.

- The lower-level components appear in the file before the higher level-components. This style follows a C programming style in that the objects you use are defined before you use them; this makes them similar to functions (the function prototype or definition needs to be known to the compiler before the function is called in the program). This is not a requirement[206], but it makes the code more readable to the human viewer.

- There is a library clause before each entity declaration, which is a requirement of VHDL. While this seems troublesome, what it does is allow you to state which library you're using for each object. The idea supported here is that your design reference modules in different libraries. What you need to remember here is to cut and paste these lines and drop them in before each entity declaration even if your design only uses one library.

- Keep in mind that if you do put all your models into one file, make sure that you fully comment each of the modules. In other words, each module should contain all the information generally included with stand-alone VHDL files.

Figure 17.18 shows the multiple file approach to structural modeling for Example 17-2. Here are a few worthwhile things to note:

- The font is small and the meaningful comments are missing…

- Each of the cells in the table is considered to be in different physical VHDL file. In the end, it is up to you and/or you development environment to make sure the higher-level entity finds the lower level entities/components. Once again, if you're using modules from device libraries, the VHDL synthesizer needs to be able to find all this stuff. As far as VHDL development environments go, the software generally does all the magic stuff for you. The software analogy is of course that the environment creates its own makefile, which is nothing more than a master file that does that is required to implement your design. .

- The VHDL models in Figure 17.17 and Figure 17.18 are 100% equivalent. My personal feeling is that separate modules should only be in the same file if they in fact are somehow related. If they are not related, they should be in different file.

[206] Actually, there may be an environment out there where it is required.

```
--------------------------------------------------------------------------------
--------------- Filename: all_in_one_file.vhd ----------------------------------
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end and_2;

architecture and_2 of and_2 is
begin
   F <= A AND B;
end and_2;

------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end or_2;

architecture or_2 of or_2 is
begin
   F <= A OR B;
end or_2;

------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ckt_ex is
    Port ( A,B,C : in std_logic;
           F : out std_logic);
end ckt_ex;

architecture ckt2 of ckt_ex is

   component and_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   component or_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   signal a1_s, a2_s : std_logic;
   signal inv1_s, inv2_s : std_logic;

begin

   and1: and_2
   port map (A => A,
             B => B,
             F => a1_s);

   and2: and_2
   port map (A => inv1_s,
             B => inv2_s,
             F => a2_s;

   or1: or_2
   port map (A => a1_s,
             B => a2_s,
             F => F);

   inv1_s <= not A;
   inv2_s <= not C;

end ckt2;
```

Figure 17.17: The one-file approach to structural modeling.

```
-------------------------------------------------
-- here is the first file
-------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end and_2;

architecture and_2 of and_2 is
begin
   F <= A AND B;
end and_2;
```

```
-------------------------------------------------
-- here is the second file
-------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or_2 is
    Port ( A,B : in std_logic;
           F : out std_logic);
end or_2;

architecture or_2 of or_2 is
begin
   F <= A OR B;
end or_2;
```

```
-------------------------------------------------
-- here is the third file
-------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ckt_ex is
    Port ( A,B,C : in std_logic;
           F : out std_logic);
end ckt_ex;

architecture ckt2 of ckt_ex is

   component and_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   component or_2 is
      Port ( A,B : in std_logic;
             F : out std_logic);
   end component;

   signal a1_s, a2_s : std_logic;
   signal inv1_s, inv2_s : std_logic;

begin

   and1: and_2
   port map (A => A,
             B => B,
             F => a1_s);

   and2: and_2
   port map (A => inv1_s,
             B => inv2_s,
             F => a2_s;

   or1: or_2
   port map (A => a1_s,
             B => a2_s,
             F => F);

   inv1_s <= not A;
   inv2_s <= not C;
end ckt2;
```

Figure 17.18: The multiple file approach to structural modeling.

Example 17-3: Adder/Checker Thing Circuit

Design a circuit that performs and a few interesting tasks. The circuit has four 8-bit inputs: A, B, C, and D. If the 8-bit sum of A+B equals the 8-bit sum of C+D, then the EQ output of the circuit is a '1'. In addition, if both addition operations generate a carry-out, the CO2 output of the circuit should be a '1'. For this problem, use modular design for the solution with all modules other than simple gates being instantiated in the final design.

Solution: This is potentially a long problem, but we'll take some shortcuts in order to emphasize the structural modeling aspects of the problem. Figure 17.19 show the first task in this solution, which is to generate a black box diagram. The information required to generate Figure 17.19 is given in the problem description.

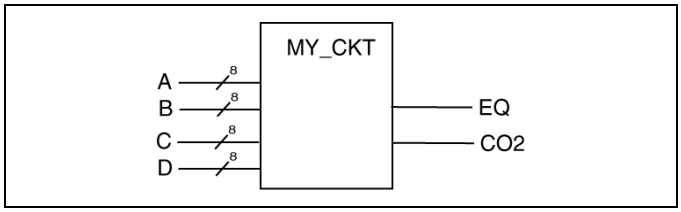


Figure 17.19: The high-level black-box model for this problem.

From the problem description, you can see that the final circuit is going to need two RCAs since we there are two stated addition operations. In addition, since the problem states that the circuit needs to check to see if the two intermediate results from the output of the RCAs are equal. Note that the problem was also careful to state that we did not need to worry about carry-outs from the RCAs as part of the comparisons. Finally, we need to indicate when the results of the two summation operations generate carry-outs; this can be done with a simple AND gate. After plopping down these modules and properly connecting them to each other and the associated inputs and outputs of the high-level block diagram, we end up with the result shown in Figure 17.20.

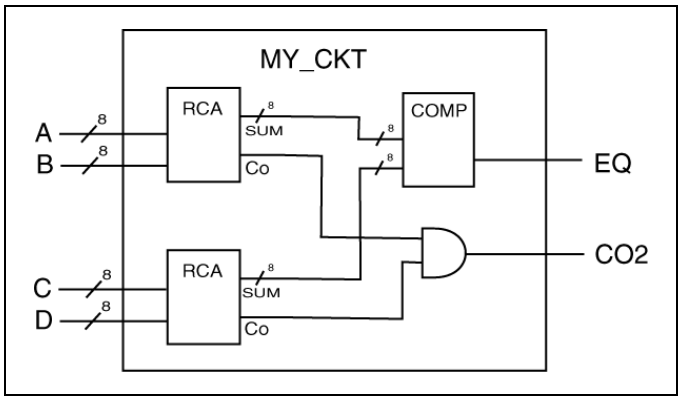


Figure 17.20: The full black-black-box model for this problem.

What we need now is a VHDL structural model solution to the problem. The toughest part of any design is generating the black box diagram; translating that diagram to a VHDL structural model is

essentially grunt work. However, Figure 17.21 shows the VHDL structural model solution to this problem. The following list shows a few useful comments.

- We're assuming that that RCA and the comparator have been defined elsewhere and can be found by the synthesizer. We have worked with these circuits previously so we won't bore you with the implementation details here.

- The AND function was implemented with a simple statement. This is a typical approach to VHDL structural modeling; you'll use this type of implementation quite often

```
entity my_ckt is
    Port ( A,B,C,D : in  STD_LOGIC_VECTOR (7 downto 0);
           EQ,CO2 : out  STD_LOGIC);
end my_ckt;

architecture Behavioral of my_ckt is

   -- RCA component declaration
   component RCA
   Port ( A,B : in  STD_LOGIC_VECTOR(7 downto 0);
          SUM : out  STD_LOGIC_VECTOR(7 downto 0);
           CO : out  STD_LOGIC);
   end component;

   -- comp component declaration
   component comp
   Port ( A,B : in  STD_LOGIC_VECTOR(7 downto 0);
           EQ : out  STD_LOGIC);
   end component;

   -- intermediate signal declarations
   signal s_sum_ab, s_sum_cd : std_logic_vector(7 downto 0);
   signal s_co_ab, s_co_cd: std_logic;

begin

   -- RCA instantiation
   rca_ab: RCA
   port map ( A => A,
              B => B,
              SUM => s_sum_ab,
              CO => s_co_ab);

   -- RCA instantiation
   rca_cd: RCA
   port map ( A => C,
              B => D,
              SUM => s_sum_cd,
              CO => s_co_cd);

   -- glue logic for EQ output
   CO2 <= s_co_cd AND s_co_ab;

   -- RCA instantiation
   comp_abcd: comp
   port map ( A => s_sum_ab,
              B => s_sum_cd,
              EQ => EQ);

end Behavioral;
```

Figure 17.21: Most of the final solution for Example 15-5.

Chapter Summary

- Structural modeling in VHDL supports hierarchical design concepts, which are hallmark of all digital design. The ability to abstract digital circuits to higher levels is the key to understanding and designing complex digital circuits. VHDL structural modeling is similar to higher-level programming in its abstraction and modularization capabilities.

- VHDL structural model supports the reuse of design units. This includes units you have previously designed as well as the ability to use pre-defined module libraries.

- VHDL structural modeling is can be divided into the four distinct steps: 1) generate the higher-level entity declaration, 2) declare the lower-level design units, 3) declare the intermediate signals, and 4) instantiated the required design units.

- Design unit instantiation includes the design unit mapping. Instantiation can use either *direct mapping* or *implied mapping*. It is, however, problematic to use implied mapping and always much better to use direct mapping.

- VHDL structural modeling establishes a design hierarchy, which is considered a multi-level design. A *flat* design is a VHDL model that does not use a structural model and is only used by total wankers such as one professor/author/wanker in the Cal Poly EE Department[207].

-

[207] The good news is that this person finally retired. The bad news is that the damage this person did to students will require a finite amount of time to repair. The bad new is that once an academic wanker retires, new and younger wankers generally replace them. This is how academia operates.

Chapter Exercises

1) Draw a block diagram of the circuit represented by the VHDL code listed below. Be sure to completely label the final diagram.

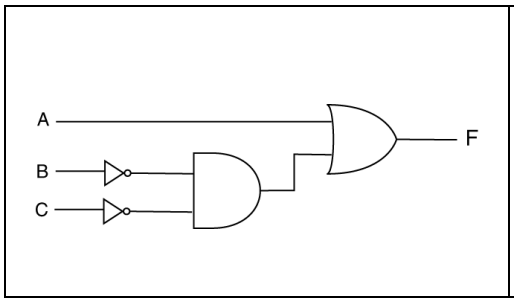
```
entity ckt is
    Port ( EN1, EN2 : in std_logic;
           CLK : in std_logic;
           Z : out std_logic);
end quiz1_ckt;

architecture ckt of ckt is

   component T_FF
      port ( T,CLK : in std_logic;
                Q : out std_logic);
   end component;

   signal t_in, t1_s, t2_s : std_logic;

begin
   t1 : T_FF
   port map (   T => t_in,
              CLK => CLK,
                Q => t1_s );

   t2 : T_FF
   port map (   T => t1_s,
              CLK => CLK,
                Q => t2_s );

   Z <= t2_s OR t1_s;
   t_in <= EN1 AND EN2;

end ckt;
```

(a)

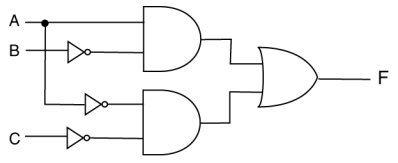
```
entity ckt is
   port ( A,B : in std_logic;
             C : out std_logic);
end ckt;

architecture my_ckt of ckt is

   component bb1
      port (   D,E : in std_logic;
             F,G,H : out std_logic);
   end component;

   component bb2
      port ( L,M,N : in std_logic;
                 P : out std_logic);
   end component;

   signal x1,x2,x3 : std_logic;

begin

   b1: bb1
   port map ( D => A,
              E => B,
              F => x1,
              G => x2,
              H => x3);

   b2: bb2
   port map ( L => x1,
              M => x2,
              N => x3,
              P => C);

end my_ckt;
```

(b)

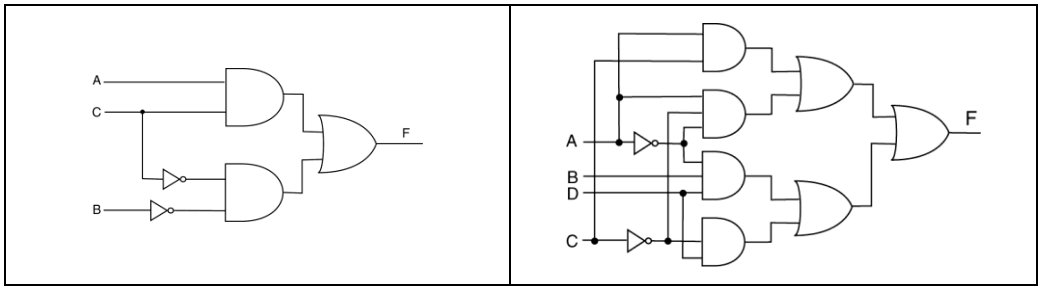
2) Provide VHDL structural models for the circuits listed below. Each of the associated VHDL models should contain at least two levels.



(a)



(b)



(c) (d)

3) Provide a VHDL structural model for a 4-bit ripple carry adder. Assume the lowest order bit is implemented with a full adder instead of a half adder.

18 Chapter Eighteen

(Bryan Mealy 2012 ©)

18.1 Chapter Overview

We are not telling you the entire story regarding VHDL. The approach we've been taking is to get our feet wet with VHDL in order to develop a basic understanding of the methodology used to generate basic digital circuits. However, since your knowledge of digital logic and design has increased, you need to increase your level knowledge regarding using VHDL to model digital logic circuits. As you'll see in this chapter, VHDL is much more structured and more powerful than we've been leading on. You're ready to understand both this structure and the VHDL modeling paradigm.

The slight problem regarding the material presented in this chapter is the fact that we base it around the simple logic functions we've been dealing with up to this point. As you'll surely discover in the upcoming chapters, digital design does not have that much to do with the implementation of functions. Implementing functions with VHDL is a low-level approach to digital design. We're at the point now where we're ready to perform our designs at a higher level. Digital design performed at a higher level of abstraction is more efficient than designing at lower levels.

Main Chapter Topics

- **VHDL MODELING:** The entity/architecture pair forms the interface and functional description of digital circuit behavior. Various VHDL modeling constructs supports the inherent parallelism in digital circuit.
- **VHDL CONCURRENT STATEMENT TYPES:** VHDL contains four major types of signal assignment statements: concurrent signal assignment, conditional signal assignment, selective signal assignment, and process statements. These statements are referred to as concurrent statements in that they are interpreted as acting in parallel (concurrently) to all other concurrent statements.
- **VHDL MODEL TYPES:** VHDL contains three major model types: 1) dataflow, 2) behavioral, and 3) structural models. The type of concurrent statements used in the model determines its model type.
- **VHDL BEHAVIORAL MODELING:** The process statement generally describes digital circuits in terms of its behavior as opposed to its low-level logic functions. Process statement use sequential statements to describe circuit behavior.

Why This Chapter is Important

This chapter is important because it provides the post-intro basics of modeling digital circuits using VHDL including the various flavors of concurrent statements.

18.2 More Introduction-Type Verbage

Painful as it may sound, implementing functions is the main topic of this chapter. While this is somewhat stupid[208], it allows you to get your feet wet with VHDL before doing meaningful design/modeling. There are times in digital design when you truly need to implement functions, but it is not the intent of this chapter to show you the best way to do that. The intent of this chapter is to provide an overview of VHDL from the standpoint of the possibilities that are out there lurking.

The truth here is that when you use VHDL, there is no need to bother reducing your functions. Essentially, you can assume the VHDL synthesizer will do the reduction for you and do it without making all the mistakes you probably make when you're reducing functions by hand. Yes, the Karnaugh Map is generally a relic of the past glory days of digital design. The only reason you're taught to work with K-maps is in case some old dude asks you to reduce a function during a job interview[209].

Lastly, really… Something you should become aware of in this chapter is that you've been dealing with "tables" a lot. There is a special type of a table in computer science-and that you should be aware as it is perfectly analogous to the truth tables you've been working with so far. In computer science, look-up tables, or LUTs are very useful and you find them quite often in viable computer programming code. Implementing a truth table via a function in digital-land is essentially the same thing as utilizing a LUT in computer science-land. We'll discuss this more later, but as with computer programming, you should always be on the lookout for placed where you can use a LUT rather than trying to use outdated techniques to generate some fancy logic functionality.

18.3 The VHDL Programming Paradigm

Our previous work with VHDL was limited to the idea of the basic VHDL design units: the entity and the architecture. We spent most of the time describing the architecture simply because there is so much less involved when compared to the entity[210]. The underlying theme of this chapter is to describe some of the structured modeling techniques used by the architecture bodies to describe digital circuits. In other words, VHDL has the ability to describe complex digital circuits; this chapter introduces some of those mechanisms.

Before we get into the newer details of architecture specification, let's step back and remember what it is we're trying to do with VHDL. We are, for one reason or another, describing a digital circuit. Realizing this simple fact is massively important. The tendency for students with computer programming backgrounds is to view VHDL as another programming language they need to learn to pass another class. Although many students have used this approach to pass digital design courses, this is a bad approach.

When viewed correctly, VHDL represents a completely paradigm than standard programming languages. This misuse of VHDL most likely arises because VHDL has many similarities to other programming languages. The main similarity is that they both use a syntactical and rule-based language to describe something relatively abstract. However, the difference is that they are describing two

[208] Digital design is not about "implementing functions". Even a chimpanzees and academic administrators can implement digital functions. Generating circuits that do something useful is the more impressive task.
[209] This is not exactly true. Working with K-maps also represents an instructive practice when first learning the nuts and bolts of digital design. It is sort of fun anyways. Above all, K-map questions appear on exams such as the EIT.
[210] Recall that the entity declaration describes the interface of a circuit to the outside world. The architecture describes how the circuit functions.

completely different things. Realizing this fact will help you to truly understand the VHDL programming paradigm and language, to churn out more meaningful VHDL code, and illuminate a nice contrast between a language that describes hardware and a language that executes software on that hardware.

18.3.1 Concurrent Statements

The heart of most programming languages is the statements that form the source code. These statements represent finite quantities of "actions" that the entity running the program needs to take. A statement in an algorithmic programming language such as C or Java represents an action or actions for the processor to take. Once the processor finishes one action, it moves onto the next action specified somewhere in the associated source code[211]. This makes sense and is comfortable to us as humans because just like the processor, we generally are only capable of doing one thing at a time and once we finish that one thing, we move onto the next thing. This description lays the foundation for an algorithmic programming in that the processor does great job at following a set of rules, which are comprised of instructions in the source code. When the rules are meaningful and well structured, the processor can do amazing things.

VHDL modeling is significantly different than computer programming. Whereas in computer programming where a processor steps one-by-one through a set of statements, VHDL can "execute[212]" a virtually unlimited number of statements at the same time (in other words, in parallel). Once again, the key thing to remember is that we use VHDL to describe digital hardware. Parallelism, or things happening *concurrently*, in the context of hardware is a much more straight-forward concept in hardware-land than it is in the world of software. As you've already had the basic introduction digital logic and its associated hardware, you're already both familiar and comfortable with the concept of concurrency whether you realize it or not.

Figure 18.1 shows a simple example of a circuit that is operating in parallel. As you know, logic gates are generally stupid in that the gate outputs are a simple function of the gate inputs. Anytime a gate input changes, there is a possibility that it will cause a change in the gate output. This is true of all the gates in Figure 18.1 or in any digital circuit in general.

The key here is that the changes in the input to these gates can happen simultaneously. In other words, changes to gate inputs can occur in parallel; once these changes occur, the inputs are re-evaluated and the gate outputs may change accordingly; this activity generally happens simultaneously to all gates in a particular design. Although the circuit in Figure 18.1 only shows a few gates, this idea of concurrent operation of all the elements in the circuit is the same in all digital circuits no matter how large or complex the circuits become.

[211] Either the next instruction, or somewhere else in the program in the case of a function call or branch.
[212] But it's not really executing statements…

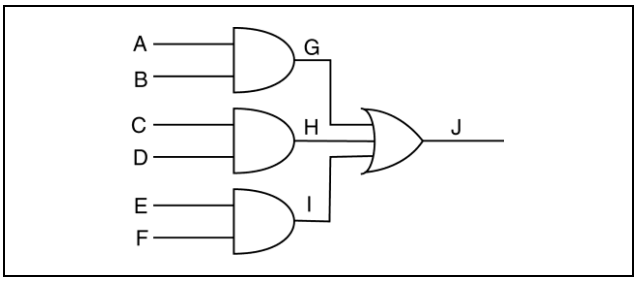


Figure 18.1: Some common circuit that is well known to "execute" parallel operations.

In computer programming languages, the instructions that form programs are usually targeted for execution on a single processor. In this case, there is no option for parallelism[213]. The computer program is generally a fixed set of instructions intended for execution on a pre-designed chunk of hardware. In contrast, VHDL modeling typically models digital circuits such as those in computers. Once again, the differences between VHDL and computer programming language are both significant and distinct. Once again, you need to keep these differences in mind as you learn more about VHDL and start using VHDL to model digital circuits of increasing complexity.

Here's the trick. Since most of us are human[214], we're only capable of reading one line of text at a time. This same limitation follows us around when we try to write some text, not to mention entering some lines of text into a file to be stored on a computer. So how then are we going to use text to describe some circuit that is inherently parallel using lines of text? We didn't have this problem when discussing something inherently sequential such as standard algorithmic programming by a higher level language. When writing code using an algorithmic programming language, there is generally only one processing element to do all the work. The processing element generally executes one instruction at a time and does so in a sequential manner that is determined by the order of appearance of instructions in the program that is running.

The VHDL programming paradigm built around the concept of expressing parallelism and concurrency with its textual descriptions of circuits. The heart of VHDL programming is the *concurrent statement*. Though these statements appear similar to the statements in algorithmic languages, they are significantly different because the VHDL statements, by definition, express concurrency. In other words, individual VHDL statements are ***interpreted*** as being concurrent.

Figure 18.2 lists the code that implements the circuit shown in Figure 18.1. This code shows four *concurrent signal assignment* statements. The "<=" construct is referred to as a *signal assignment operator*, which your currently familiar with from previous chapters. The reality is that we can't write these four statements at the same time but we can interpret these statements as actions that occur at the same time, or better stated, actions that occur *concurrently*. Once again, the concept of concurrency is a key concept in VHDL so keep this in mind anytime you are dealing with VHDL code. If you feel that algorithmic style of thought creeping into your soul, try to snap out of it quickly. A more complete discussion of concurrent signal assignment appears in the following section.

Because of the concurrent nature of VHDL statements, the three chunks of code appearing in Figure 18.3 are equivalent to the code shown in Figure 18.2. Once again, since the statements are interpreted as occurring concurrently, the order that these statements appear in your VHDL source code makes no

[213] The processor itself most likely exploits some form of parallelism, but that's not the form of parallelism we're referring to here. This notion of parallelism is considered an advanced concept and you'll hopefully learn about it later.
[214] Though most academic personnel don't qualify for this descriptive label…

difference. Generally speaking, it would be a better idea to describe the circuit as shown in Figure 18.2 since it somewhat reflects somewhat of a natural organization of statements.

```
G <= A AND B;
H <= C AND D;
I <= E AND F;
J <= G OR H OR I;
```

Figure 18.2: VHDL code that describes the circuit of Figure 18.1.

```
G <= A AND B;
J <= G OR H OR I;
H <= C AND D;
I <= E AND F;
```

```
G <= A AND B;
I <= E AND F;
J <= G OR H OR I;
H <= C AND D;
```

```
J <= G OR H OR I;
G <= A AND B;
H <= C AND D;
I <= E AND F;
```

Figure 18.3: Three equivalent sets of statements describing the circuit shown in Figure 18.1.

Figure 18.4 shows some "C" code that looks similar to the code listed in Figure 18.2. In this case, the logic functions are replaced with addition operators and the signal assignment operators are replaced by assignment operators. The statements in this code fragment execute sequentially as opposed to concurrently as is the case for the VHDL code of Figure 18.2. In other words, the statements shown in Figure 18.4 are intended for execution by some type of processing element. This processing element executes one statement and then moves onto the next statement. Once again, although the two snippets of code appear somewhat similar, they have completely different meanings. Keep in mind that if you were to rearrange the statements shown in Figure 18.4, they would have a completely different meaning: the order of statement appearance is massively important for higher-level computer languages.

```
A = A + B;
G = A + B;
H = C + D;
I = E + F;
J = G + H + I;
```

Figure 18.4: Higher-level language code similar to the VHDL code in Figure 18.2.

18.3.2 The Signal Assignment Operator: "<="

Algorithmic programming languages always have some type of *assignment operator*. In "C", this is the well known "=" sign. In programming languages, the assignment operator signifies a transfer of data from the right side of the operator to the left side. VHDL uses two consecutive characters to represent the assignment operator: "<=". VHDL uses this combination because it is different from the assignment operators in most other common algorithmic programming languages. The operator is known as a *signal assignment operator* to highlight its true purpose[215]. The signal assignment operator specifies a relationship between signals. In other words, the signal on the left side of the signal assignment operator is dependent upon the signals on the right side of the operator. In yet other words, the value of the

[215] This operator is actually borrowed from register transfer notation, a higher-level design approach for computer-oriented circuitry.

signal (or expression) on the right side of the signal assignment operator is assigned to the value on the left side of the operator.

With these new insights into VHDL, you should be able to understand the code of Figure 18.2 and its relationship to Figure 18.1. The statement "`G <= A AND B;`" indicates that the value of the signal named "G" represents an ANDing of the signals A and B. The similar statement in written in an algorithmic programming language, "`G = A + B;`" indicates that the value represented by variable A is added to the value represented by variable B and the result is then represented by variable G. In an algorithmic programming language, the values of G, A, and B are generally representative of memory locations somewhere in the associated hardware. The distinction between these two types of statements in VHDL and higher-level languages will become clearer the more you work with it.

VHDL has four types of concurrent statements. We've already briefly discussed the concurrent signal assignment statement and we'll soon examine it in greater detail and put it in context of actual circuits. The three other types of concurrent statements of immediate interest to us are *process statements*, *conditional signal assignments*, and *selected signal assignments*.

In essence, the four types of statements represent tools that you can use to implement digital circuits. You'll soon be discovering the versatility of these statements and applying them liberally. Unfortunately, this versatility effectively adds a fair amount of steepness to the learning curve. As you know from your experience in other programming languages, there are always multiple ways to do the same things. Stated differently, several seemingly different pieces of code can actually produce the same result. The same is true for VHDL code: several considerably different pieces of VHDL code an actually generate the exact same or functionally equivalent digital circuit. Keep this in mind when you look at any of the examples provided in this text.

Any VHDL code used to solve a particular problem is more than likely one of many possible solutions to that problem. Some of the VHDL models presented in this text are presented to show that something "can" be done a certain way, but that does not necessarily mean they "should" be done that way. It's good that your circuit works, but it's even better if your circuit both works and was based on a great looking piece of VHDL code. Always strive for clarity when using VHDL to model digital circuits.

18.4 Signal Assignment Statements in VHDL

As mentioned previously, there are four different types of signal assignment statements in VHDL. In addition, because we're discussing VHDL, each of these signal assignment statements is considered concurrent. There are some advantages to using one type of these statements over another, but these differences won't be discussed much in this chapter. This chapter sort of tosses out all the information but does not do so in the context of useful circuitry. The next chapters discuss some useful digital circuitry; by the time you get to those circuits, we'll be able to present them by using different forms of signal assignment statements.

18.4.1 Concurrent Signal Assignment Statements

The examples presented in earlier VHDL problems in this text used concurrent signal assignment statements. This section presents a formal introduction to concurrent signal assignment despite the fact that most of the information presented is not new.

Figure 18.5 shows the general form of a concurrent signal assignment statement. In this case, *target* is a signal that receives the values of the evaluated *expression*. An expression is defined by a constant, a signal, or a set of operators that operate on other signals and evaluate to some value. The target is generally considered an output while the expression is generally considered the input or a combination

of inputs. Most of the concurrent signal assignment statements you've used thus far were examples of expressions; more examples of expressions are provided in the examples that follow.

```
target <= expression;
```

Figure 18.5: Syntax for the concurrent signal assignment statement.

Example 18-1

Write the VHDL code that models a three input NAND gate. Use A, B, and C for the three input signal names; use F for the output signal name.

Solution: Even though this is a simple example, it's always good practice to draw a diagram of the circuit you're modeling. Furthermore, though we could draw a diagram showing the familiar symbol for the NAND gate, we'll choose to keep the diagram general and take the black box approach instead. Remember, the black box is a nice aid when it comes to writing the entity declaration. Figure 18.6 shows the dark box diagram for this example and while Figure 18.7 shows the associated VHDL model.

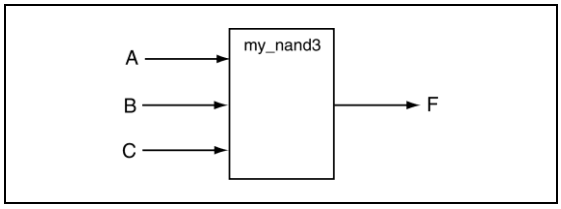


Figure 18.6: Black box diagram for Example 18-1.

```
entity my_nand3 is
    port ( A,B,C : in std_logic;
                F : out std_logic);
end my_nand3;

architecture exa_nand3 of my_nand3 is
begin
    F <=  NOT (A AND B AND C);
end exa_nand3;

architecture exb_nand3 of my_nand3 is
begin
  F <= A NAND B NAND C;
end exb_nand3;
```

Figure 18.7: Solution to Example 18-1

This example contains a few new ideas that are worth further mention.

- This example highlights the use of several logic operators. Some of the logic operators available in VHDL are **AND**, **OR**, **NAND**, **NOR**, **XOR**, and **XNOR**. The **NOT** operator is technically speaking not a logic gate but is also available. Moreover, these are *binary* logic operators in that they operate on the two values appearing on the left and right side of the operator. The **NOT** operator is a *unary* operator in that it only operates on the value appearing to the right of the operator. When you use the **NOT** operator, you should always use parenthesis.

- Two different architectures have been provided in this solution; they are both associated with the same entity. Note that both architectures reference the same entity declaration.

- The "stuff" between the **begin** and **end** keywords is indented. Proper indentation is an unspeakably good practice as it quickly transfers information to the reader.

Example 18-1 demonstrates the use of the concurrent signal assignment (CSA) statement in an actual VHDL model. However, since there is only one CSA statement, the concept of concurrency is not readily apparent. The idea behind any concurrent statement in VHDL is that the output may change anytime one of the input signals changes. In other words, the output is re-evaluated anytime a signal in the input expression changes. If this re-evaluation causes the output value to change, the change occurs immediately[216]. This is a key concept in developing a true understanding of VHDL (so you may want to read that sentence a few more times). The following examples more clearly define idea of concurrency.

Example 18-2

Write VHDL code to implement the function expressed in the following truth table.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: While you might consider the first step in this solution is to reduce the given function, it's really not. The reality is that you never should reduce a function if you're going to model it with VHDL; doing so defeats the purpose of using VHDL. The problem is that reducing a function by hand has severe limitations[217]; the best approach is to allow the VHDL synthesizer to do the optimization work for you. Keep in mind that the approach we'll take to solving this problem is not optimal and is more for academic purposes. We'll present the preferred approach later in this chapter.

Figure 18.8 shows the black box diagram for this example while Figure 18.9 shows the associated VHDL model. One important thing to note from this VHDL model is that it is visually appealing. What makes it so nice to look at is the fact that we've constructively used the fact that VHDL ignores white

[216] In the ideal case; non-idealized models include propagation delays (which we'll talk about in later chapters).
[217] Tools such as K-maps are limited in the number of independent variables. Moreover, anytime you try to do the reduction yourself, you're either going to make a mistake in the reduction process. Don't try this at home.

space and lined up stuff to make the code visually pleasing for the human element. You must continually strive to make your VHDL code as neat and organized as possible. One last comment regarding the solution shown in Figure 18.9 is that it does not reflect the concept of concurrency since the architecture only contains one concurrent signal assignment statement.

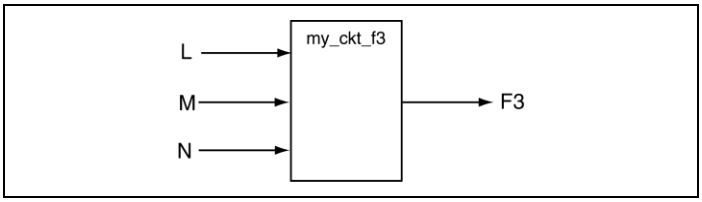


Figure 18.8: Black box diagram for Example 18-2.

```
-- non-reduced implementation of F3
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
              F3 : out std_logic);
end my_ckt_f3;

architecture f3_1 of my_ckt_f3 is
begin
   F3 <= ( (not L) AND (not M) AND N) OR
         (L AND M AND (not N)) OR
         (L AND M AND N);
end f3_1;
```

Figure 18.9: The non-reduced solution to Example 18-2.

Although the function itself is somewhat useless, there is some worthwhile information associated with other approaches to modeling this function. For the other models to this example, let's deal with a reduced version of the function; Equation 18-1 shows the reduced version of the function given in this example. Figure 18.10 shows an alternate solution to Example 18-2; unfortunately, this solution once again does not demonstrate the concept of concurrency.

$$F3 = \overline{L}\,\overline{M}N + LM$$

Equation 18-1

```
architecture f3_2 of my_ckt_f3 is
begin
   F3 <= ((NOT L) AND (NOT M) AND N) OR (L AND M);
end f3_2;
```

Figure 18.10: Alternate solution to Example 18-2.

Figure 18.11 shows one final alternative solution to Example 18-2. This final solution once again shows an important feature in VHDL modeling. The solution shown in Figure 18.11 uses some special statements in order to implement the circuit. These special statements are used to provide what is often referred to as *intermediate results*. This approach is analogous to declaring extra variables in an

algorithmic programming language to be used for specifically for storing intermediate results. The need for intermediate results in VHDL is provided by the declaration of extra signal values which are often referred to *intermediate signals*.

Note in Figure 18.11 that the declaration of the intermediate signals is similar to the port declarations appearing in the entity declaration except the mode specification is missing. The intermediate signals must be declared within the body of the architecture because the associated signals have no linkage to the outside world and thus do not appear in the entity declaration. Specifically, intermediate signal declaration appears in the declarative region of the architecture body. In other words, the outside world does not need to know about these signals so they only need appear in the architecture.

The cool thing about the solution shown in Figure 18.11 is the fact there are three distinct concurrent signal assignment statements. Despite the fact that these three statements are listed sequentially, the VHDL synthesizer interprets these statements as being concurrent. Since the statements are interpreted as being concurrent, the order of their appearance in the architecture statement region has not effect on the overall function of the VHDL model. To drive this point home, Figure 18.12 shows yet another functionally equivalent solution to Example 18-2. The solutions shown in Figure 18.11 and Figure 18.12 only differ by the ordering of the concurrent signal assignments. However, since the ordering of concurrent statements makes no difference in VHDL, the solutions are equivalent.

```
architecture f3_3 of my_ckt_f3 is
   signal s_a1,s_a2 : std_logic; -- intermediate signals
begin
   s_a1 <= ((NOT L) AND (NOT M) AND N);
   s_a2 <= L AND M;
   F3 <= s_a1 OR s_a2;
end f3_3;
```

Figure 18.11: Alternative but functionally equivalent architecture for Example 18-2.

```
architecture f3_4 of my_ckt_f3 is
   signal s_a1,s_a2 : std_logic; -- intermediate signals
begin
   F3 <= s_a1 OR s_a2;
   s_a2 <= L AND M;
   s_a1 <= ((NOT L) AND (NOT M) AND N);
end f3_4;
```

Figure 18.12: Yet another functionally equivalent architecture for Example 18-2.

Although the approach of using intermediate signals is not mandatory for this example, their use brings up some good points. First, the use of intermediate signals is the norm for most VHDL models. The use of intermediate signals was option in this example because the example was modeling a relatively simple circuit. As circuits become more complex, there are many occasions where you must use intermediate signals. VHDL structural modeling is an excellent example of where intermediate signals are required. Secondly, using intermediate signals is technique that you'll often need to use in your VHDL models. The thought here is that you're trying to describe a digital circuit using a textual description language: you'll often need to use intermediate signals to accomplish your goal of modeling the circuit.

The important idea regarding the use of intermediate signals is that they allow you to more easily model digital circuits but do not make the generated hardware more complicated[218]. The tendency in using VHDL is to think that since there is more text written on your page, that the circuit you're describing and/or the resulting hardware is larger or more complex. This is simply not true[219]. The notion of intermediate signals do not always support the notion of hardware in a synthesized circuit. What you'll see many times is that intermediate signals provide a mechanism that supports the textual description of a circuit using VHDL.

Generally speaking, it's the VHDL synthesis tools that have the final say in the size of your final design. The main theme of VHDL is that you should use the VHDL tools at your disposal in order to model your circuits in the simplest possible way. Simple circuits have a higher probability of someone understanding them and actually working. Most importantly, the overall complexity of a given VHDL model does not necessarily relate to the length of the VHDL code describing it[220].

Finally, the ease of the solution for Example 18-2 made the example trivial because the problem was not overly complicated. The point is that concurrent signal assignment statements are useful statements. However, as functions become more complicated (more inputs and outputs), an equation entry approach, particularly an equation that has been reduced by you, becomes pointless. Luckily, there are a few other types of concurrent constructs that mitigate this tedium.

18.4.2 Conditional Signal Assignment

Concurrent signal assignment statements discussed in the previous section have only one target and only one expression. The term *conditional signal assignment* describes statements that have only one target but can have more than one associated expression that can be assigned to the target where each of the multiple expressions is associated with a certain condition. The individual conditions are evaluated sequentially in the conditional signal assignment statement until the first condition evaluates as TRUE. In this case, the synthesizer evaluates the expression and assigns the result to the target. In a conditional signal assignment statement, only one assignment is applied per conditional signal assignment statement.

Figure 18.13 shows the syntax of the conditional signal assignment. The *target* in this case is the name of a signal. The *condition* is based upon the state of some other signal or signals in the given model. Note that there is only one signal assignment operator associated with the conditional signal assignment statement.

```
target <= expression when condition else
          expression when condition else
          expression;
```

Figure 18.13: The syntax for the conditional signal assignment statement.

[218] In actuality, intermediate signals are a message to the synthesizer to make a "connection" between two items; this connection does not necessary correspond to a physical piece of hardware.
[219] Which is yet another reason you should strive to make your VHDL models readable with the liberal use of white space such as blank lines and meaningful comments.
[220] But it is a well-known universal constant that only highly intelligent digital designers can consistently generate highly understandable VHDL models.

The conditional signal assignment statement is probably easiest to understand in the context of a circuit. For our first example, let's simply redo the Example 18-2 using conditional signal assignment instead of concurrent signal assignment.

Example 18-3

Write VHDL code to implement the generic decoder expressed by the accompanying truth table. Use only conditional signal assignment statements in your VHDL code.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: The entity declaration does not change from Example 18-2 so the solution only needs a new architecture description. We re-listed the entity declaration here for your enjoyment. Figure 18.14 lists one possible solution using conditional signal assignment to this example.

```
-- the same entity declaration as used previously
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
              F3 : out std_logic);
end my_ckt_f3;

architecture f3_5 of my_ckt_f3 is
begin
   F3 <= '1' when (L = '0' AND M = '0' AND N = '1') else -- m1
         '1' when (L = '1' AND M = '1' AND N = '0') else -- m6
         '1' when (L = '1' AND M = '1' AND N = '1') else -- m7
         '0';
end f3_5;
```

Figure 18.14: Solution to Example 18-2.

There are a couple of interesting points to note about this solution shown in Figure 18.14.

- It's not much of an improvement over the VHDL code written using only concurrent signal assignment statements. In fact, it looks a bit less efficient in terms of the amount of code and general understandability. The important thing to note is that we modeled the function without first reducing it. Unfortunately, it required a long time to write all the text.

- If you look carefully at this code and notice that there is in fact one target and a bunch of expressions and conditions. The associated expressions are the single digits surrounded by single quotes; the associated conditions follow the **when** keyword. In other words, there is only one signal assignment operator used for each conditional signal assignment statement.

- The conditional signal assignment statement evaluates the conditions sequentially. Once one statement is evaluates are true, the associated expression is assigned to the target and none of the other expression/condition pairs are evaluated. However, despite this order of appearance characteristic, the conditional signal assignment statement is a true concurrent statement[221].

- The last expression in the signal assignment statement is the catch-all condition. If none of the conditions listed above the final expression evaluate as TRUE, the last expression is assigned to the target. In other words, the "else" clause assures that some value will always be assigned to F3 when the conditional signal assignment statement is processed. Unless you have some great reason not to, make sure you include an else in your conditional signal assignment statements. This is a massively important point that we'll come back to in later chapters.

- The solution uses *relational operators*. There are actually six different relational operators available in VHDL. Two of the more common relational operators (as opposed to assignment operators) are the "**=**" and "**/=**" operators which are the "is equal to" and "is not equal to" operators, respectively.

- The choice of listing the minterms associated with the function used in the solution was arbitrary. The solution could have just as easily worked with maxterms. To drive home this point, Figure 18.15 shows an alternative but functionally equivalent solution to this example. Note that Figure 18.15 shows an alternative solution that requires more code than the solution of Figure 18.14.

```
-- the maxterm approach to this problem
architecture f3_6 of my_ckt_f3 is
begin
   F3 <= '0' when (L = '0' AND M = '0' AND N = '0') else -- M0
         '0' when (L = '0' AND M = '1' AND N = '0') else -- M2
         '0' when (L = '0' AND M = '1' AND N = '1') else -- M3
         '0' when (L = '1' AND M = '0' AND N = '0') else -- M4
         '0' when (L = '1' AND M = '0' AND N = '1') else -- M5
         '1';
end f3_6;
```

Figure 18.15: Alternate solution to Example 18-3.

18.4.3 Selected Signal Assignment

Selective signal assignment statements are the third form of concurrent statements that we'll examine. As with conditional signal assignment statements, selective signal assignment statements only utilize one signal assignment operator. Selective signal assignment statements differ from conditional assignment statements in that the statement bases its assignment on the evaluation of a single expression as opposed to many different expressions. Figure 18.16 shows the syntax for the selected signal assignment statement.

[221] No joke: this is a complicated notion and is without doubt the most complicated aspect of VHDL. The sooner you can wrap some dendrites around this, the better VHDL models you'll be writing.

```
with choose_expression select
   target <= {expression when choices, }
             expression when choices;
```

Figure 18.16: Syntax for the selected signal assignment statement.

Example 18-4

Write VHDL code to implement the generic decoder expressed by the accompanying truth table. Use only selective signal assignment statements in your VHDL code.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: This is yet another version of the my_ckt_f3 example originally appearing in Example 18-2. Figure 18.17 shows the full solution to Example 18-4. Some highly interesting comments regarding the solution are soon to follow.

```
-- the same entity declaration as used previously
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
              F3 : out std_logic);
end my_ckt_f3;


architecture f3_7 of my_ckt_f3 is
begin
   with ((L = '0' and M = '0' and N = '1')  or  -- m1
         (L = '1' and M = '1' and N = '0')  or  -- m6
         (L = '1' and M = '1' and N = '1')) or  -- m7
   select

   F3 <= '1' when '1',
         '0' when '0',
         '0' when others;

end f3_7;
```

Figure 18.17: Solution to Example 18-4

Figure 18.17 shows a solution that is somewhat special because it represents poor VHDL modeling practice. Although this example models the function using selective signal assignment, the solution is not clear. This is an example of a bad choice of signal assignment statements. We'll show a

much more intelligent use of selective signal assignment later in this chapter. However, there are a few interesting things to note regarding this solution.

- There is only one signal assignment operator associated with the selective signal assignment statement. This makes intuitive sense in that only one assignment is made; the body of the selective signal assignment statement can be thought of as deciding the appropriate assignment based on the given choosing expression.

- The multiple sub-expressions in the main choosing expression are the confusing part of this problem. Generally speaking, the choosing expression for selective signal assignment statements is relatively simple and often times consist of just a single signal name.

- The selective signal assignment statement uses a "**when others**" clause as the final entry in the statement. In actuality, the middle clause ("'0' **when** '0'") could be removed from the solution without changing the meaning of the statement. In general, it is considered good VHDL programming practice to include all the *expected* cases in the selective signal assignment statement followed by the "when others" clause. Similar to the final **else** clause in the conditional signal assignment statement, the "when others" clause in selective signal assignment statements act as a "catch-all" statements. The ramifications of a catch-all statement are of considerable importance in VHDL[222]. In general, it is bad practice (and sometimes impossible) to include all the *possible* cases in the selective signal assignment statement.

One general rule of programming languages or hardware description languages is that if you find yourself going to a lot of trouble modeling your circuit, there probably some trick embedded in the language that you're not aware of and thus not using to simplify your model. In other words, there's always an easier way to model some troubling circuits using VHDL. Unfortunately, some of the easier ways are out of reach of the beginning VHDL designer.

18.4.4 The Process Statement

The process statement is the fourth and final concurrent statement we'll look at. To understand the process statement, we'll first examine the similarities between it and the concurrent signal assignment statement. Once you grasp these similarities, we'll start discussing the differences between the statements and of course work a few examples.

Figure 18.18 shows the general syntax for the process statement. The main thing to notice about this syntax is that the body of the process statement comprises of *sequential statements*. The main difference between concurrent signal assignment statements and process statements lies with these sequential statements. The verbage below discusses a few of the more interesting points regarding the process statement syntax.

```
label: process(sensitivity_list)
begin
   {sequential_statements}
end process label;
```

Figure 18.18: Syntax for the process statement.

[222] Don't worry too much about this now. We'll provide a full explanation in a later chapter. It has to do with the unintended consequence of generating latches (which is not necessarily good).

- As the definition in Figure 18.18 implies, all the statements that appear between the **begin** and **end** keywords are evaluated in a sequential manner. This is in stark contrast to concurrent statements in general which are evaluated concurrently. The reality is that the process statement is a concurrent statement that necessarily contains sequential statements. There are three main types of sequential statements, which we'll describe soon, but once again, let's stick to the similarities before we dive into the differences.

- The *label* listed in Figure 18.18 is optional but should always be included to promote the self-commenting of your VHDL code.

Example 18-5

Implement an XOR function using both concurrent signal assignment and a process statement. Feel free to use the XOR operator in your solution.

Solution: Although we should draw a diagram for this circuit, let's skip it just this one time. No one will know. Figure 18.19 shows an entity declaration for a XOR function. Figure 18.20 shows both a concurrent signal assignment and a process statement architecture for the entity of Figure 18.19. The main difference between the two architecture descriptions is the presence of the *process* statement in latter version.

- Recall that the concurrent signal assignment statement operates as follows. Since it is a concurrent statement, anytime there is a change in any of the signals listed on the right side of the signal assignment operator, the signal on the left side of the operator is re-evaluated. A similar mechanism exists for the process statement but you actually have more control compared to the concurrent signal assignment statements. For the process statement, any time there is a change in any signal in process *sensitivity list*, all of the sequential statements in the process are revaluated[223]. The signals in the sensitivity list control the evaluation of the process statement. These two approaches are effectively the same but the syntax is significantly different.

- So here's where it gets strange. Even though both of the architectures listed in Figure 18.20 have the exact same signal assignment statement (`F <= A XOR B;`), execution of the statement in the behavioral style architecture is controlled by what signals appear in the process sensitivity list. Anytime there is a change in signal A or signal B, the statement appearing in the concurrent signal assignment architecture is evaluated. This difference is significant in that the process sensitivity list allows you more degrees of control in modeling the final circuit[224].

```
entity my_xor is
    port ( A,B : in std_logic;
             F : out std_logic);
end my_xor;
```

Figure 18.19: Entity declaration for circuit performing XOR function.

[223] This is not exactly true, but good enough for now.

[224] It's definitely hard to see why more control would be an advantage with a circuit this simple. This will make more sense as you start modeling more complex circuits.

```
-- concurrent signal assignment
architecture my_xor_con_sig of my_xor is
begin
   F <= A XOR B;
end my_xor_con_sig;


-- process statement
architecture my_xor_process of my_xor is
begin
   xor_proc: process(A,B)
   begin
      F <= A XOR B;
   end process xor_proc;
end my_xor_process;
```

Figure 18.20: Concurrent signal assignment and process statement descriptions of exclusive OR function.

18.4.4.1 Sequential Statements

The term "sequential statement" is derived from the fact that the statements within the body of a process statement are interpreted in a sequential manner. Execution of the sequential statements (the statements appearing in the process body) is initiated when a change in any signal contained in the process sensitivity list occurs. Execution of statements within the process body continues until execution reaches the end of the process body[225].

The strangeness of process statements evokes a philosophical dilemma: the process statement is a concurrent statement yet it is comprised of sequential statements. This is actually a tough concept to grasp so it may take a while to gain a grasp of this concept due to the fact it seems like a contradiction. Keep in mind that you're using VHDL to model (or describe) a digital circuit and that VHDL is primarily a modeling tool. The key to understanding sequential evaluation of statements occurring in a concurrent statement is to accept the fact that the VHDL synthesizer is going to examine your VHDL model and attempt to generate a digital circuit from it. In addition, since the ins and outs of this interpretation are not always readily apparent, we'll take some implementation details for granted until the time comes when you really need to fully understand the process statement[226].

The temporary solution to not fully comprehending this distinction is between sequential statements and concurrent statements are to keep your process statements as simple as possible. There is a tendency with new VHDL designers is to use the process statement as a repository for a bunch of loosely related sequential statements. Although syntactically correct, the code is not understandable in the context of digital circuit generation. In order to avoid this dilemma, you should strive to keep your process statements simple and to the point. Divide your intended functionality into several different process statements that communicate with each other rather than attempting to stuff all of your code into one giant, complicated, bizarre, ugly, disgusting process statement. Remember, process statements are concurrent statements: they all execute concurrently; you must take advantage of concurrency in order to simplify your circuit descriptions.

[225] Process statements are massively versatile; this statement becomes less true as your VHDL models become more complex.

[226] As you go along in VHDL, you'll gain more and more knowledge regarding process statement; this chapter is not attempting to tell you everything.

There are three types of sequential statements that we'll be discussing. We'll not say too much about the first type though because we've already been sort of dealing with it in that it is identical to a concurrent signal assignment statement. The other two types of statements are the *if statement* and the *case statement*. The nice part about both the *if* statement and the *case* statement is that you've worked with similar statements before in algorithmic programming languages and the VHDL syntax for these statements is strikingly similar. But alas, keep in mind that you're not programming a computer; you're describing digital hardware.

18.4.4.2 Signal Assignment Statements

The sequential style of a signal assignment statement is syntactically equivalent to the concurrent signal assignment statement. Another way to look at it is that if a signal assignment statement appears inside of a process than it is a sequential statement; otherwise, it is a concurrent signal assignment statement. Once again, Figure 18.20 shows the similarities and differences between these two statements.

18.4.4.3 IF Statements

The *if* statement is used to create a branch in the execution flow of the sequential statements. Depending on the conditions listed in the body of the *if* statement, either the instructions associated with one or none of the branches is executed when the *if* statement is processed. Figure 18.21 shows the general form of the *if* statement.

```
if (condition) then
   { sequence of statements }
elsif (condition) then
   { sequence of statements }
else
   { sequence of statements }
end if;
```

Figure 18.21: Syntax for the *if* statement.

The concept of the *if* statement should be familiar to you in two regards. First, its form and function are similar to the *if*-genre of statements found in most algorithmic programming languages. The syntax, however, is a bit different. Secondly, the VHDL *if* statement is the sequential equivalent to the VHDL conditional signal assignment statement. These two statements essentially do the same thing but the *if* statement is a sequential statement found in a *process* body while the conditional signal assignment statement is one specific form of concurrent signal assignment. In other words, the *if* statement is a sequential statement version of a conditional signal assignment statement.

Yet again, there are a couple of interesting things to note about the listed syntax for the *if* statement shown in Figure 18.21.

- The parenthesis placed around the *condition* expression is optional. They should be included in most cases to increase the readability of the VHDL source code, particularly when you use complex conditional expressions.

- Each *if*-type statement contains an associated *then* keyword. The final *else* clause has no *then* keyword associated with it.

- As written in Figure 18.21, the *else* clause is a catch-all statement. If none of the previous conditions evaluate as true, then the *if* statement evaluates the sequence of statements associated with the final *else* clause. In other words, if an *else* clause is used in an *if* statement, every possible situation is covered and either one of the *if* clauses or the *else* clause will be evaluated when the process itself is evaluated

- The final *else* clause is optional. Not including the final *else* clause presents the possibility that none of the sequence of statements associated with the *if* statement will be evaluated. This has deep ramifications that we'll discuss later[227]. For now, make sure any time you use an *if* statement to always include an *else* clause (which acts as a catch-all statement).

Example 18-6

Write some VHDL code that implements the following function using an *if* statement.

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$

Solution: Although it is not directly stated in the problem description, the VHDL code for this solution utilizes a process statement because an *if* statement can only appear in VHDL source code inside the body of a process statement. This problem implements a simple function; we mentioned earlier that functions such as these are not overly popular when designing anything but simple digital circuits. Later chapters provide better examples of process statements (so try not to get too discouraged now). Figure 18.22 shows the VHDL model for the solution to this example. We've opted again to leave out the black box diagram in this case since the problem is relatively simple.

```
entity my_ex is
    port ( A,B,C : in std_logic;
           F_OUT : out std_logic);
end my_ex;

architecture fun_example of my_ex is
begin

   proc1: process(A,B,C)
   begin
      if (A = '1' and B = '0' and C = '0') then
         F_OUT <= '1';
      elsif (B = '1' and C = '1') then
         F_OUT <= '1';
      else
         F_OUT <= '0';
      end if;
   end process proc1;

end fun_example;
```

Figure 18.22: Solution to Example 18-6.

There is not too much new information presented in Example 18-6. One good thing worth knowing is that the conditions portions of the *if* clauses evaluate to a Boolean value (either *true* or *false*). Once the

[227] Once again, it's the unintended creation of a latch.

first *if* clause evaluates as true, the code performs the assignment associated with that *if* clause. The catch-all *else* statement assures an assignment is made each time one of the signals in the process sensitivity list changes.

Once again, using an *if* statement is probably not the optimal approach to implementing Boolean functions. These examples do, however, show an *if* statement in action. Just to drive the point further into the ground, Figure 18.23 shows an alternate architecture for Example 18-6. While the solution shown in Figure 18.23 is technically correct, it obviously more cluttered and more confusing than the solution shown in Figure 18.22[228].

```
architecture bad_example of my_ex is
begin

   proc1: process(A,B,C)
   begin
      if ((A = '0' and B = '0' and C = '0') or
          (B = '1' and C = '1')) then
         F_OUT <= '1';
      else
         F_OUT <= '0';
      end if;
   end process proc1;

end bad_example;
```

Figure 18.23: An alternate solution for Example 18-6.

18.4.4.4 Case Statements

The *case* statement is somewhat similar to the *if* statement in that a sequence of statements are executed if an associated expression evaluates as true. The *case* statement differs from the *if* statement in that the resulting choice is made depending upon the value of the single control expression. Only one of the set of sequential statements executes for each execution of the *case* statement and is sole dependent upon the first *when* branch to evaluate as true. Figure 18.24 shows the syntax for the *case* statement. Following Figure 18.24 are some important case statement considerations.

```
case (expression) is
   when choices =>
       {sequential statements}
   when choices =>
       {sequential statements}
   when others =>
       {sequential statements}
end case;
```

Figure 18.24: Syntax for the *case* statement.

- The *case* statement is a different and more compact form of the *if* statement. It is not as functional, however, because all the choices are based on the same expression.

- The *case* statement is similar in both form and function to case or switch-type statements in algorithmic programming languages. Recall in that case-type statements typically remove the need in many cases to include a long list of if-else clauses.

[228] Cluttered is bad unless you're an academic administrator trying to justify your continued existence.

- The *case* statement is the sequential equivalent to the VHDL selective signal assignment statement. The case statement and selective signal assignment statements essentially have the same capabilities but the *case* statement is a sequential statement found in a *process* body while the selected signal assignment statement is one form of concurrent signal assignment.

- The **when others** clause is not required but should generally always be used unless you really know what you're doing. You know more what you're doing later, so for now, always use a **when others** clause. .

Example 18-7

Write a VHDL model that implements the following function using a *case* statement.

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$

Solution: This solution once again falls into the category of not being the best way to model a function using VHDL; it does illustrate another useful feature in the VHDL. The first part of this solution requires that we list the function as a sum of minterms, which requires us to multiply the non-minterm product term given in the example by 1. In this case, 1 is equivalent to $(A + \overline{A})$. Figure 18.25 shows this factoring operation.

$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC$$
$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + BC(A + \overline{A})$$
$$F\_OUT(A,B,C) = A\overline{B}\overline{C} + ABC + \overline{A}BC$$

Figure 18.25: Expanding the equation for Example 18-7.

Once you've listed the equation in standard minterm form, generating the VHDL model is based on the individual indexes associated with the minterms. Figure 18.26 shows the complete solution for Example 18-7. An interesting feature in this solution is the grouping of the three input signals, which allowed for the use of a *case* statement in the solution. This approach required the declaration of an intermediate signal which was appropriately labeled "ABC" in the spirit of self-commentation.

Once again, there are happier approaches to implementing functions but this example does highlight the need to be resourceful and creative when modeling digital circuits. The general rule you'll find to be true in VHDL is the fact that it is easier to do anything with bundled signals rather than work with the individual signals inside of the bundle.

```
entity my_example is
    port ( A,B,C : in std_logic;
           F_OUT : out std_logic);
end my_example;

architecture my_soln_exam of my_example is
   -- intermediate signal declaration
   signal ABC: std_logic_vector(2 downto 0);
begin

   ABC <= A & B & C; -- create bundle from signals

   my_proc: process (ABC)
   begin
      case ABC is
          when "100" =>  F_OUT <= '1';
          when "011" =>  F_OUT <= '1';
          when "111" =>  F_OUT <= '1';
          when others => F_OUT <= '0';
      end case;
   end process my_proc;

end my_soln_exam;
```

Figure 18.26: Solution to Example 18-7.

Another similar approach to Example 18-7 is to use the "don't care" feature built into VHDL. This allows the implementation of the logic function without having to massage the inputs. As with everything, if you have to modify the problem before you arrive at the solution, you stand a finite chance of creating an error that would not have been created had you taken a more clever approach.

Figure 18.27 shows an alternative solution (architecture only) for the Example 18-7. One definite drawback of using *don't care*s in your VHDL code is that some synthesizers and some simulators often times do not handle them correctly. Most VHDL-type textbooks recommend not to use don't care symbols in your VHDL models, so beware. All and all, it's not a good idea to use don't cares in VHDL. The VHDL model of Figure 18.26 is a better approach despite the fact that you had to massage the equation before you were able to model it in VHDL.

```
-- a solution that uses a don't care
architecture my_soln_exam2 of my_example is
   signal ABC: std_logic_vector(2 downto 0);
begin

   ABC <= A & B & C; -- create bundle from signals

   my_proc: process (ABC)
   begin
      case (ABC) is
          when "100" =>  F_OUT <= '1';
          when "-11" =>  F_OUT <= '1';
          when others => F_OUT <= '0';
      end case;
   end process my_proc;

end my_soln_exam2;
```

Figure 18.27: An alternate solution for Example 18-7.

18.4.4.5 Caveats Regarding Sequential Statements

As you begin to work with sequential statements, you tend to start getting the feeling that you're doing algorithmic programming using a higher-level language. This is because sequential statements have a similar look and feel to some of the similar programming constructs in higher-level languages. The bad part of this tendency is when your VHDL coding approach becomes similar to that of higher-level languages.

Using VHDL sequential statements as higher-level language programming constructs is a common error made by those new to VHDL. This being the case, it is appropriate to remind you once again that ***VHDL is not computer programming***: ***VHDL is a tool to describe hardware designs***. You are generally not implementing algorithms in VHDL; you're ***describing hardware***. It's a massively different paradigm and requires a different mindset.

If you attempt to implement a relatively large circuit using one process statement, you're going to fail at many levels. Although your code appears like it should work in terms of the provided statements, this is an illusion based on the fact that your mind is interpreting the statements in terms of a higher-level language. What's even worse is when your code simulates as you expect it to but the synthesized hardware does not work properly. In this case, you have no choice but to change your original VHDL model. A better approach is simply not to abuse VHDL sequential statements.

The reality is that circuit design methodology using VHDL is somewhat mysterious in that you are trusting that the VHDL synthesizer to magically know what you're trying to describe. If you don't understand the ins and outs of VHDL at a low level, you're circuit is not going to synthesize properly a good portion of the time. Small and simple VHDL models are easy to understand and generally straightforward to make work. The general VHDL programming approach is to break large and/or complex VHDL modules into small and simple sub-modules.

You should strive to keep your VHDL models simple, particularly your process statements. The best approach is to keep your process statements centered about a single purpose and have many process statements that communicate with each other[229]. The bad approach is to have one massive process statement that does everything for you. The magic of VHDL is that if you provide simple code to the synthesizer, it's more than likely going to provide you with a circuit that works and an implementation that is simple and eloquent. If you provide with synthesizer with complicated VHDL code, the circuit, may work as intended but it may not be efficient in both time and space considerations.

As opposed to higher-level languages where small amounts of code often translates to code of relatively high efficiency, efficiency in a VHDL model is obtainable by compact and simple partitioning of the VHDL code based on the underlying hardware constructs[230]. In other words, simple VHDL models are better and you'll achieve simplicity by proper partitioning and description of the model. So try to fight off the urge to impress your friends with the world's shortest VHDL model; your hardware friends will know better.

18.5 Standard Models in VHDL Architectures

As you remember, the VHDL architecture describes the functionality associated with a VHDL entity declaration. The architecture is comprised of two parts: the declarative region and followed by a

[229] Communication such referenced here is done with "signals" in the overall design.

[230] And, having a good synthesizer helps. Companies that provide free synthesis tools will not generally give you their latest and/or greatest: you have to pay for that.

collection of concurrent statements. We've studied four types of concurrent statements thus far: concurrent signal assignment, conditional signal assignment, selected signal assignment, and process statements. Concurrent statements pass information to other concurrent statements though the use of signals[231].

There are three main accepted approaches to writing VHDL architectures. These approaches are known as *dataflow style*, *behavioral style*, and *structural style* architectures. The standard approach to learning VHDL is to introduce each of these architectural styles individually and design a few circuits using that style. Although this approach is good from the standpoint of keeping things simple while immersed in the learning process, it's also somewhat misleading because more complicated VHDL circuits generally use a mixture of these three styles.

Because the digital circuits we've modeled up to this point are relatively simple, one of the three model styles could easily describe our VHDL models. As our circuits become more complex, most VHDL models employ some type of structural modeling. In other words, structural modeling supports the interconnection of black boxes but does not have the ability to describe the logic functions used to model the circuit operation. For this reason, structural modeling is less of a modeling style and more of an approach for interfacing previously designed modules.

In the end, the concept of using either a dataflow, behavioral, or structural approach to VHDL modeling is somewhat of a pointless matter. The reality is that you'll find yourself "doing what needs to be done" in order to model a circuit. As you gain experience modeling digital circuits with VHDL, you simply don't put much thought into the style of architecture you're using. The most important factor in VHDL is to make your models as simple as possible. Simple models are more robust, require less testing, and are easier to reuse. Moreover, the better approach is always to ensure that your complex VHDL models are comprised of a collection of relatively simple sub-modules. But then again, the terms presented in this section are somewhat standard in the world of VHDL so you really need to be aware of them.

18.5.1 VHDL Dataflow Style Architecture

A "dataflow style architecture", or "dataflow model" specifies a circuit as a concurrent representation of the flow of data through the circuit. Dataflow models describe circuits by showing the input and output relationships using the various built-in components of the VHDL language. The built-in components of VHDL include operators such as AND, OR, XOR, etc. The three forms of concurrent statements we've talked about up until now (concurrent signal assignment, conditional signal assignment, and selective signal assignment) are all statements that are found in dataflow style architectures. In other words, if you exclusively use concurrent, conditional, and selective signal assignment statements in your VHDL models, you are using a dataflow model.

If you were to re-examine some of the examples we've done so far, you can in fact sort of see how the data "flows" through the circuit. To put this in other terms, if you have a working knowledge of digital logic, it's fairly straight-forward to imagine the underlying circuitry in terms of standard logic gates. These signal assignment statements effectively describe how the data flows from the signals on the right side of the assignment operator (<=) to the signal on the left side of the operator.

The dataflow style of architecture has its strong points and weak points. It is good that you can see the flow of data in the circuit by examining the VHDL code. The dataflow models also allow you to make an intelligent guess as to how the actual logic will appear should you decide to synthesize the circuit. Dataflow modeling works fine for small and relatively simple circuits. However, as circuits become more complicated, it is usually advantageous for many reasons to switch to another modeling style.

[231] Also included in architecture bodies are component instantiations. There are not concurrent statements, though it is something comforting to think of them as such.

18.5.2 VHDL Behavior Style Architecture

In comparison to the dataflow style architecture, the "behavioral style architecture", or "behavioral model", does not necessarily provide details as to how the design is implemented or synthesized in actual hardware. VHDL code written in a behavioral style does not necessarily reflect how the synthesizer implements the circuit. Instead, the behavioral style models how the circuit outputs will react to (or "behave") the circuit inputs or the sequence of circuit inputs.

Whereas in dataflow modeling you somewhat needed to have a feel for the underlying logic in the circuit, behavioral models provide you with various tools to describe how the circuit will behave and leaves the implementation details up to the synthesis tool. In other words, *dataflow modeling describes how the circuit should look in terms of gates whereas behavioral modeling describes how the circuit should act*. For these reasons, behavioral modeling is higher-up on the circuit abstraction level as compared to dataflow models. In one sense, behavioral style modeling is the ultimate "black box" approach to designing circuits.

The heart of the behavioral style architecture is the *process* statement, which was the fourth type of concurrent statement that we'll discuss in this chapter. As you've seen, the process statement is significantly different from the other three concurrent statements in several ways. The major difference lies in the process statement's approach to concurrency with its notion of containing nothing but sequential statements, which is the major sticking point in learning to deal with process statements.

18.5.3 VHDL Structural Models: Not a Behavioral vs. Dataflow Argument

The third type of model in VHDL is the structural model. We've dealt with structural models in previous chapters so we'll not say too much about it here. The issue is that it seems that dataflow and behavioral models represent two approaches to the same thing. In contrast, structural models seem like a different beast[232].

Maybe I'm simply unclear on the concept regarding modeling. What I do know is that VHDL models can quickly become complex. When this occurs, you absolutely have to resort to representing the model in a hierarchical manner, which means a structural model in VHDL terms. While pure structural modeling is an issue of component declaration and instantiation of pre-defined modules, structural models can also be a mixture of components, dataflow models, and/or behavioral models. In addition, the modules that are instantiated as part of structural models can be implemented using some combination of these three model types.

In the end, you need to do whatever you need to do in order to model your digital circuits. Your goal is to make the models as simple as possible as this makes the synthesizer, the simulator, and any human reading your circuit much happier. Placing a label on a model in an attempt to classify it as something meaningful serves no purpose and has a tendency to confuse the matter.

18.5.4 Behavioral vs. Dataflow

[232] At least to me, anyway. I've included these terms in this chapter because you may actually run into them out there in digital design-land.

Often times when using VHDL, you'll find yourself faced with a small dilemma: should you use a dataflow or behavioral model? The answer to this question is not simple. The bottom line is that as you gain experience, you'll be able to answer this question without too much thought. In reality, when making this decision, you need to think about the ultimate goal of your VHDL model. In all likelihood, you're modeling a circuit using VHDL because that circuit will eventually be implemented in one form or another. The truth is that the VHDL synthesizer is the intermediary between your VHDL model and the final form of your circuit. This being the case, you need to defer to the characteristics of the synthesizer you're using.

Listed below are a few simple guidelines governing the issues of behavioral models vs. dataflow models. These items were gathered from personal experience and reading many books on VHDL. As you gather your own experience with VHDL, you'll be able to generate your own set of guidelines.

- The constructs of behavioral modeling allow you to describe circuits at a relatively high level of abstraction, which allows you to describe the operation of circuits without bogging yourself down with the low-level implementation details. Therefore, if you need to model a complex circuit or a complex behavior, your optimal choice is using a behavioral model. However, keep in mind that modeling at this high of level forces you to put a lot of faith into the VHDL synthesizer. This is sometimes less that a good decision. No matter what you do, make sure you at least have some remote vision of what hardware your circuit implementation should be using. There are not that many basic types of digital modules out there and even your most complex circuitry will be comprised of these modules.

- Although you don't know it yet, none of the circuits we've been designing have the ability to "memorize" things[233]. We'll get to this in later chapters, but for now, keep in mind that behavioral modeling is usually more useful when your circuits have memory elements.

- If you need to design a relatively simple circuit, you're often faced with the choice of behavioral vs. dataflow. The word on the street is that if you can model your circuits using a dataflow model, it sometimes has advantages during circuit synthesis. The "sometimes" in the previous sentence is based on the synthesizer characteristics and qualities. The "advantages" refers to the fact that sometimes using a dataflow model magically directs the synthesizer to generate a physically smaller circuit while exhibiting the desired functionality. In other words, if the synthesizer needs to thinks less, it often times thinks better[234].

- In the end, you should always use a dataflow model if you can do so without too much trouble. Don't go way out of your way to force a design to be a dataflow model, however: the benefits generally are not there.

18.6 Mealy's Third and Fourth Laws of Digital Design

As you progress in digital design, and particular, in the modeling of digital circuits using VHDL, there are a few key issues to never lose sight of. In general, the VHDL synthesizer is your friend; you therefore don't want to expect too much of it. The synthesizer's job is to take your VHDL model and translate that into a form that can be ultimately implemented as a real circuit. This is a big process as some piece of software (written by humans) must take a file of text and somehow create a working circuit out of that. In the end, it's not as easy as you would think and can sometimes create problems.

[233] This is a loose reference to the concept of memory and the hallmark of sequential circuits. We'll introduce these concepts in an upcoming chapter.
[234] I can personally relate to this.

The best way to avoid these types of problems is to follow Mealy's third and fourth laws of digital design.

> **Mealy's Third Law of Digital Design:** Don't rely on the VHDL synthesizer; create your VHDL models by having a remote vision of what underlying hardware should look like in terms of standard digital modules.

> **Mealy's Fourth Law of Digital Design**: Model circuits using many smaller sub-modules as opposed to fewer larger sub-modules. In this case, sub-modules should be true "modules" but can also be process statements.

The truth is that you still have not seen all the standard VHDL modules (we'll get to those in the upcoming chapters). What this third law is really stating is that you should always put a finite amount of faith in your own abilities to model circuits and not rely to heavily upon the magic of the VHDL synthesizer. What the fourth law is stating is that your circuits should be comprised primarily of smaller modules as they are easier to understand, write, and test. Keep in mind that even the most complex digital designs are decomposable into a set of standard digital modules.

18.7 Meaningful CSA Examples

Now that you've seen all the concurrent statements in VHDL, it's time to use them. This section contains a few examples of CSA usage in VHDL. These examples use the standard digital modules we previously introduced; better examples appear in later chapters where we'll introduce other standard digital modules.

> **Example 18-8: 8-Bit Comparator**
>
> Design a circuit that compares two 8-bit values A & B. This circuit should contain three outputs: EQ, GT, and LT. These inputs indicate whether A=B, A>B, or A<B, respectively. Describe your design using a VHDL behavioral model.

Solution: The comparator is a standard digital design circuit that we previously discussed. The new twist on this problem is the addition of the two non-equality outputs. The unstated purpose behind this problem is to show off the power of VHDL behavioral modeling, as you certainly would not want do complete this design with a truth table (BFD) or some sort of IMD. Figure 18.28 shows the high-level black box diagram for this problem while Figure 18.29 shows the associated VHDL entity.

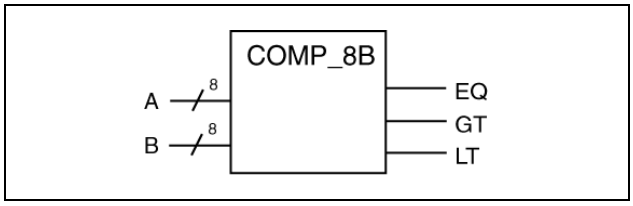


Figure 18.28: High-level black-box diagram for Error! Reference source not found..

```
entity COMP_8B is
    Port (      A,B : in  STD_LOGIC_VECTOR (3 downto 0);
           EQ,GT,LT : out  STD_LOGIC);
end COMP_8B;
```

Figure 18.29: High-level black-box diagram for Error! Reference source not found..

There many ways to do this problem despite the fact that the problem states you should use a behavioral model. We'll show you two ways as this presents an important point with using behavioral models. Figure 18.30 shows the first solution, which has a few new items in it.

- The process statement uses a less-than ("<") and a greater-than (">") operator in addition to the equality operator ("="). These operators are all binary operators provided as part of the VHDL language.

- In the process statement, a value is assigned to each of the three outputs in every case. This is a massively important VHDL consideration. The notion here is that in any given situation, you need to make sure to assign all the values that are associated with the process statement an output. In other words, if a process statement assigns values to three signals, you always must assign values to each of those three signals during each evaluation of the process. If you do not assign an output in this situation, the VHDL synthesizer assumes you want to "induce memory", which is actually something you don't want or need to do here[235].

- The if statement has an associated "else" statement. This is a catch-all statement which has a special purpose. Generally speaking, all your circuit models should contain a catch-all statement.

Figure 18.31 shows an alternative architectural, which solves Example 18-8. In this solution, the first thing that occurs during the evaluation of the process is that all of the signals that are eventually assigned in the process are assigned before the "if" statement is evaluated. This approach then removes the requirement that each clause of the "if" statement assigned a value to each output.

[235] This topic of memory is something we have not discussed yet in either terms of digital logic or VHDL. In short, VHDL has a special way of inducing memory; the approach taken in this problem of always making sure that you don't induce memory by making sure every output is always specified in each path taken by the process statement. We'll talk a lot more about this later.

```
architecture my_comp_8b_1 of COMP_8B is
begin

   comp8ba: process (A,B)
   begin
      if (A < B) then
         EQ <= '0'; GT <= '0'; LT <= '1';
      elsif (A > B) then
         EQ <= '0'; GT <= '1'; LT <= '0';
      else -- catch-all statement
         EQ <= '1'; GT <= '0'; LT <= '0';
      end if;
   end process;

end my_comp_8b_1;
```

Figure 18.30: One approach to the architecture solving Example 18-8; this approach assigns every output assigned by the process statement each time one output is assigned.

```
architecture my_comp_8b_2 of COMP_8B is
begin
   comp8b2: process (A,B)
   begin

      -- initialize all values assigned in process
      LT <= '0';  GT <= '0';  EQ <= '0';

      if (A < B) then
         LT <= '1';
      elsif (A > B) then
         GT <= '1';
      else -- catch-all statement
         EQ <= '1';
      end if;

   end process;
end my_comp_8b_2;
```

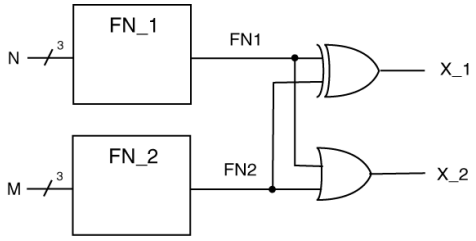
Figure 18.31: Another approach to modeling the solution to Example 18-8; this approach assigns a default value to every signal that is assigned in each clause of the "if" statement.

The two approaches to his problem illustrate an important point in VHDL. In both of these approaches, the main goal was to make sure that memory was not induced and we made sure that did not happen by making sure that we always assigned a value to every signal that is assigned a value somewhere in the process statement. The fact is that both of these solutions functionally equivalent despite the fact that there are apparently nine signal assignments in the first solution and six signal assignments in the second solution. In the end, the synthesizer would most likely generate the exact same hardware in each case. One other key important item to note is that both solutions contained "else" statements; these are the famous catch-all statements that your circuits should contain (unless they have a really good reason not to).

Example 18-9: Academic Exercise Problem

Implement the following circuit with a flat design using no more than four concurrent statements. The following equations describe the functions for the FN_1 and FN_2 black boxes.

$$FN\_1 = \sum(0,2) \qquad FN\_1 = \sum(0,1,2,4,6,7)$$



Solution: This is a typical academic exercise problem, but it does show several interesting items. The notes below describe a few new items presented by this problem. Figure 18.32 shows the black box model for this solution.

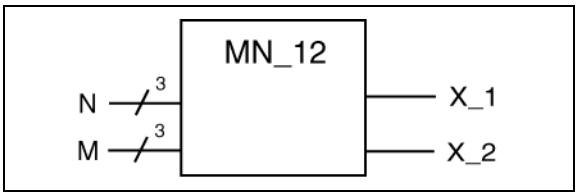


Figure 18.32: Black box diagram for Example 18-9 solution.

Figure 18.33 shows the final VHDL solution for Example 18-9. Here are a few interesting things to note about this solution beside the fact that this problem could have been done about a gazillion different ways.

- There are four concurrent statements used. The choice of concurrent statement was arbitrary, as the problem statement did not provide any specific direction.
- Since the four statements used in the solution are concurrent, the ordering of the statements has no affect on the solution.
- Since this model contains both a process statements and signal assignment statements, it can't be considered strictly a dataflow or behavioral model.
- The solution did not attempt to reduce the stated function. The one thing the solution did do was to implement one of the functions in POS form despite the fact the function was described in a compact SOP form.

```
entity MN_12 is
    Port (     M,N : in  STD_LOGIC_VECTOR (2 downto 0);
           X_1,X_2 : out  STD_LOGIC);
end MN_12;

architecture my_ckt of MN_12 is
   -- intermediate signals
   signal s_FN_1, s_FN_2 : std_logic;
begin

   f1: process(N)
   begin
      if (N = "000" OR N = "010") then
         s_FN_1 <= '1';
      else
         s_FN_1 <= '0';
      end if;
   end process;

   s_FN_2 <= ( (not M(2)) OR M(1) OR M(0))  AND  (M(2) OR (not M(1)) OR M(0));

   X_1 <= s_FN_1 XOR s_FN_2;

   X_2 <= s_FN_1 OR s_FN_2;

end my_ckt;
```

Figure 18.33: VHDL model for the solution of Example 18-9.

Example 18-10: Three-Value 10-Bit Comparator

Design a circuit that compares three 10-bit values. If all three 10-bit values are equivalent, the EQ3 output of the circuit will be a '1', otherwise it will be a '0'. Use only standard comparators in this design. Use any support logic you may require but minimize the amount of hardware used in this circuit. Use the modular design approach for this problem and provide both a block-level diagram and the VHDL structural model for the solution.

Solution: This is the same problem we solved in a previous chapter; the previous solution was presented as a structural model as requested. This problem requests the solution as a behavioral model which means we can exercise the power of behavioral modeling and make this a shorter problem than before. Figure 15.10 shows the black box model for the solution.

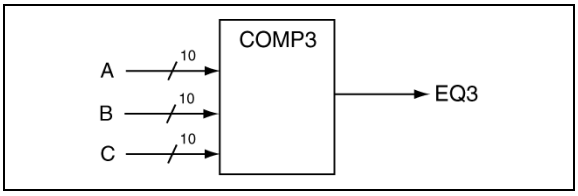


Figure 18.34: Black box diagram for Example 18-10 solution.

Using VHDL behavioral modeling, we can describe the circuit's operation with one process statement. Figure 15.11 shows the complete solution for this problem. Note that in this solution we used a complex conditional statement to describe the functionality of the circuit. You've seen the equality operators before, but the conditional statement used in this problem also contains an "AND" conditional. While this seems like this could be the familiar bit-oriented AND operator, it actually represents a logical operator, which officially operates on two Boolean-type values. In particular, the expression (A = B) and (B = C) are evaluated and return a "true" or a "false". The AND operator is overloaded and is evaluates Boolean values here and returns a Boolean value to the condition clause of the "if" statement.

This problem provides a rough feel for the power of behavioral modeling in VHDL. Note that modeling the solution to this problem in any other way would have required much more work and would have been less straight-forward.

```
entity COMP3 is
    Port ( A,B,C : in  STD_LOGIC_VECTOR (7 downto 0);
           EQ : out  STD_LOGIC);
end COMP3;

architecture my_comp3 of COMP3 is
begin

   comp: process (A,B)
   begin
      if (A = B AND B = C) then
         EQ <= '1';
      else
         EQ <= '0';
      end if;
   end process;

end my_comp3;
```

Figure 18.35: The final VHDL model for Example 18-10.

Example 18-11: In the Dark Car Alarm

Design a digital circuit that will control the unlocking of a car door. The car door has four push-buttons that control the unlocking mechanism of the car door. In most combination lock-type problems such as this one, the user must have all four of the buttons in the correct position (thus inputting the correct combination).But in this design, only three of the four buttons need to be in the correct position in order to unlock the door.

Solution: Once again, the first step is generating a block diagram of the final circuit. This diagram simply lists the inputs and outputs to the circuit; the rest of the solution involves deriving a relationship between the output and inputs in such a way as to solve the given problem. The UNLOCK output is considered a signal that is directed to some lock mechanism responsible for locking and unlocking the door. The idea here is that when we assign the UNLOCK signal a '1', the door unlocks; when we assign the UNLOCK signal a '0', the door locks.

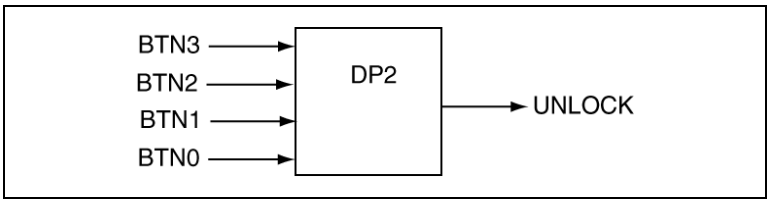


Figure 18.36: The black-box diagram of the final circuit.

The key to understanding this problem is knowing that somewhere inside of this circuitry, the code that unlocks the door has been *hard-coded* into the circuitry. In order for the door to unlock, the code the user inputs must be the same was the code that is hard-coded into the circuitry. The catch in this problem is that the user only needs to have three of the bits of the code correct in order to open the lock. A 4-bit comparator would have been adequate for this problem had it been the case where the user needed all of the buttons correct in order to unlock the door.

The solution to this problem is similar to a comparator. Instead of an AND gate on the output of the comparator, this circuit must add some special logic. We'll divide this problem into two sections as follows: the LOGIC_A and the LOGIC_B block as shown in Figure 18.37.

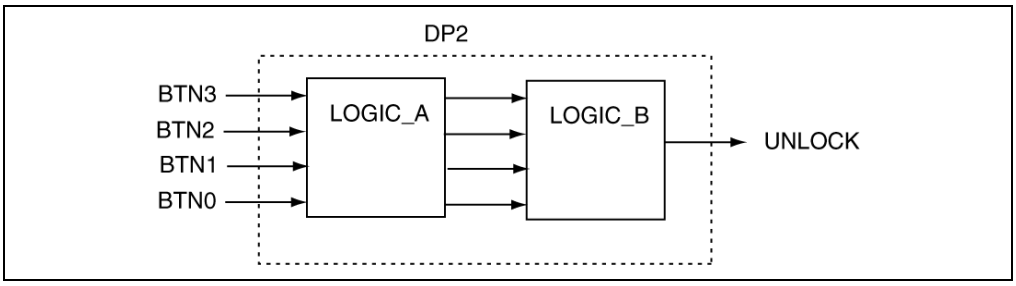


Figure 18.37: Lower-level block diagram of circuit solution.

The LOGIC_A block is similar to the comparator. In this case, the circuit compares each of the external button outputs to the internal preset combination to the lock. Figure 18.38 shows a diagram of this circuit. Note that each of the EXNOR gates simply compares the outputs of each of the buttons which is the expected comparator action. The dotted lines show that the comboX values are internal to the circuit.

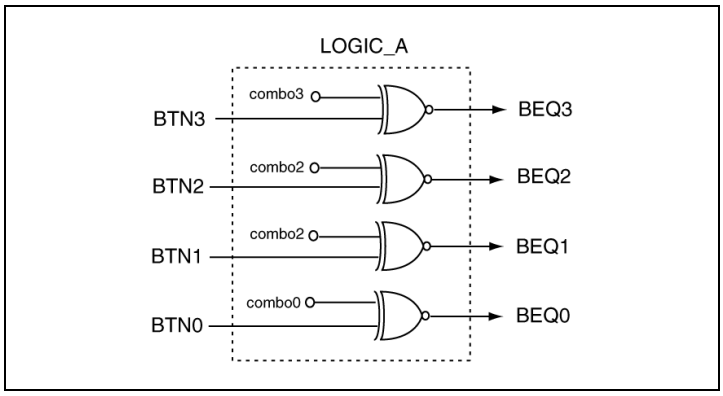


Figure 18.38: A detailed look at the LOGIC_A block.

The LOGIC_B block is a circuit that has an asserted output when at least three of the inputs are in a '1' state. Figure 18.39 shows a truth table that models the logic for the UNLOCK input. Equation 18-2 shows a reduced expression implementing this functionality. Finally, Equation 18-2 shows an expression describing the final solution. Figure 18.40 provides a VHDL model for the LOGIC_B block.

BEQ3	BEQ2	BEQ1	BEQ0	UNLOCK
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Figure 18.39: The truth table modeling the UNLOCK output.

$$F = (BQ3 \cdot BQ2 \cdot BQ0) + (BQ3 \cdot BQ2 \cdot BQ1) + (BQ3 \cdot BQ1 \cdot BQ0) + (BQ2 \cdot BQ1 \cdot BQ0)$$

Equation 18-2: Boolean expression describing the solution circuitry.

```
--------------------------------------------------------------------------
-- A standard decoder-type function implementation
-- for the LOGIC_B block.
--------------------------------------------------------------------------
entity my_ex is
    port (    BTN : in std_logic_vector(3 downto 0);
           UNLOCK : out std_logic);
end my_ex;

architecture my_ex_arch of my_ex is
begin
   with BTN select
   UNLOCK <= '1'  when "0111" | "1011" | "1101" | "1110" | "1111",
             '0'  when others;
end my_ex_arch;
```

Figure 18.40: The VHDL model for Error! Reference source not found.**.**

Chapter Summary

- The entity/architecture pair form the interface and functional description, respectively, of how a digital circuit behaves.

- The main design consideration in VHDL modeling supports the fact that digital circuits operate in parallel. In other words, the various design units in a digital design process and store information independently of each other. The various design units communicate with each other through the use of signals, often referred to as intermediate signals.

- Signals that are declared as OUTs in the entity declaration cannot appear on the right side of a signal assignment operator. Intermediate signals are similar to signals declared in entities except that they contain no mode specifier. Intermediate signals are declared in the declarative region of the architecture body.

- The four major signal assignment types in VHDL are concurrent signal assignment, conditional signal assignment, selective signal assignment, and process statements. These statements are referred to as concurrent statements in that they are interpreted as acting in parallel (concurrently) to all other concurrent statements.

- Concurrent signal assignment, conditional signal assignment, and signal assignment statements are considered low-level statements. These statement types are primarily designed to model digital logic at a low level and are associated with "dataflow" models.

- The process statement is primarily used to describe the behavior of circuits on a high level (higher than then other three types of concurrent statements). The body of the process statement can contain any number of sequential statements. There are three types of sequential statements: signal assignment statements, *if* statements, and *case* statements. Process statements are associated with behavioral models.

- There are three major model types in VHDL: 1) dataflow, 2) behavioral, and 3) structural models. Most complex designs use a combination of these model types. Use of concurrent signal assignment, conditional signal assignment, and selective assignment statements indicate you are using a dataflow model. Use of a process statement indicates that you're using a behavioral model. Structural models can employ both dataflow and behavioral model types. Structural modeling is different from the other two model types in that it is used to gather design units and glue them together.

- In general, dataflow modeling describes how the circuit should look in terms of gates whereas behavioral modeling describes how the circuit should act and provides no information regarding implementation details.

- The ***if*** statement has a direct analogy to the conditional signal assignment statement used in dataflow modeling. The ***if*** statement is a sequential statement while the conditional signal assignment statement is a concurrent statement.

- The ***case*** statement has a direct analogy to the selective signal assignment statement used in dataflow modeling. The ***case*** statement is a sequential statement while the selective signal assignment statement is a concurrent statement.

- Both the ***case*** statement and the ***if*** statement can be nested. Concurrent, conditional, and selective signal assignment statements cannot be nested.

- It is generally easier in any VHDL model to work with bundled signals rather than work with the individual signals inside of the bundle. Any time you can work with a bundle, you should work with a bundle.

- VHDL has the ability to use "don't cares" in various aspects of modeling. The general rule, however, is to never use "don't cares" as they have a tendency to confound the synthesizer and/or simulator when they are used.

- **Mealy's Third Law of Digital Design:** Don't rely on the VHDL synthesizer; create your VHDL models by having a remote vision of what underlying hardware should look like in terms of standard digital modules.

- **Mealy's Fourth Law of Digital Design**: Model circuit using many smaller sub-modules as opposed to fewer larger sub-modules. In this case, sub-modules can be either true "modules" or individual concurrent statements.

- The table on the following page is a summary of the information presented in this chapter. It is part of the well-known VHDL cheat sheet.

Concurrent Statements		Sequential Statements
Concurrent Signal Assignment (dataflow model)	⇔	**Signal Assignment**
target <= *expression*;		*target* <= *expression*;
A <= B **AND** C; DAT <= (D **AND** E) **OR** (F **AND** G);		A <= B **AND** C; DAT <= (D **AND** E) OR (F **AND** G);
Conditional Signal Assignment (dataflow model)	⇔	***if* statements**

target <= *expressn* **when** *condition* **else**
expressn **when** *condition* **else**
expressn;
		if (*condition*) **then**
		{ sequence of statements }
		elsif (*condition*) **then**
		{ sequence of statements }
		else --(the else is optional)
		{ sequence of statements }
		end if;

F3 <= '1' **when** (L='0' **AND** M='0') **else** '1' **when** (L='1' **AND** M='1') **else** '0';		**if** (SEL = "111") **then** F_CTRL <= D(7); **elsif** (SEL = "110") **then** F_CTRL <= D(6); **elsif** (SEL = "101") **then** F_CTRL <= D(1); **elsif** (SEL = "000") **then** F_CTRL <= D(0); **else** F_CTRL <= '0'; **end if**;
Selective Signal Assignment (dataflow model)	⇔	***case* statements**
with *chooser_expression* **select** target <= *expression* **when** *choices*, *expression* **when** *choices*;		**case** (*expression*) **is** **when** *choices* => {sequential statements} **when** *choices* => {sequential statements} **when others** => -- (optional) {sequential statements} **end case**;
with SEL **select** MX_OUT <= D3 **when** "11", D2 **when** "10", D1 **when** "01", D0 **when** "00", '0' **when others**;		**case** ABC **is** **when** "100" **=>** F_OUT **<=** '1'; **when** "011" **=>** F_OUT **<=** '1'; **when** "111" **=>** F_OUT **<=** '1'; **when others =>** F_OUT **<=** '0'; **end case**;
Process (behavioral model)		
opt_label: **process**(*sensitivity_list*) **begin** {*sequential_statements*} **end process** *opt_label*;		

proc1: **process**(A,B,C)
begin
if (A = '1' **and** B = '0') **then**
F_OUT **<=** '1';
elsif (B = '1' **and** C = '1') **then**
F_OUT **<=** '1';
else
F_OUT **<=** '0';
end if;
end process proc1;		

Chapter Exercises

1) Why is it a good idea to keep your VHDL models as simple as possible? Briefly describe.

2) Why is it a good idea to break keep individual concurrent statements as simple as possible? Briefly describe.

3) For the following function descriptions, write VHDL models that implement these functions using concurrent signal assignment.

(a) $F(A,B,C) = \sum(2,3,5)$

(b) $F(A,B,C) = m_2 + m_5$

(c) $F(A,B,C,D) = \sum(4,3,15)$

(d) $F(W,X,Y,Z) = \sum(12,14,15)$

(e) $F(A,B,C,D) = \overline{A}C\overline{D} + \overline{B}C + BC\overline{D}$

(f) $\overline{F}(A,B,C) = \sum(5,7)$

(g) $\overline{F}(W,X,Y) = \sum(2,3,4,7)$

(h) $\overline{F}(R,S,T) = \sum(0,1,4,7)$

(i) $F(A,B,C) = \prod(1,2)$

(j) $F(A,B,C) = M_1 \cdot M_6 \cdot M_7$

(k) $F(L,M,N) = \prod(1,6)$

(l) $F(W,X,Y,Z) = \prod(1,4,5,8,13)$

(m) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D)$

(n) $\overline{F}(A,B,C) = \prod(0,2,7)$

(o) $\overline{F}(A,B,C) = \prod(0,1,5)$

(p) $\overline{F}(W,X,Y,Z) = \prod(5,7)$

4) For the following function, write VHDL behavioral models that implement these functions using both a case statements and if statements (two separate models for each function).

(a) $F(A,B,C) = \sum(0,3,6)$

(b) $F(A,B,C) = m_1 + m_6$

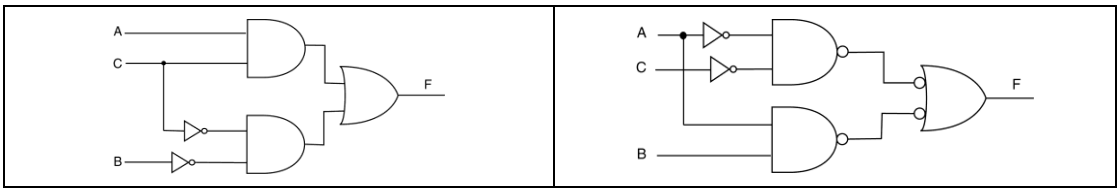
(c) $F(A,B,C,D) = \overline{A}C + \overline{B}C + BC$

(d) $F(W,X,Y) = \prod(1,2,6,7)$

(e) $F(A,B,C) = M_2 \cdot M_5$

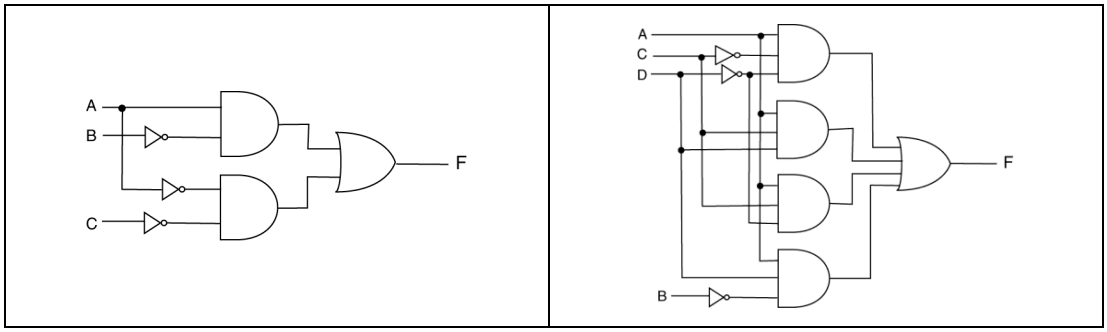
(f) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D) \cdot (C + D)$

5) Implement the following functions using concurrent signal assignment.



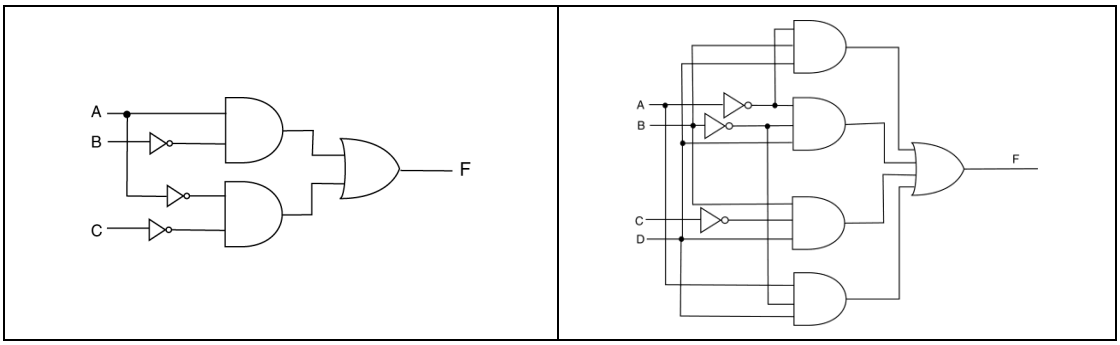
(a) (b)

6) For the following function, write VHDL behavioral models that implement these functions using both a case statements and if statements (two separate models for each function).



(a) (b)

7) Implement the following functions using concurrent, conditional, and selective signal assignment.



(a) (b)

8) Provide a VHDL model of an 8-input AND gate using concurrent, conditional, and selective signal assignment as well as a process statement.

9) Provide a VHDL model of an 8-input OR gate using concurrent, conditional, and selective signal assignment as well as a process statement.

10) Provide a VHDL model for a 5-input NAND gate using concurrent, conditional, and selective signal assignment as well as a process statement.

19 Chapter Nineteen

(Bryan Mealy 2012 ©)

19.1 Chapter Overview

As stated earlier, the act of digital design refers to establishing an input/output relationship on a digital circuit such that it solves some problem or does something useful. This definition is specific in the result but extremely non-specific in actual implementations. In most cases, this genericity allows for more freedom regarding the ways you can model a circuit. As you've been seeing, VHDL affords you the ability to model circuits in many different but functionally equivalent ways, particular circuits that can be characterized by simple Boolean functions. This chapter builds on this genericity by presenting a generic method to represent the basic input/output relationship of digital circuits.

The slight problem regarding the material presented in this chapter is the fact that we base it around the simple logic functions we've been dealing with up to this point. As you'll surely discover in the upcoming chapters, digital design does not have that much to do with the reduction and implementation of functions. Implementing functions is a low-level approach to digital design, but we're at the point now where we're ready to perform our designs at a higher level. As you may guess, digital design performed at a higher level of abstraction is always more efficient.

Main Chapter Topics

- **DECODERS:** The chapter introduces decoders, which is a standard digital circuit. The two types of decoders include standard and generic decoders. Standard decoders are useful in memory circuits while generic decoders are facilitate the implementation of digital circuits represented in tabular formats.

Why This Chapter is Important

This chapter is important because it describes both the generic and standard decoders. Generic decoders as extremely useful in digital design based on their ability to implement circuits modeled using a table format.

19.2 An Introduction to Decoders

The standard digital device that represents input/output relationships is what I refer to as the *generic decoder*, or just *decoder*. The term generic decoder is something that I made up in an attempt to classify the most common approach to digital design. The generic decoder is something you've already been working with although you may not realize it yet. In addition to the generic decoder, there is also a *standard decoder*. Please realize that "generic" and "standard" decoders are two terms that you won't find in any other digital design text. The standard decoder is a special type of a generic decoder and has a special relationship between the inputs and outputs. In other words, a standard decoder is a subset of a

generic decoder as shown in Figure 19.1. In general, the standard decoder has some specific uses while the generic decoder is non-specific.

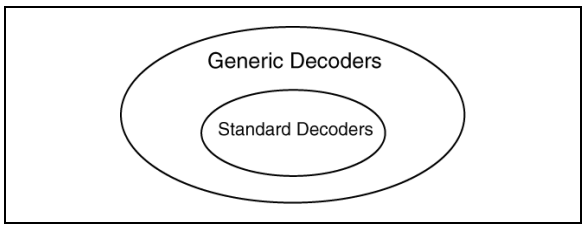


Figure 19.1: Venn diagram showing the hierarchy of decoders.

In general, VHDL uses the generic decoder to model look-up-tables, or LUTs. You're used to working with LUTs as the truth table was a very well-known table that you've toiled over. Truth tables nicely model Boolean functions; VHDL supports the implementations of truth tables without the pressing need for any type of reduction to take place. This means that if you have to implement a function in VHDL, you need to do your best to represent that function in a tabular format as these formats easily translate to VHDL models. So for now, we'll be mixing the notation between "function" and "decoder", since by our definition, they're the same thing.

Our new working definition of a generic decoder is this: *any non-sequential digital device that establishes a functional relationship between the device input(s) and output(s)*[236]. Generic decoders are use to model LUTs. This is so massively important that we need to coin yet another one of Mealy's new laws.

> **Mealy's Fifth Law of Digital Design:** Always first consider modeling a digital circuit using some type of a look-up table (LUT).

Keep in mind that the main reason this idea is so important is that if you can model something with a LUT, you can easily implement it in VHDL. You're probably thinking that items such as truth tables can quickly become large; but there are typically ways around this issue[237]: You've have not seen this yet, but many digital circuits can be described cleverly in a tabular format without resorting to the exhaustive representations that you've been used to up to this point in regards to truth tables. These clever digital circuits[238] also easily translate to VHDL models. We'll discuss this more later, but as with computer programming, you should always be on the lookout for opportunities to use decoders rather than trying to generate some fancy logic functionality.

19.3 Truth-table-based Generic Decoder Implementations

As mentioned earlier, if you really had to implement a generic decoder in VHDL that was defined by a truth table[239], you would not attempt to reduce it first. Instead, you whip out one of the following

[236] In case you did not notice, we did not define the term "non-sequential". This is a simple concept that we can skip over for now; we'll get to the full definition in a few chapters.
[237] Meaning that a circuit may have many inputs, but not all of the inputs are meaningful at the same time.
[238] Typically you'll find many digital ICs are described by using tables in the associated datasheet.
[239] This would also be true of compact minterm and/or maxterm forms. For that matter, it would also be true of standard SOP and POS forms, but you rarely see those used actual digital designs.

shortcut forms and implement it that way. I personally don't memorize the forms that follow; anytime I actually need to use one, I need to pull out my cheat notes to remind myself of the proper syntax for these various forms.

19.3.1 Selective Signal Assignment for Generic Decoders

This is the worn-out example we used for much of the verbage in some previous chapter. Here it is again for your pleasure.

Example 19-1

Write VHDL code to implement the decoder modeled by the accompanying truth table. Use selective signal assignment in your solution.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: The previous approach to this problem required the user to list the product terms associated with the function in longhand notation. Even with cutting and pasting on an editor, the previous solution was tedious. The new approach hopefully makes your generic decoder implementing life a bit more comfortable. As always, let's start with a black-box diagram; see Figure 19.2.

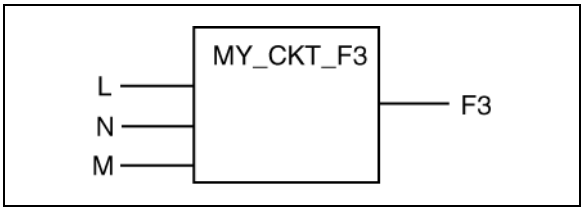


Figure 19.2: Black-box diagram for Example 19-1.

The first thing to note about the problem is that it is presented in truth table form. In order to make the solution more straightforward, you should first convert the problem to a more compact representation. In other words, this problem cries out for compact minterm form shown in Equation 19-1. You don't really need to take this step, but if you don't be careful as you may flip a bit somewhere.

$$F3(L,M,N) = \sum(1,6,7)$$

Equation 19-1

Figure 19.3 shows the final solution to Example 19-1; a few valuable comments are sure to follow the solution.

```
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
                F3 : out std_logic);
end my_ckt_f3;


architecture f3_8 of my_ckt_f3 is
   -- declaring the bundle
   signal s_sig : std_logic_vector(2 downto 0);
begin

   -- assigning the bundle using concatenation operator
   s_sig <= (L & M & N);

   with (s_sig) select
   F3 <= '1' when "001" | "110" | "111", -- listing the implicated minterms
         '0' when others;

end f3_8;
```

Figure 19.3: The no-nonsense solution to standard function implementation problems.

- Note the VHDL model includes comments (as do all good VHDL models).

- The VHDL model once again does not attempt to reduce the code. Any reduction of the code is thus a responsibility of the VHDL synthesizer.

- The entity declaration happens to describe the inputs as single signals. There certainly is a better approach to this problem. The solution is to create a bundle out of the individual input signals and use that bundle in the selective signal assignment statement. This indicates another common use for intermediate signals. The signal declaration creates the bundle. Assignment of the **L**, **M**, and **N** signals to the bundle is accomplished using the concatenation operator: "&".You should strive to use bundle notation in your VHDL operators whenever possible as it is rare that using individual signals are a better choice than bundle notation.

- There are two concurrent statements in the solution. It may initially feel like the concurrent signal assignment statement (the statement using the concatenation operators) is taking up space and doing something in hardware. The truth is that this is typical VHDL programming practice. The VHDL synthesizer does not interpret the concurrent signal assignment statement as requiring new hardware. In essence, this technique is a simple tool that you can use to "do what you need to do" in order to make your VHDL models more understandable.

- Only the TRUE cases (the cases where the function outputs are '1') are listed in this solution while the **when others** clause is used to handle the all of the other cases. This approach saves listing all of the cases where the function output is '0'. If there were fewer "0's" listed in the truth table, you would list the maxterm designators instead of the minterm designators and adjust the VHDL model accordingly.

- Because we are using bundle notation, the literals are assigned constant binary values using double quotes. Note that VHDL uses single quotes in the assigning of single bits.

Figure 19.4 shows one final solution to Example 19-1. This solution differs from the previous solution in that bundle notation was chosen to represent the three inputs signals instead of the previously used single signal entries. This approach allows the model to be implemented using less VHDL code. This may or not be considered a better approach. This example does highlight the fact that if you're the designer and you have the option of using bundle notation, you should always use bundle notation.

```
entity my_new_ckt_f3 is
    port ( LMN : in std_logic_vector(2 downto 0);
           F3 : out std_logic);
end my_ckt_f3;

architecture f3_1 of my_new_ckt_f3 is
begin

   with (LMN) select
   F3 <= '1' when "001" | "110" | "111", -- listing the minterms
         '0' when others;

end f3_1;
```

Figure 19.4: Use bundle notation in the entity declaration whenever possible.

19.3.2 Conditional Signal Assignment for Generic Decoders

Conditional signal assignment statements are also useful when using generic decoders to implement functions. The example shows this approach; though not optimal, we provide it here for completeness[240].

Example 19-2

Write VHDL code to implement the decoder modeled by the accompanying truth table. Use conditional signal assignment in your solution.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: Figure 19.5 provides the solution to Example 19-2. Note that this solution is somewhat similar to the solution of the previous example, but with different syntax.

[240] This is a wimpy justification for wasting space.

```
entity my_example is
    port ( L,M,N : in std_logic;
           F3 : out std_logic);
end my_example;


architecture my_soln_exam of my_example is
   -- intermediate signal declaration
   signal LMN: std_logic_vector(2 downto 0);
begin

   -- group signals
   LMN <= L & M & N;

   F3 <= '1' when LMN = ("001" or "110" or "111")
         else '0';

end my_soln_exam;
```

Figure 19.5: A conditional signal assignment implementation of a generic decoder.

19.3.3 Process Statement for Generic Decoders

Implementing generic decoders using process statements follow approaches similar to selective signal assignment and conditional signal assignment. Once again, there are two approaches to takes when using process statements: case statements and if statements[241]. Just for kicks, we'll redo the tired example using each of these statements. The one important thing to note is that the solution using the *case* statement strongly resembles the selective signal assignment statement while the solution using the *if* statement resembles the conditional signal assignment. This is not coincidental; there is a strong correlation between these types of statements.

Example 19-3

Write VHDL code to implement the decoder modeled by the accompanying truth table. Use a process statement in your solution. Provide a solution using both **if** and **case** statements.

L	M	N	F3
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution: Figure 19.6 provides the solution to Example 19-3 using an "**if"** statement while Figure 19.7 provides the other solution using a "case" statement. Yes, there are many ways to do the same thing in VHDL; it's up to you to decide the best approach.

[241] And also simple signal assignment, but we've done that already and won't mention it here.

```
entity my_example is
    port ( L,M,N : in std_logic;
           F3 : out std_logic);
end my_example;

architecture my_soln_exam of my_example is
   signal LMN: std_logic_vector(2 downto 0);
begin

   -- group signals to make thing easier
   LMN <= L & M & N;

   my_proc: process (LMN)
   begin
      if (LMN = "001" or LMN = "011" or LMN = "111")
         then F3 <= '1';
      else F3 <= '0';
         end if;
   end process my_proc;

end my_soln_exam;
```

Figure 19.6: A process-based implementation of a generic decoder using an "if" statement.

```
entity my_example is
    port ( L,M,N : in std_logic;
           F3 : out std_logic);
end my_example;

architecture my_soln_exam of my_example is
   -- intermediate signal declaration
   signal LMN: std_logic_vector(2 downto 0);
begin

   -- group signals for ease of working with case statement
   s_LMN <= L & M & N;

   my_proc: process (LMN)
   begin
      case s_LMN is
          when "001" | "110" | "111" =>  F3 <= '1';
          when others => F3 <= '0';
      end case;
   end process my_proc;

end my_soln_exam;
```

Figure 19.7: A process-based implementation of a generic decoder using a "case" statement.

19.4 Advanced Generic Decoders

The power of VHDL to model generic decoders is more apparent[242] when modeling functions with multiple inputs and outputs, which you'll do often in digital-land. Up until this point, you've generally have been using a truth table to represent this input/output relationship; from there you transferred this

[242] In case you may have not noticed this awesome power in the previous examples.

information into a K-map and then implemented the circuit. This approach worked well for single functions but is tedious when the number of input and/or the number of outputs becomes so large that you start to question whether you really want to be a digital designer or not. As you've already seen the in the previous examples, the generic decoder provides a method to bypass some of the more tedious parts of the design process such as the reduction of Boolean equations.

The generic decoder examples that follow highlight the fact that the generic decoder paradigm is something you should be familiar with from your experience with higher-level programming languages. Often times in algorithmic type programming, you'll have a need to perform some calculation over and over again. But instead of tying up processor resources by grinding out the calculation each time you need it, you opt to use a look-up table (LUT). With a LUT, you simply pre-calculate and pre-store all the required calculations somewhere in memory. In this way, you simply "look-up" the answer based on a given set of inputs instead of wasting computational effort to recalculate the value every time you need it. This approach is fast and simple and only has the drawback of requiring a finite amount of memory.

You can also perform look-up table-like operations in VHDL. You've actually been doing them for most of this chapter but this fact will become more obvious in the example that follows. For example, you'll sometimes need to add one value to another value. When you see the word "add" you may think you'll need to include a ripple carry adder into your circuit (extra hardware). But in some situations[243], you can simply use a generic decoder to "assign" the answer to the outputs instead of actually doing the calculation. This is the LUT approach and is a well known trick in digital design-land, be sure to keep this fact in mind when you're asked to design various circuits.

Example 19-4

Provide a VHDL model that implements the functionality described in the following truth table. Use a generic decoder in your VHDL model. Provide both a dataflow and a behavior description of the non-standard decoder.

A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

[243] If the cases of numbers being added are limited, for example.

Solution: The provided truth table represents a circuit with four inputs and three outputs. Your job is to design a circuit that would implement the three functional relationships shown in the truth table. Using a standard approach to digital design would require that you generate three K-maps and generate three Booleans equations from those K-maps. Modeling the circuit with VHDL is a much better option. As with most circuits, you can model this circuit in many different ways; this example shows two of the more intelligent approaches.

Figure 19.8 shows the black box diagram for this solution. Note that we have opted to represent both the inputs and outputs as bundles; this will make the implementation of the subsequent VHDL models more straightforward. Also, note that the bundle names conveniently relate to the columns in the original truth table. Figure 19.9 shows the associated entity declaration.

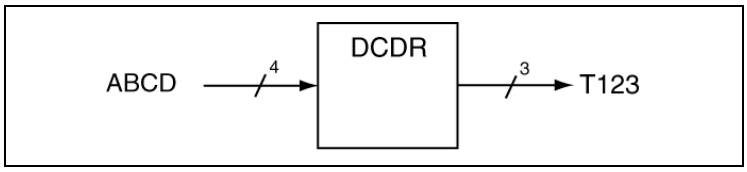


Figure 19.8: Dark box diagram for Example 19-4.

```
entity dcdr is
   port ( ABCD : in std_logic_vector(3 downto 0);
          T123 : out std_logic_vector(2 downto 0));
end dcdr;
```

Figure 19.9: The entity declarations associated with Example 19-4.

Figure 19.10 shows two different models that implement the input/output relationship of Example 19-4. The model in Figure 19.10(a) is a dataflow model while the model in Figure 19.10(b) is a behavioral model. Both of these models effectively restate the information presented in original truth table but in a different form. Both of these truth tables also contain another interesting feature. Note that for both of these cases, the entire truth table is not directly represented. In the model of Figure 19.10(a), the *when others* statement covers the last set of cases in the truth table; this is possible because the output associated with the final rows in the truth table are all "000". The same type of statement appears in Figure 19.10(b).

Both of these cases represent typical approaches to using VHDL to model relatively complex functional relationships. It would have been possible to represent all of the rows in the truth table, but that would only have served to increase the length of the VHDL model which is this case would be pointless. Either way, the size of the synthesized circuit would have most likely been the same in each case. The VHDL architectures shown in Figure 19.10(a) and Figure 19.10(b), and particularly the information presented in these architecture bodies, does in fact look somewhat like a table[244].

[244] Squinting your eyes may help you see this mo better.

```
-- dataflow modeling approach
architecture dec_dataflow of dcdr is
begin
   with ABCD select
   T123 <= "110" when "0000",
           "000" when "0001",
           "100" when "0010",
           "010" when "0011",
           "000" when "0100",
           "101" when "0101",
           "000" when "0110",
           "101" when "0111",
           "010" when "1000",
           "000" when others;
end dec_dataflow;
```

(a)

```
-- behavioral modeling approach
architecture dec_behavioral of dcdr is
begin
   dec: process(ABCD)
   begin
      case ABCD is
         when "0000" => T123 <= "110";
         when "0001" => T123 <= "000";
         when "0010" => T123 <= "100";
         when "0011" => T123 <= "010";
         when "0100" => T123 <= "000";
         when "0101" => T123 <= "101";
         when "0110" => T123 <= "000";
         when "0111" => T123 <= "101";
         when "1000" => T123 <= "010";
         when others => T123 <= "000";
      end case;
   end process;
end dec_bahavioral;
```

(b)

Figure 19.10: A dataflow (a) and behavioral (b) model of a decoder implementing the truth table shown in Example 19-4.

Example 19-5

Provide a VHDL model that implements the functionality described in the following truth table. Not listed in the following table is a CE input; when the CE input is a '0', the outputs of the circuit are all '0's; otherwise, the table below is used. Use a generic decoder in your VHDL model.

A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Solution: This problem has only seemingly slight modification from the previous problem. The CE input essentially "enables" this decoder to do something meaningful. This example is therefore the same as the previous example except the inclusion of the enable input. One other major difference between

this problem and the previous problem is that this problem did not state how to model the solution. Once again, you can solve this problem many ways, but by far Figure 19.12 shows the easiest approach. However, Figure 19.11 shows the required modifications to the dark box diagram to support this solution.

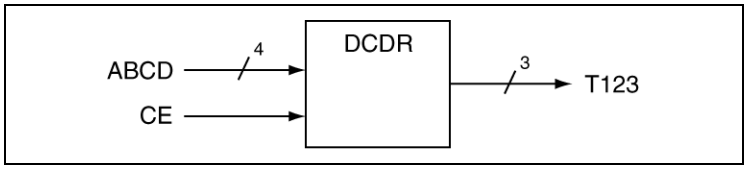


Figure 19.11: Dark box diagram for Example 19-4.

The solution shown in Figure 19.12 is classic VHDL. Note that the solution embeds a **case** statement into an **if** statement. This is definitely the best way to model an "enable" input in VHDL. I fully admit that I borrowed the body of the **case** statement (cut and pasted) from the solution of the previous example.

```
entity dcdr is
   port ( ABCD : in std_logic_vector(3 downto 0);
             CE : in std_logic;
           T123 : out std_logic_vector(2 downto 0));
end dcdr;

architecture dec_behavioral of dcdr is
begin
   dec: process(ABCD,CE)
   begin
      if CE = '1' then
         case ABCD is
            when "0000" => T123 <= "110";
            when "0001" => T123 <= "000";
            when "0010" => T123 <= "100";
            when "0011" => T123 <= "010";
            when "0100" => T123 <= "000";
            when "0101" => T123 <= "101";
            when "0110" => T123 <= "000";
            when "0111" => T123 <= "101";
            when "1000" => T123 <= "010";
            when others => T123 <= "000";
         end case;
      else
            T123 <= "000";
      end if;
   end process;
end dec_behavioral;
```

Figure 19.12: The entity declarations associated with Example 19-4.

19.5 Standard Decoders

The reality is that there are two types of decoders out there. As a result, when you hear the word "decoder", it does not actually refer to a specific type of circuit. Decoders come in one of two flavors: the standard decoder and the non-standard decoder discussed previously. While the non-standard decoder establishes a specific relationship between circuit inputs and outputs, the standard decoder has a more rigid definition. What I see is that the references made to decoders in digital-land fall into two

categories: what I refer to as standard decoders, and everything else. The approach described in this text highlights the two general uses of a decoder: a non-standard decoder is look-up table (LUT) while a standard decoder has a special relationship that we'll describe in the next few paragraphs.

As with non-standard decoders, the standard decoder also establishes a relationship between circuit inputs and outputs. The main difference is that the number and function of circuit inputs and outputs are fixed by the definition of the decoder. Figure 19.13(a) shows a gate-level diagram of a 2:4 standard decoder. Note that in Figure 19.13(a), due to the configurations of the inputs S1 and S0, only one of the AND gates will be active at a given time. In other words, at any given instance in time, only one of the outputs F0, F1, F2 or F3 will be a '1', while the three others will be a '0'[245].

The condition that makes this a standard decoder is the relationship between the number and attributes of the inputs and outputs. The bulleted item listed below highlight these main attributes:

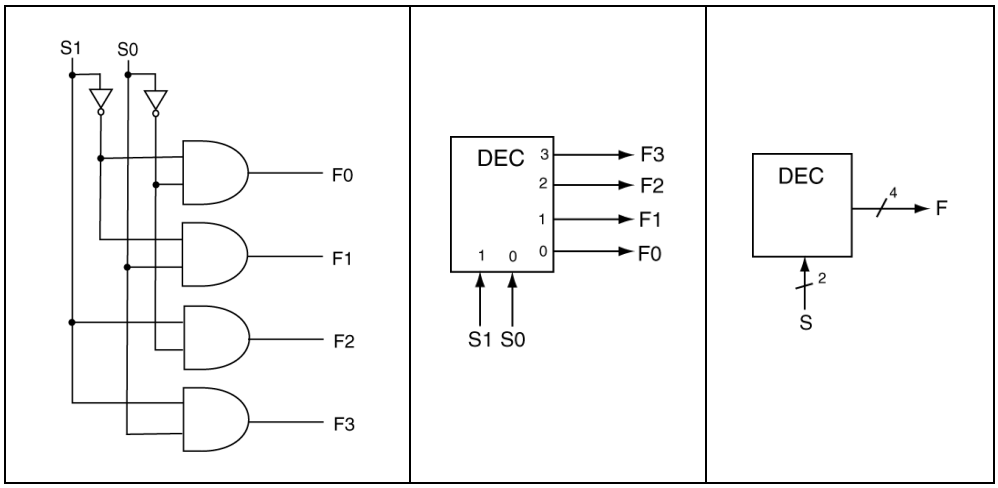
- Standard decoders always have a binary-type relationship between the inputs and outputs. For example, standard decoders always come in flavors such as 1:2, 2:4, 3:8, 4:16, etc. Note that this progression has a **n:2^n** relationship. In this notation, the first digit refers to the number of control variables that the circuit contains while the second variable refers to the number of outputs the circuit contains. Also, note that **n** input variables (binary) can reference **2^n** unique binary combinations.

- Although the schematic diagram of circuit of Figure 19.13(b) is an adequate approach to describing a standard decoder, the schematic diagram of Figure 19.13(c) is more common, particularly in the context of VHDL. Note that in Figure 19.13(b), the small numbers associated the circuit inputs and outputs indicate a weighting on those inputs and outputs. You must attach these numbers unless you are using the bundle notation shown in Figure 19.13(c). Without these numbers in the non-bundle case, there would be no way of knowing the ordering associated with either of the inputs. The bundled case implies this ordering.

- Only one of the outputs of the standard decoder is active at a given time. This is true because the control variables are arranged such that only one of the internal AND gates is non-dead at a given time. Along these lines, a decoder can also use NAND gates instead of AND gates for the internal circuitry. In this approach, all of the outputs except one are high at a given time while the other output is low. Figure 19.14 shows the circuit and the associated schematic diagram for a NAND gate-based standard decoder. The final result of the NAND-based decoder is the opposite of the AND-based decoder in that only one of the outputs is '0' at a given time while the other outputs are in a '1' state. The bubbles on the output of the Figure 19.14(b) can be thought of a being the same bubbles on the NAND gates[246].

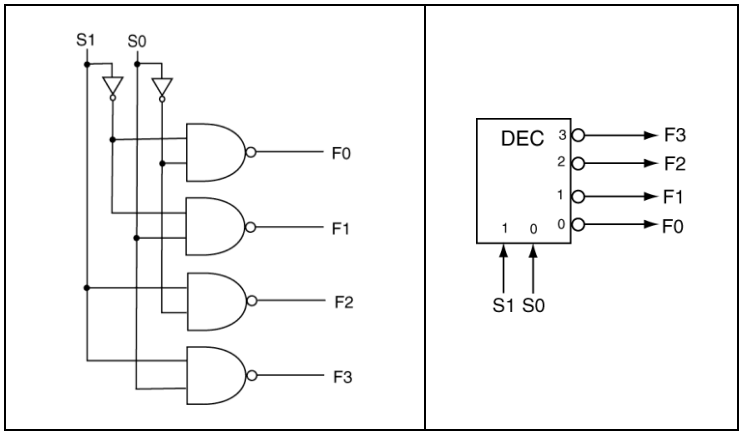
[245] A more complete explanation of this circuit appears in the MUX description in Chapter 7.

[246] This is an overly simplified description; there's actually a lot more to it. We'll be describing the entire story in a later chapter.



(a) (b) (c)

Figure 19.13: A standard 2:4 decoder in schematic and circuit forms.



(a) (b)

Figure 19.14: A standard 2:4 decoder with inverted outputs.

VHDL easily models standard decoders. Examining the VHDL model provides a viable approach to understanding the input/output relationship of the circuit if your understanding is still unclear from the associated circuit diagrams of Figure 19.13 and Figure 19.14. Figure 19.15 shows an entity declaration for a standard 2:4 decoder using bundle notation. The following figures show dataflow and behavioral models for the associated architecture.

```
entity dec4t2 is
   port ( SEL : in std_logic_vector(1 downto 0);
          F : out std_logic_vector(3 downto 0));
end dec4t2;
```

Figure 19.15: Entity declaration for standard 2:4 decoder.

Figure 19.16 shows two forms of dataflow models for the standard decoder. Figure 19.16(a) shows the decoder implemented using one selective signal assignment statement while Figure 19.16(b) implements the decoder using one conditional signal assignment statement. There are a few important items to notice in both of these cases.

- There is an implied weighting used by the bundle signals. As specified by the entity declaration, both the control signals and output signals were declared as bundles. Since the bundle notation used the "downto" notation (as opposed to the "to" notation), the left-most bits in the bundle are the most significant bits. The result of this notation is that a SEL of "00" is the lowest valued selection input value which corresponds to data outputs of "0001" which shows the least significant bit in a '1' state. This implied numbering is true for all bundle signals.

- Both dataflow models have similar structures in that they both list every possible case based on the two input signals and they both contain a "catch-all" clause. The catch-all clause is handled with the "when others" clause for selective signal assignment and with the final "else" in the conditional signal assignment. Some type of catch-all statement is massively important in VHDL models.

- One important feature to note from the VHDL models is that a default case is always provided. For both of the VHDL models of Figure 19.16, the *when others* statement is technically not required but it is considered good VHDL programming practice to include in the model. In other words, for both models the four lines before the when others statements provides every possible input combination for the SEL inputs. Because of this, there is technically no need to include the when others line. In this particular case, providing the when others line proves to be a great debugging tool in that if you detect the output of the decoder lines to be all zeros, you know there is a problem with your model.

```
architecture dec_a of dec4t2 is
begin
   with SEL select
   F <= "0001" when "00",
        "0010" when "01",
        "0100" when "10",
        "1000" when "11",
        "0000" when others;
end dec_a;
```

(a)

```
architecture dec_b of dec4t2 is
begin
   F <= "0001" when SEL = "00" else,
        "0010" when SEL = "01" else,
        "0100" when SEL = "10" else,
        "1000" when SEL = "11" else,
        "0000";
end dec_b;
```

(b)

Figure 19.16: Dataflow models for a standard 2:4 decoder.

Figure 19.17 shows two behavioral model implementations of a standard 2:4 decoder. These implementations are surprisingly similar to the dataflow implementations of Figure 19.16. The dataflow and behavioral implementations of these two devices clearly shows the analogy between the selected assignment and the **case** statement and a similar analogy between the conditional signal assignment and the **if** statement. Here are a few other important items:

- Because the process sensitivity list contains the SEL signal, the process statement is re-evaluated each time a change in the SEL signal is detected.

- Both behavioral models contain catch-all statements. For the **if** statement, the final **else** is the catch-all statement while the **when others** statement is the catch all statement for the **case** statement. The catch-all assignments of "0000" is arbitrary since this output is, by definition, not specified by the description of the standard decoder.

- Both models list every possible combination of the SEL inputs (with desired outputs) followed by a catch-all statement. Listing every possible case is not required but does represent good VHDL programming practice.

```
architecture dec_c of dec4t2 is
begin
   dec: process(SEL)
   begin
      if (SEL = "00")     then F <= "0001";
      elsif (SEL = "01") then F <= "0010";
      elsif (SEL = "10") then F <= "0100";
      elsif (SEL = "11") then F <= "1000";
      else F <= "0000";
   end process;
end dec_c;
```

(a)

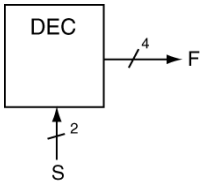
```
architecture dec_d of dec4t2 is
begin
   dec: process(SEL)
   begin
      case SEL is
         when "00" => F <= "0001";
         when "01" => F <= "0010";
         when "10" => F <= "0100";
         when "11" => F <= "1000";
         when others => F <= "0000";
      end case;
   end process;
end dec_d;
```

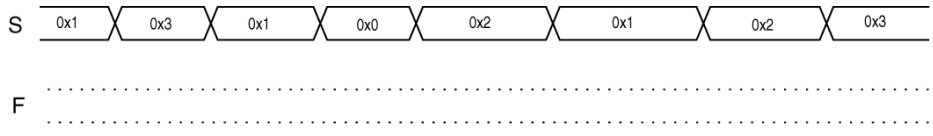
(b)

Figure 19.17: Behavioral models for the standard 2:4 decoder.

Example 19-6

Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.





Solution: Since the problem states that this is a decoder, the S input must be the selector inputs while the F are the outputs. The selection input bundle is two bits wide so it can select one of four different possible outputs. The outputs of a standard decoder have only one signal at a '1' value at any given time while the other bits are '0'. The problem description uses bundle notation in order to simplify the problem.

Note that there are only two selector bits, but the solution uses hex notation. The implication here is that the unused bits are all zero. For example, the hex value of "0x3" is actually "0011", but the general thought here is that the first two bits are not considered to affect the selection of output; arbitrarily, this problem uses only the two lower bits of the hex notation.

Figure 19.18 shows the final solution to Example 19-6. Note that the solution opts to also use hex notation. Note that the solution truly has that nice binary relationship between the selector inputs and the outputs. In addition, if you don't believe the hex notation, Figure 19.19 shows an alternate solution to this example. Once again, Figure 19.19 shows that only one of the outputs is at a '1' level at any given time.

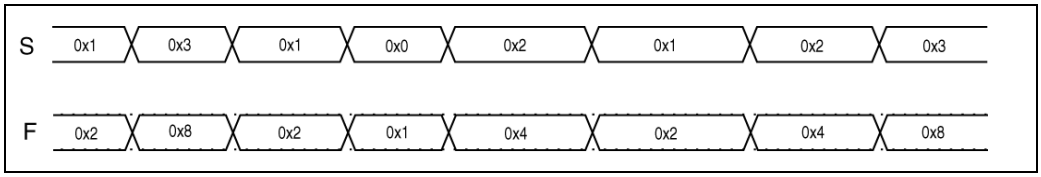


Figure 19.18: The solution to Example 19-6.

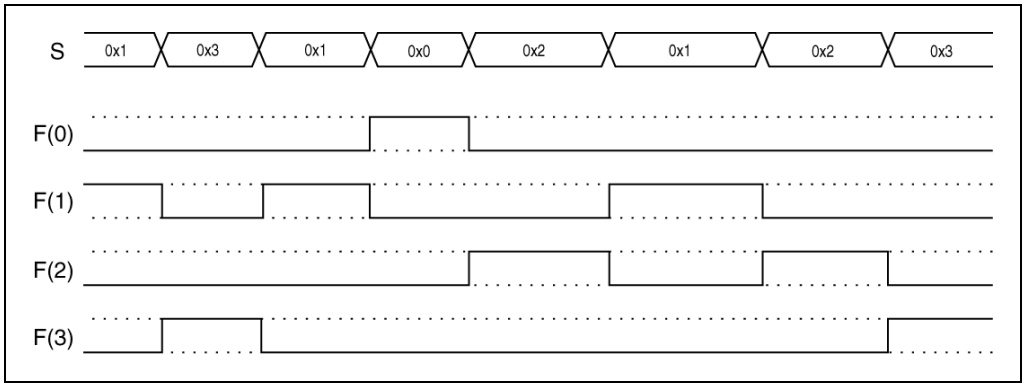
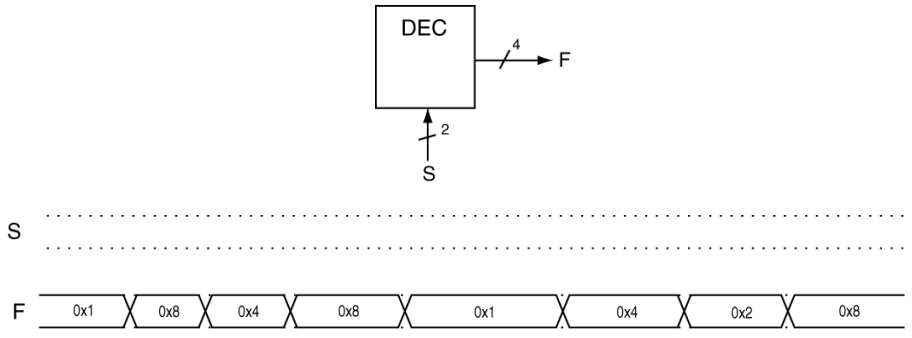


Figure 19.19: An alternate solution to Example 19-6; in this solution, the F bundle is expanded to show it constituent parts.

Example 19-7

Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



Solution: This problem is similar to Example 19-6, but we start this problem by knowing what the output values are. What we need to do is determine the values of the selector input S that generates the given output values. Since this is a standard 2:4 decoder, there can only be four output values. Note that this solution also uses bundle notation. Fun stuff.

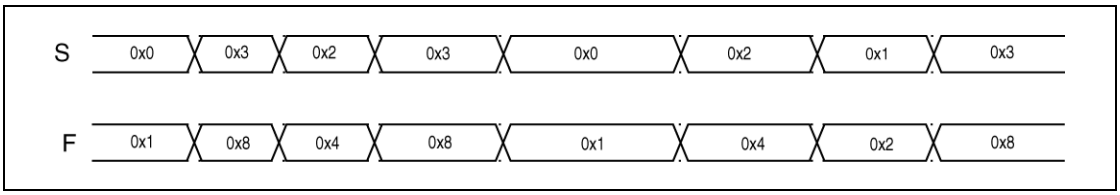


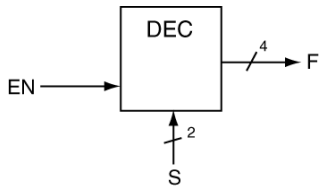
Figure 19.20: The solution to Example 19-7.

Example 19-8

Design a standard 2:4 decoder that has an EN input (enable). When the EN input is '1', the decoder outputs are all '0'. When the EN input is '0', the decoder outputs follow the accepted definition of a standard decoder.

Solution: This section described the attributes associated with a standard decoder in that the standard decoder has one set of inputs that directly control the outputs. In reality, decoders often have more control inputs. One of the typical controls on the decoder's inputs is an enable signal (or several enable signals). Once standard decoders add more input control signals, the underlying circuitry becomes more complicated and is rarely included in digital design textbooks. We'll not include it here either but we provide a table that describes the behavior of such a circuit along with a VHDL model. As an added bonus for your viewing enjoyment, we'll also provide a timing diagram with partial annotations. For the

record, a decoder with an enable input is sometimes referred to as a DMUX[247] but this is not a commonly used term[248]. A schematic symbol and a table describing the operation of a standard decoder with an enable input are shown in Figure 19.21(a) and Figure 19.21(b), respectively.



EN	S	F
0	- -	0 0 0 0
1	0 0	0 0 0 1
1	0 1	0 0 1 0
1	1 0	0 1 0 0
1	1 1	1 0 0 0

(a) (b)

Figure 19.21: A 2:4 decoder with an enable (a) and its behavior described in tablature format (b).

Figure 19.22 shows the associated VHDL model for this example. Note the ease at which the circuit is modeled by embedding a **case** statement inside of an **if** clause. The VHDL model is a prime example showing the versatility of behavioral modeling. A few sequential statements easily describe the required circuit operation. Although this device can be modeled using a dataflow model, it would have been tricky and the resulting code would have been less understandable.

```
-- 2:4 standard decoder with enable input (DMUX)
entity decoder is
   port (  EN : in std_logic;
           S : in std_logic_vector(1 downto 0);
           F : out std_logic_vector(3 downto 0));
end decoder;

architecture decoder of decoder is
begin
   dec: process(SEL)
   begin
      if (EN = '0') then
         F <= "0000";
      else
         case SEL is
            when "00" => F <= "0001";
            when "01" => F <= "0010";
            when "10" => F <= "0100";
            when "11" => F <= "1000";
            when others => F <= "0000";
         end case;
      end if;
   end process;
end decoder;
```

Figure 19.22: VHDL model of 2:4 decoder with enable input.

Figure 19.23 shows a timing diagram that describes the behavior of the circuit described in Example 19-8. There are two main features worth noting in this timing diagram.

[247] And, I've never understood exactly why; calling it a DMUX generates confusion. Plus, we have not discussed MUXes as of yet.
[248] The term does not have a commonly accepted definition either.

- The **F** bundle output is only all '0's when the enable input (EN) is '0'.
- Any time the EN input is '1', one and only one of the F bundle output signals are '1' while the remainder of the signals are '0'. This characteristic provides a quick but excellent method to verify proper operation of the decoder.

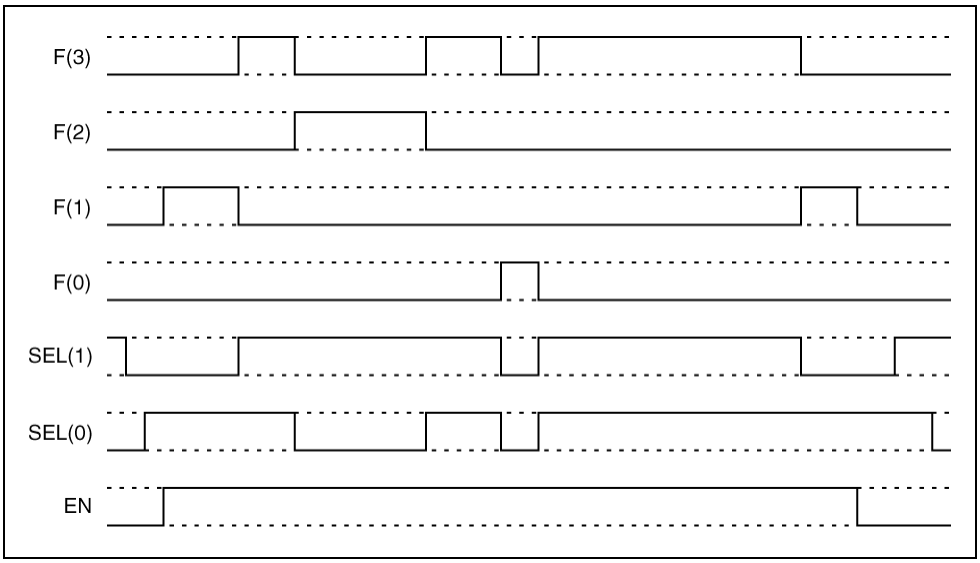


Figure 19.23: An example timing diagram for Example 19-8.

The following problem is another example of a standard-type decoder. The boundaries between circuit types can become a little fuzzy in digital design, so don't go overboard trying to place a proper label on your circuit. The more important thing is to do what you need to do to make the design work. Once your circuit works, you can name if George if you really want to.

Example 19-9

Write the VHDL code that implements the following circuit. The circuit contains an input bundle containing four signals and an output bundle containing three signals. The input bundle, **D_IN**, represents a 4-bit binary number. The output bus, **SZ_OUT**, indicates the magnitude of the 4-bit binary input number. The table below shows the desired relationship between the input and output. Use a selected signal assignment statement in the solution.

input range of D_IN	output value for SZ_OUT
0000 → 0011	100
0100 → 1001	010
1010 → 1111	001
unknown condition	000

Solution: This is another example of a standard decoder circuit. Not that there is too much more to say about standard decoders, but this example does show some useful VHDL syntax. Figure 19.24 shows the black box diagram for the solution while Figure 19.25 shows the entire VHDL model.

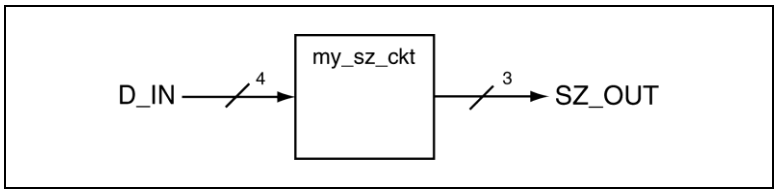


Figure 19.24: The black box diagram for Example 19-9.

The only comment for the VHDL model for Example 19-9 of is that it uses vertical bars as a selection character in the *choices* section of the selected signal assignment statement. This increases the readability of the code as it does with the similar constructs in algorithmic programming languages

Once again, the selected signal assignment statement is one form of a concurrent statement. We know this because of the fact that there is only one signal assignment statement in the body of the selected signal assignment statement. The selected signal assignment statement is evaluated each time there is a change in the *chooser_expression* listed in the first line of the selective signal assignment statement.

```
-------------------------------------------------------------------------
-- A standard decoder-type circuit for using selective signal assignment
-------------------------------------------------------------------------
entity my_sz_ckt is
    port ( D_IN   : in std_logic_vector(3 downto 0);
           SX_OUT : out std_logic_vector(2 downto 0));
end my_sz_ckt;

architecture spec_dec of my_sz_ckt is
begin
   with D_IN select
   SX_OUT <= "100"  when "0000" | "0001" | "0010" | "0011",
             "010"  when "0100" | "0101" | "0110" | "0111" | "1000" | "1001",
             "001"  when "1010" | "1011" | "1100" | "1101" | "1110" | "1111",
             "000"  when others;
end spec_dec;
```

Figure 19.25: The VHDL model solving Example 19-9.

Chapter Summary

- The more generic term "decoder" can replace a "function". The official definition of a decoder is: *a combinatorial (or non-sequential) digital device that establishes a functional relationship between the device input(s) and output(s)*. This definition defines a generic decoder, which is not to be confused with the standard decoder defined in a later chapter.

- VHDL uses generic decoders are to implement functional relationships. When using VHDL, there is no need to reduce the function before implementing it with a generic decoder. Generic decoders in VHDL have no limits to the number of inputs and outputs that the model can represent. Various types of VHDL concurrent signal assignment statements can be used to implement decoders; in all likelihood, these various approaches to implementation synthesize the same hardware.

- Standard decoders are a special type of decoder. The inputs and outputs of the a standard decoder exhibit a $\mathbf{n:2^n}$ relationship. In particular, if a standard decoder has **n** inputs, it will necessarily have $\mathbf{2^n}$ outputs. Standard decoders are used often in conjunction with hardware designed to access memory.

- **Mealy's Fifth Law of Digital Design:** Always first consider modeling a digital circuit using some type of a look-up table (LUT).

Chapter Exercises

1) For the following function descriptions, write VHDL models that implement these functions using concurrent signal assignment. Make no attempt to simplify these functions before you model them.

(q) $F(A,B,C) = \sum(2,3,5)$

(r) $F(A,B,C) = m_2 + m_5$

(s) $F(A,B,C,D) = \sum(4,3,15)$

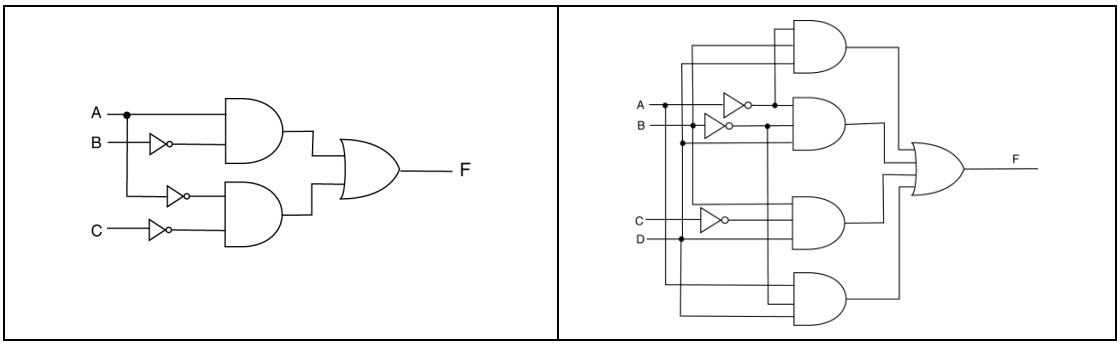
(t) $F(W,X,Y,Z) = \sum(12,14,15)$

(u) $F(A,B,C,D) = \sum(4,3,15)$

(v) $F(W,X,Y,Z) = \prod(1,4,5,8,13)$

(w) $F(A,B,C,D) = (\overline{A} + \overline{B}) \cdot (B + C + \overline{D}) \cdot (\overline{A} + D)$

2) Implement the following functions using some form of concurrent signal assignment; make no attempt to simplify the functions before you implement them.



(a) (b)

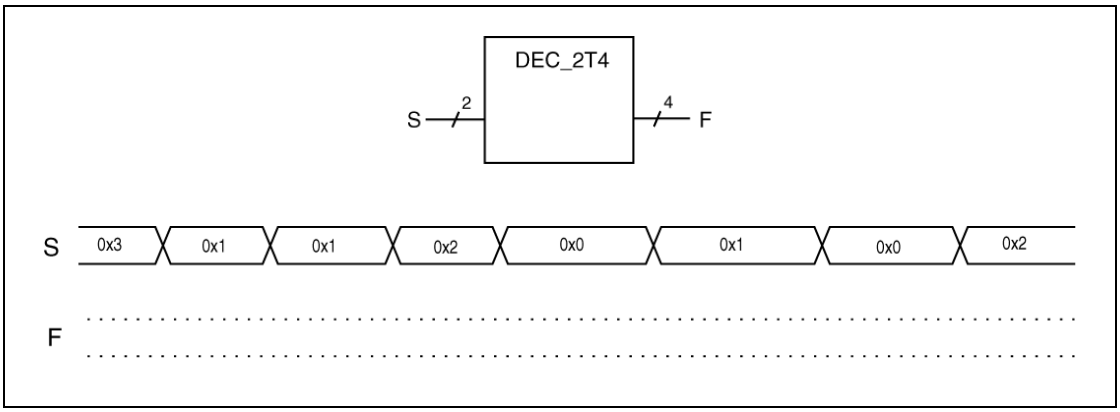
3) Write a VHDL model that implements the following truth table. Use bundles whenever humanly possible.

A	B	C	D	X1	X2	X3	X4	X5
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	1	1
0	0	1	0	0	0	0	0	1
0	0	1	1	0	1	0	1	0
0	1	0	0	0	0	0	0	0
0	1	0	1	1	0	1	1	1
0	1	1	0	0	0	0	0	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0	0
1	0	1	0	1	1	1	0	0
1	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	1	1
1	1	0	1	0	1	1	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	1

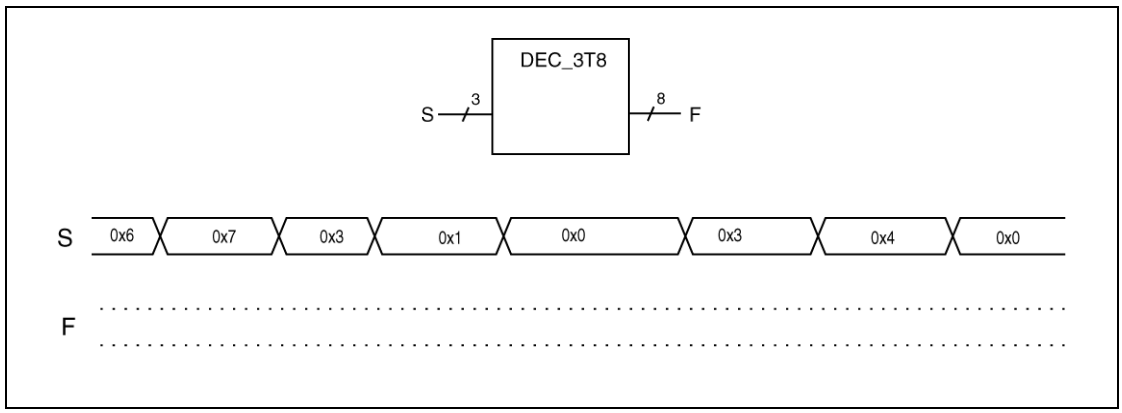
4) Write a VHDL model for a 3:8 decoder using a behavioral model.

5) Write a VHDL model for a 4:16 decoder using a dataflow model.

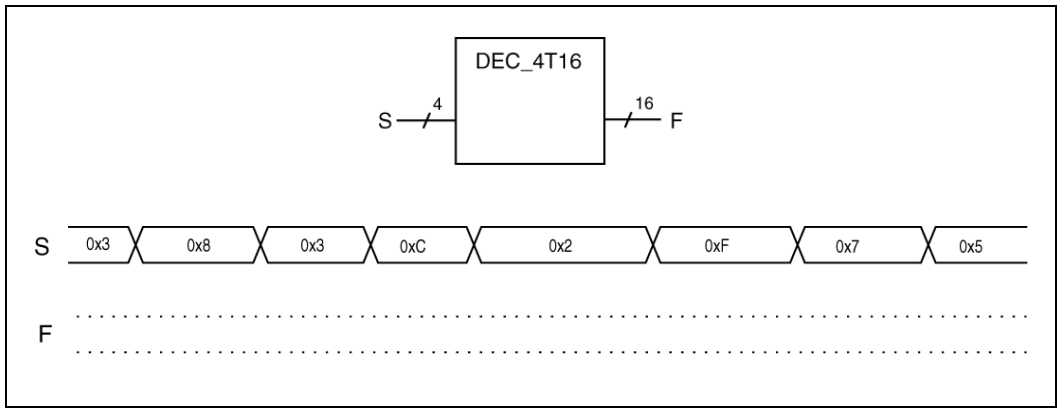
6) Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



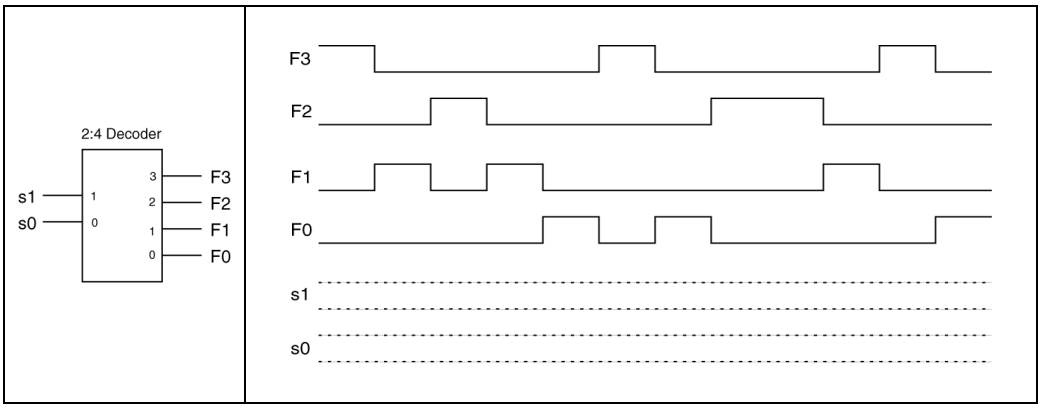
7) Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



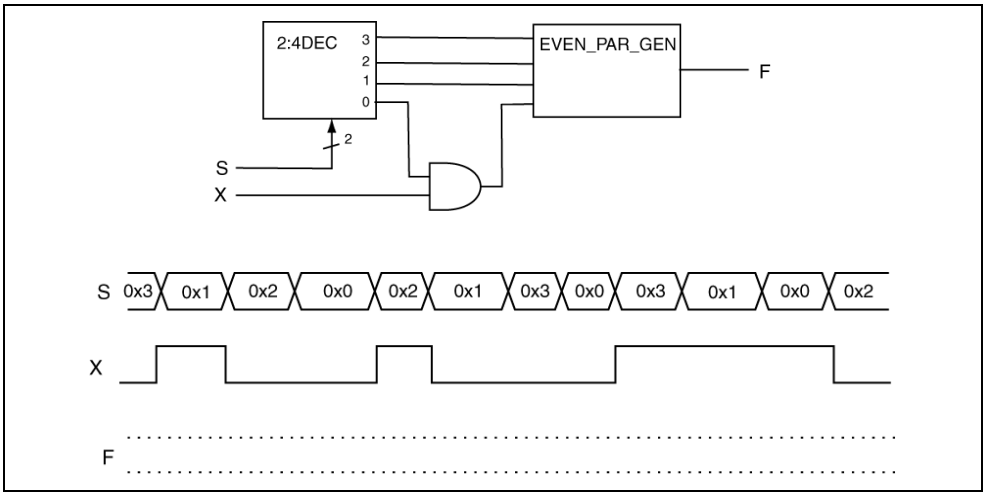
8) Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



7) Based on the standard 2:4 Decoder shown below, complete the following timing diagram by entering the values for signals s1 and s2 that would generate the listed output waveforms. Assume that propagation delays are negligible. Be sure to completely annotate this problem.

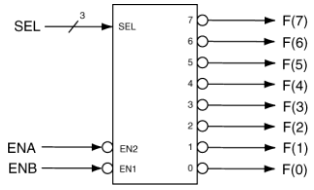


8) Use the following circuit to complete the listed timing diagram.



9) Indicate which of the three architectures best represent the standard 3:8 decoder shown below. Assume the ENA and ENB inputs are enable inputs which allow the F outputs to behave like a standard decoder.

```
entity my_dcdr is
   port (ENA,ENB : in std_logic;
               SEL : in std_logic_vector(2 downto 0);
                 F : out std_logic_vector(7 downto 0));
end my_dcdr;
```




```
architecture ckt1 of my_dcdr is
begin
   process(ENA,ENB,SEL)
   begin
       if (ENA = '0' and ENB = '0') then
            case SEL is
               when "000" =>  F <= "00000001";
               when "001" =>  F <= "00000010";
               when "010" =>  F <= "00000100";
               when "011" =>  F <= "00001000";
               when "100" =>  F <= "00010000";
               when "101" =>  F <= "00100000";
               when "110" =>  F <= "01000000";
               when "111" =>  F <= "10000000";
               when others => F <= "00000000";
            end case;
        end if;
   end process;
end ckt1;

architecture ckt2 of my_dcdr is
begin
       F <= "11111110" when (ENA = '1' and ENB = '1' and SEL = "000") else
            "11111101" when (ENA = '1' and ENB = '1' and SEL = "001") else
            "11111011" when (ENA = '1' and ENB = '1' and SEL = "010") else
            "11110111" when (ENA = '1' and ENB = '1' and SEL = "011") else
            "11101111" when (ENA = '1' and ENB = '1' and SEL = "100") else
            "11011111" when (ENA = '1' and ENB = '1' and SEL = "101") else
            "10111111" when (ENA = '1' and ENB = '1' and SEL = "110") else
            "01111111" when (ENA = '1' and ENB = '1' and SEL = "111") else
            "11111111";
end ckt2;

architecture ckt3 of my_dcdr is
begin
   process(ENA,ENB,SEL)
   begin
       if (ENA = '0' and ENB = '0') then
          if    (SEL = "000") then  F <= "11111110";
          elsif (SEL = "001") then  F <= "11111101";
          elsif (SEL = "010") then  F <= "11111011";
          elsif (SEL = "011") then  F <= "11110111";
          elsif (SEL = "100") then  F <= "11101111";
          elsif (SEL = "101") then  F <= "11011111";
          elsif (SEL = "110") then  F <= "10111111";
          elsif (SEL = "111") then  F <= "01111111";
          else F_OUT <= "11111111";
          end if;
       end if;
   end process;
end ckt3;
```

20 Chapter Twenty

(Bryan Mealy 2012 ©)

20.1 Chapter Overview

Our approach to digital design up until now has been somewhat limited[249]. Although we've done a few interesting designs, we're still not using what is probably an important piece of digital hardware: the multiplexor. Having a multiplexor in your digital bag of tricks is going to open the doors to more interesting digital designs. The basic multiplexor is not complicated; though actual implementations can become involved on a low level, higher-level abstractions are not a big deal. The good news is that any flavor of multiplexors are effortlessly modeled in VHDL. I can't think of anything else to say.

Main Chapter Topics

- **STANDARD DIGITAL MODULE: THE MULTIPLEXOR:** This chapter introduces the notion of a multiplexor from both a low-level and user-level standpoint. The low-level multiplexor hardware is instructive but is multiplexors quickly become complex as they are given more features.

Why This Chapter is Important

This chapter is important because it describes the Multiplexor, a standard digital model that serves as the basic "selection" mechanism in digital design.

20.2 Making Decisions in Hardware and Software Land

More likely than not, you probably have some experience programming computers[250]. Since computer programs generally "react" to various things, there needs to be a programming construct that handles decisions. The general notion in programming is that there is one processor and this processor does one thing at a time[251]. The notion of decisions as they relate to programming is based on the notion that programs make decisions based on the current conditions in a program: the program with either execute one set of instructions or another set of instructions. In other words, the program will choose to take either one instructional path or another path based on some condition the program views as important. Roughly speaking, a "conditional" statement is the mechanism used by the program to choose one path of execution over another.

[249] If you're still bored with digital design, then blame it on these limitations.

[250] If you don't, then this section is not going to make much sense, so you can skip it.

[251] Generally speaking, a processor executes one instruction at a time. Overall, a processor executes one instruction and then moves on to the next instruction.

Hardware design is similar. In general, your hardware will need to react to certain conditions in the circuit and choose one "result" over another "result" based on those conditions. Don't worry about the details right now, but it is a multiplexor that allows the hardware to choose something over another something in digital design.

I added this section because many students seemed to have issues with the notion of "choosing" something in software vs. "choosing" something in hardware. Here's the issue: the notion in software design is that one path of execution is chosen over another path. The main idea here is that it would be massively inefficient to somehow "execute" both paths and then choose the result you're interested in.

The problem arises in hardware when you choose between two "things". The general notion in hardware is that all the things you're choosing between are calculated in parallel (or concurrently). The multiplexor simply chooses the result based on the state of the hardware. In other words, if you need to choose between two different results, always plan to create the hardware to generate both results and then choose the result required by your circuit based on the state of some signal in the circuit[252].

The trouble that many people get into is that they try to design their hardware to run like a software-based choosing operation. While there are some good things to say about this approach, they are advanced issues and we'll simply pretend that they don't exist at this point. So for now, when you're designing "choosing" operations in hardware, know that you're going to always need to generate all the desired results and then simply choose the desired result once everything is ready. I know this may not make sense as of yet, but take a mental note to come back and read this again once you know more about multiplexors.

20.3 Multiplexors

The *multiplexor* is yet another standard and highly used digital circuit. When you hear the word multiplexor, or *MUX* as it is more commonly referred to[253], you need to think *selector*. A MUX is a generally a circuit with many inputs and one output; the output of the device generally represents a direct transfer of one of the inputs. That's about all there is to a MUX: it's a device that outputs one of multiple inputs based on a set of selection inputs (or control inputs).

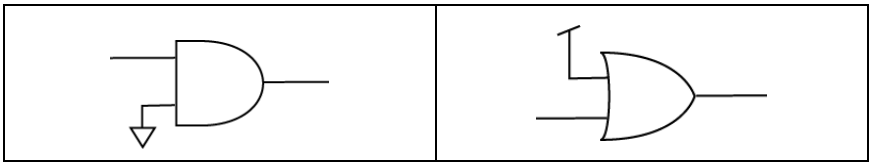
As boring as it may seem, we need to examine the internals of a simple MUX in order to give you a solid understanding of how they work. Other standard digital components (namely a standard decoder) share the gate-level circuitry in a MUX so it is worth looking at here. The MUX also has historical significance at the gate-level so you're generally expected to know how they work at both high and low levels.

The first thing we need to look at is a specific function of the basic AND and OR logic gates. You can effectively kill the output of a AND and OR gates by tying their outputs to '0' and '1', respectively. In other words, if you connect one of the inputs to a AND gates to a '0', the output of the gate, regardless of the state of any of the other inputs, is always be '0'. Similarly, if you connect one input of an OR gate to '1', the output of the OR gate is then always be '1'.

Figure 20.1 shows a gate-level depiction of the gate-killing functionality. The circuit in Figure 20.1(a) uses an inverted arrowhead to indicate a connection with ground ('0'). Figure 20.1(b) shows the slanted T symbol to indicate a connection to the circuit's high voltage ('1').

[252] As you start working with digital circuits, you'll see that this is not always optimal. In reality, any time a transistor in a digital circuit changes state, there is going to be a loss of power. If you can avoid calculating a result that you may not need, you avoid it. But for now, don't worry about efficiency; that will come naturally as you become more used to dealing with digital circuits.

[253] And for the record, the correct pronunciation is "mucks" and not "mooks".



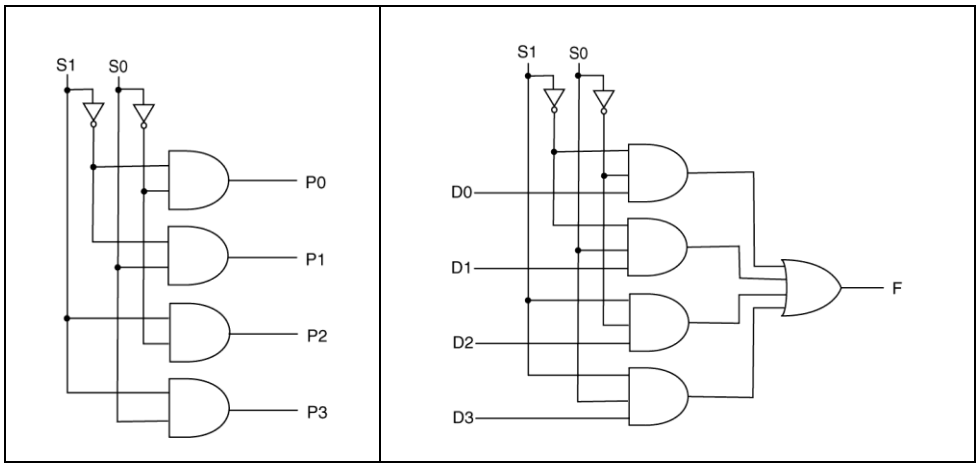
(a) (b)

Figure 20.1: Killing the AND (a) and OR (b) gates.

The next step in developing the MUX is assembling the selection circuitry shown in Figure 20.2(a). In this circuit, there are two variables S1 and S0 that are referred to as *selection variables*. In this circuit, only one of the P outputs will be a '1' at any given time while all other P outputs will be '0'. Note that each of the four AND gates are connected such that they will each have different input values based on the state of the selection variables. In other words, the inputs to the AND gates will all be different based on the method used to connect the selection variables. Convince yourself that one and only one AND gate will be a '1' at any given time before reading on. The effect this creates is that at any given time, the output of three of the AND gates will '0' while the other AND gate will have an output of '1'. In relation to Figure 20.1(a), three of the AND gates will be dead.

Figure 20.2(b) shows the final portion of the MUX circuitry. Knowing that three of the AND gates are officially dead (they have an output of '0'), the only hope that the circuit output F will be a '1' is if the D input on the un-dead AND gate is a '1'. In other words, if the D input on the non-dead gate is a '0', all of the AND gates will be dead and the F output will be a '0'. If however, the D input on the non-dead gate is a '1', then the non-dead AND gate output will be a '1', the OR gate will have an input of '1' and the OR gate output F will be a '1'.

What this circuit connection effectively does is transfer the value of one D input to the output F. To put this in MUX language, the MUX *selects* one of the D inputs to appear on the F output. The D input that appears on the F output is dependent upon which AND gate is un-dead which is inherently dependent on the values of the S1 and S0 variables (the selection variables). In terms of the MUX, the S1 and S0 inputs are the data selection inputs, which effectively select one of the D inputs to appear on the outputs. In as simple terms as possible, the selector inputs choose which data input will appear on the output. Once again, in official MUX language, the S1 and S0 inputs are referred to as the data selection inputs while the D inputs are the data inputs.



(a) (b)

Figure 20.2: The MUX input circuitry (a) and the complete MUX (b).

The MUX shown in Figure 20.2(b) is referred to as a 4:1 MUX because the device chooses between one of four inputs to appear on the output. MUXes generally have that binary relationship between the number of selection variables and the number of data inputs. Common flavors of MUXes include 2:1, 4:1, 8:1, 16:1 etc[254]. As you'll see later on, this is the most basic form of a MUX. In reality, MUXes come in many different flavors and quickly become complicated enough such that you'll want to avoid modeling them with gate-level logic. The truth is that you rarely need to model MUXes in anything other than VHDL. There are discrete ICs that contain MUXes, but they are somewhat limited; you'll find yourself using bunches of these to actually do what you need to do.

Example 20-1

Provide the following VHDL models for a 4:1 MUX: conditional signal assignment, selective signal assignment, ***if*** statement, and a ***case*** statement. Consider the data inputs and selection inputs to be bundles.

Solution: As the problem implies, there are many ways to model a MUX. There arguably no best approach to modeling a MUX using VHDL; but the worst way may be to use concurrent signal assignment. A MUX is a complicated enough circuit such that we want to avoid low-level data flow models. Being good digital designers, let's start this problem off with a black box diagram shown in Figure 20.3.

[254] But you're free to model anything you need; there are only a few items you need to pay attention to.

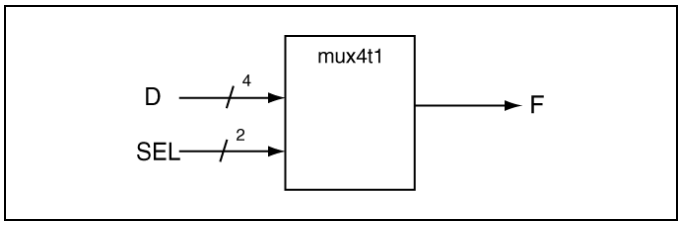


Figure 20.3: Black box diagram of bundle-based 4:1 MUX.

Figure 20.4 shows the entity declaration for the 4:1 MUX using bundles for the data and selection inputs. Each of the requested models uses this entity declaration.

```
entity mux4t1 is
   port (   D : in std_logic_vector(3 downto 0);
          SEL : in std_logic_vector(1 downto 0);
            F : out std_logic);
end mux4t1;
```

Figure 20.4: The entity declaration associated with the VHDL models of Example 20-1**.**

Figure 20.5 shows the conditional signal assignment model of a 4:1 MUX. There are a couple of items worth noting in this solution.

- The solution looks somewhat efficient compared to the amount of logic that would have been required if concurrent signal assignment statements were used. The VHDL code appears nice and is pleasing to the eye, which is a quality desirable for readability.

- The "=" is an "equivalence operator" which is a relational operator is used in conjunction with a bundle signal. In this case, the values on the bundle SEL lines are accessed using double quotes around the specified values. In other words, VHDL uses single quotes to describe values of single signals and double quotes to describe values associated with multiple signals, or bundles. Generally speaking, if you can use a bundle as opposed to individual signals, you should. Just for the heck of it, Figure 20.6 shows an alternative solution not using bundles for the SEL lines. As you can see, it does not represent an improvement over the model in Figure 20.5.

- VHDL uses the bundle access operator "( )" (the parenthesis) to specify signals within a bundle. In this example, the MUX needs to apply a single signal to the output F. The data inputs to the MUX use bundle notation, which subsequently requires the use to the bundle access operator in the model.

- For the sake of completeness, we've included every possible condition for the SEL signal plus a catch-all **else** statement. We could have changed the line containing '0' to D0 and removed the line associated with the SEL condition of "00". This would be functionally equivalent to the solution shown in but not nearly as impressive looking. You should clearly provide all the options in the code and not rely on a catch all statement for intended signal assignment. This is particular true when you need to debug your VHDL models.

Remember, a conditional signal assignment is a type of concurrent statement. In this case, the conditional signal assignment statement is "executed" any time a change occurs on the conditional signals (the signals listed in the expressions on the right side of the signal assignment operator). This is

similar to the concurrent signal assignment statement where the statement is "executed" any time there is a change in any of the signals listed on the right side of the signal assignment operator.

```
architecture mux4t1a of mux4t1 is
begin
   MX_OUT <= D(3) when (SEL = "11") else
             D(2) when (SEL = "10") else
             D(1) when (SEL = "01") else
             D(0) when (SEL = "00") else
             '0';
end mux4t1a;
```

Figure 20.5: Conditional signal assignment model of 4:1 MUX.

```
architecture mux4t1b of mux4t1 is
begin
   MX_OUT <= D(3) when (SEL(1) = '1' and SEL(0) = '1') else
             D(2) when (SEL(1) = '1' and SEL(0) = '0') else
             D(1) when (SEL(1) = '0' and SEL(0) = '1') else
             D(0) when (SEL(1) = '0' and SEL(1) = '0') else
             '0';
end mux4t1b;
```

Figure 20.6: An alternative conditional signal assignment solution.

Figure 20.7 shows the selected signal assignment model of a 4:1 MUX. Once again, there are a few items of interest regarding this solution.

- The VHDL code has several similarities to the conditional signal assignment solution of Figure 20.5. The general appearance is the same; both solutions are organized and clear.

- A ***when others*** clause is used as the catch-all statement. For this solution, the output is assigned a '0' in case each of the other options fails. The truth is that each possible value of the SEL was previously listed so the when other clause should never be evaluated. Adding the ***when others*** clause is considered good VHDL modeling practice and has ramifications in the simulation of VHDL models.

```
architecture mux4t1c of mux4t1 is
begin
   with SEL select
   MX_OUT <= D(3) when "11",
             D(2) when "10",
             D(1) when "01",
             D(0) when "00",
             '0'  when others;
end mux4t1c;
```

Figure 20.7: Selected signal assignment architecture for 4:1 MUX.

Figure 20.8 shows two flavors of behavioral models for the 4:1 MUX: Figure 20.8(a) shows an **if** statement model while Figure 20.8(b) shows a **case** statement model. It's more than worthwhile to list some useful observations regarding these two models.

- The two models of Figure 20.8 are similar. This being the case, there is no real advantage to using one approach over the other, although the case statement is generally accepted as being a

more straightforward solution. In general, if you have more than three "if" conditions, you should use a case statement instead.

- The two models of Figure 20.8 are also similar to the dataflow model implementations. Specifically, use of an ***if*** statement is similar to conditional signal assignment; use of a ***case*** statement is analogous to selected signal assignment. Convince yourself of this.

- The process sensitivity lists include both the D and SEL inputs. This means that anytime state of D or SEL change, the process statement is re-evaluated. This makes sense in that the output of the MUX could change anytime a change in the data or select inputs occurred.

```
architecture mux4t1d of mux4t1 is
begin
   mux: process(D,SEL)
   begin
      if (SEL = "00")    then F <= D(0);
      elsif (SEL = "01") then F <= D(1);
      elsif (SEL = "10") then F <= D(2);
      elsif (SEL = "11") then F <= D(3);
      else F <= '0';
      end if;
   end process;
end mux4t1d;
```

(a)

```
architecture mux4t1e of mux4t1 is
begin
   mux: process(D,SEL)
   begin
      case SEL is
         when "00" => F <= D(0);
         when "01" => F <= D(1);
         when "10" => F <= D(2);
         when "11" => F <= D(3);
         when others => F <= '0';
      end case;
   end process;
end mux4t1e;
```

(b)

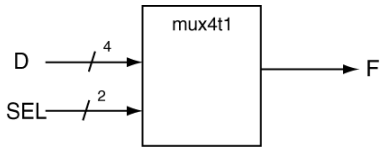
Figure 20.8: 4:1 MUX modeled with if statements (a), and case statement (b).

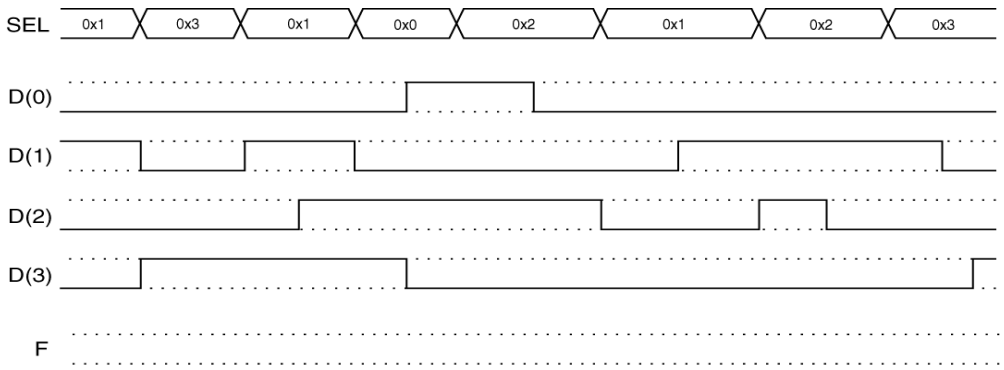
If you haven't noticed by now, modeling digital circuits using VHDL is massively powerful. In other words, using the various VHDL constructs, you can model any conceivable digital circuit. This is an important concept, but its important becomes more obvious in the context of MUXes. The issue in digital design is that you constantly need to choose between "things" (things in this case mean signals or bundles), but, there is not necessarily going to be one existing MUX model that covers every possible case in digital design. Therefore, you really need to understand these modules. Moreover, the better you understand the basic VHDL concurrent statements, the better[255] your VHDL models will appear.

[255] By better, we mean more clear, more readable, more understandable, and generally more good.

Example 20-2

Use the following block diagram to complete the provided timing diagram. For this problem, consider the block diagram to represent a 4:1 MUX containing no surprises.





Solution: Since this is a 4:1 MUX, the output will match one of the four data inputs depending upon which input is selected by the selector inputs. The SEL input is provided in bundle notation while the D input bundled has been expanded for reasons you'll see momentarily. Based on how a MUX operates, and based upon the VHDL models provided in the solution to Example 20-1.

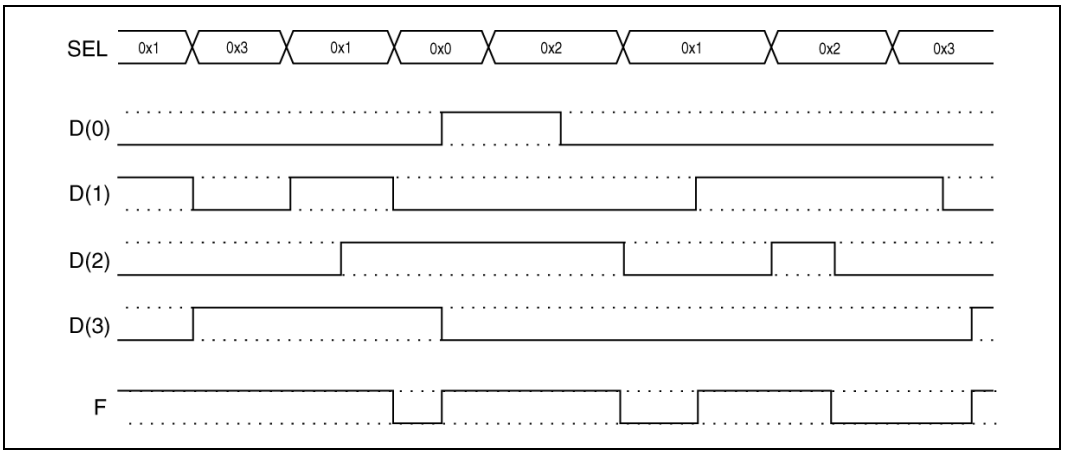
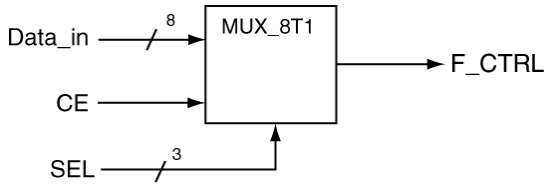


Figure 20.9: The solution to Example 20-2.

Example 20-3

Provide a VHDL model that describes the 8:1 MUX as shown below. Model this MUX using ***if*** statements. In the black box diagram shown below, the CE input is a chip enable. When CE = '1', the output behaves as a standard MUX. When CE is '0', the output of the MUX is '0'; otherwise, the device acts like a normal 8:1 MUX.



Solution: The problem shows an 8:1 MUX with a chip enable (or chip select) input. In general, the chip enable input turns the device "on" or "off"[256]. There are many approaches to modeling this circuit; Figure 20.10 show one possible solution. There are a few things to note about this solution.

- The problem stated what the outputs would be when the device was not enabled; this action is modeled by the "if" statement in the solution in Figure 20.10.
- The solution in Figure 20.10 uses "if" statements as specified by the problem. It probably would have been easier (and/or clearer) to use a case statement.

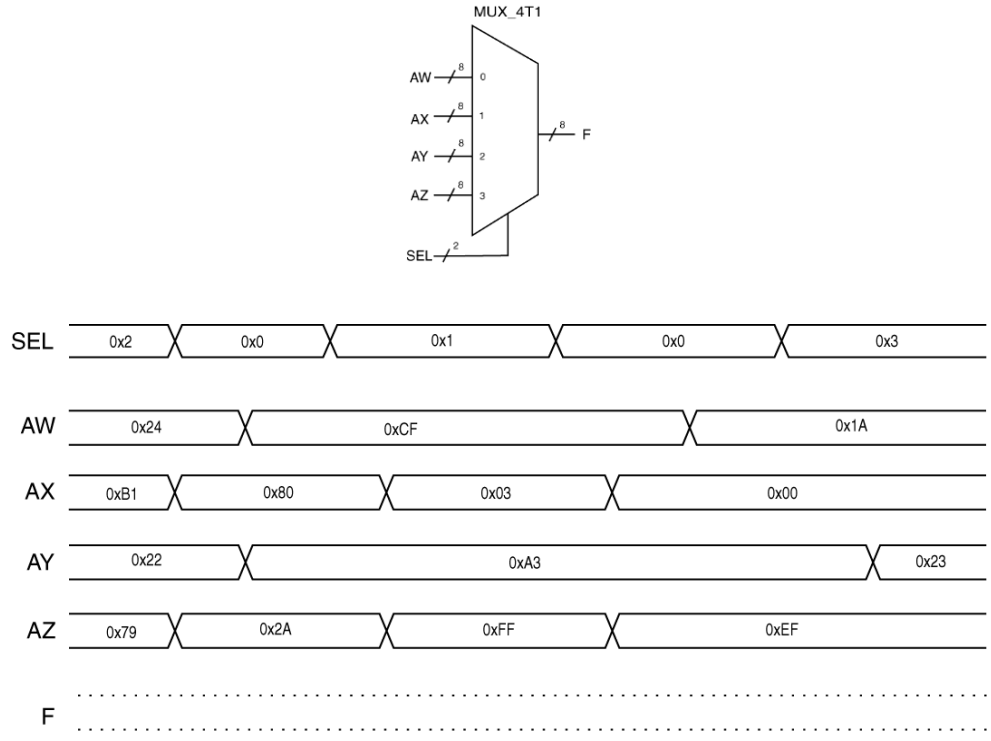
```
entity mux_8t1_ce is
    port ( Data_in : in std_logic_vector (7 downto 0);
                SEL : in std_logic_vector (2 downto 0);
                 CE : in std_logic;
            F_CTRL : out std_logic);
end mux_8t1_ce;

architecture my_8t1_mux of mux_8t1_ce is
begin

   my_mux: process (Data_in,SEL,CE)
   begin
      if (CE = '0') then
          F_CTRL <= '0';
      else
         if    (SEL = "111") then F_CTRL <= Data_in(7);
         elsif (SEL = "110") then F_CTRL <= Data_in(6);
         elsif (SEL = "101") then F_CTRL <= Data_in(5);
         elsif (SEL = "100") then F_CTRL <= Data_in(4);
         elsif (SEL = "011") then F_CTRL <= Data_in(3);
         elsif (SEL = "010") then F_CTRL <= Data_in(2);
         elsif (SEL = "001") then F_CTRL <= Data_in(1);
         elsif (SEL = "000") then F_CTRL <= Data_in(0);
         else  F_CTRL <= '0';
         end if;
      end if;
   end process my_mux;

end my_8t1_mux;
```

Figure 20.10: The VHDL model for Example 20-3.

[256] Keep in mind that it if it is not explicitly stated what is meant by "on" and "off", you the digital design will need to decide for yourself and implement it in the associated VHDL model.

Example 20-4

Using the following diagram of a 4:1 MUX, complete the provided timing diagram. Also provide a VHDL model that implements the 4:1 MUX.



Solution: The first thing to notice about this example is the fact that it uses a new and distinctive shape for the MUX. Circuit diagrams usually use this shape to represent MUXes and are therefore important for all digital designers you to use this shape also. The issue with the shape is that when you see the shape in a black box diagram, you'll know immediately what the device is doing: namely, choosing which of the inputs is going to appear on the output. In other words, the distinctive shape immediately alerts the reader of the diagram that this particular device is "selecting" an input to appear on the output.

Also of killer importance for this MUX diagram is the notion that the data inputs must also contain indexing numbers. Notice that the numbers associated with the data inputs range from [0,3], which by design corresponds to the numbers that can be represented by the control inputs (represented by a 2-bit wide bundle). If the data inputs are not numbered, you'll not know exactly how the selection inputs control which data input will appear on the output. In summary, when using MUXes, you should always use this distinctive shape and always label the data inputs with selection indexes.

In addition, to note in this example is that the fact that both the data and selection inputs use bundle notation. This heavily implies that the output "F" is also going to use bundle notation.

Beyond all the supporting details listed above, the problem is rather straightforward. Figure 20.11 shows the solution to this example. One important issue regarding this solution is the use of the vertical dotted lines; including these lines help you generate the correct answer. This is actually not an overly complicated timing diagram, but they can often become quite complicated. In order to handle these

types of complications, you must do what you can to make sure you don't mess up the solution. The first step in arriving at this solution would be to draw the vertical lines that correspond to the change in selector inputs. More likely than not, these vertical lines indicate on the output variable when the output is changing. As you can see from the problem, the output also changes at other times based on some of the data selection inputs; this is something you need to be aware of also.

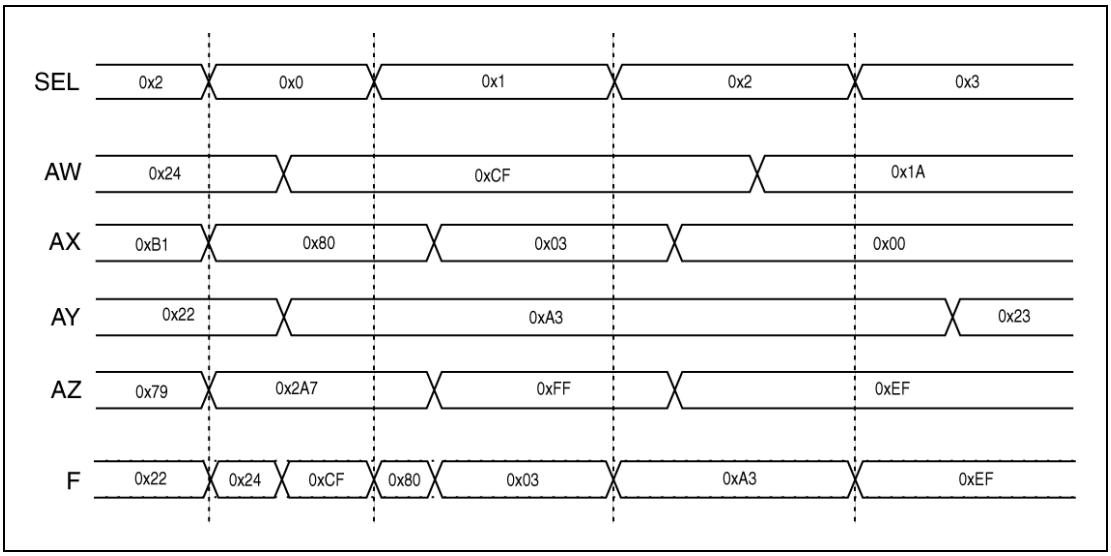


Figure 20.11: The solution to Example 20-4.

Finally, Figure 20.12 shows a VHDL model for this example. When I created this VHDL model, I did not start writing from scratch; I instead went back to the 4:1 MUX implemented with selective signal assignment and used that model as a starting point. It is always easier and more efficient to start from something that you already have rather than starting from square zero. The only new and useful thing to notice about the solution in Figure 20.12 is that the "catch-all" statement uses an interesting VHDL terminology. Notice that the model uses "(**others => '0'**); this means that the output, no matter how wide it is, is assigned the value of zero. This terminology is quite useful in terms of making your code generic because now you don't need to know the width of the output you're assigning to. In truth, there are many ways to make your VHDL models generic, but this text does not cover many of them.

```
entity mux_4t1 is
    Port (            SEL : in  STD_LOGIC_VECTOR (1 downto 0);
          AW, AX, AY, AZ : in  STD_LOGIC_VECTOR (7 downto 0);
                       F : out STD_LOGIC_VECTOR (7 downto 0));
end mux_4t1;

architecture my_mux of mux_4t1 is
begin

   with SEL select
   F <= AZ when "11",
        AY when "10",
        AX when "01",
        AW when "00",
        (others => '0') when others;

end my_mux;
```

Figure 20.12: The VHDL model for the solution for Example 20-4.

Example 20-5: Sorting Circuit

Design a circuit that has two 8-bit inputs **A** and **B**, and two 8-bit outputs **GT** and **LT**. If the **A** input is greater than or equal to the **B** input, the **A** input will appear on the **GT** output and the **B** input will appear on the **LT** output. Otherwise, the **B** input will appear on the **GT** output and the **A** input will appear on the **LT** output. Support your solution with a block diagram and any required VHDL modules.

Solution: The key to this classic sorting circuit problem is noticing that there is something similar to a comparator present in the problem as well as some selection logic. The basic comparator circuit needs some modifications in order to make the device usable in this problem. Recall that our basic comparator design only had an EQ output that indicated when the two inputs where equal. We did, however, add some modifications to the comparator in a later example.

The second key to this problem is that we have some "selection" stuff going on in order to "select" the correct inputs to feed to the correct outputs. This implies that there will be a MUX in this design. As always, let's start with a block diagram of the solution; the diagram appears in Figure 20.13.

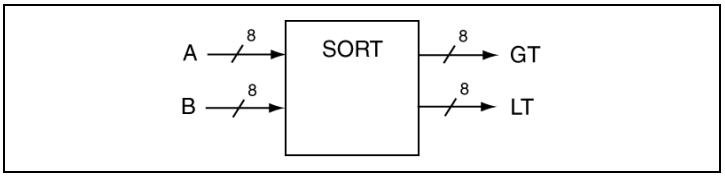
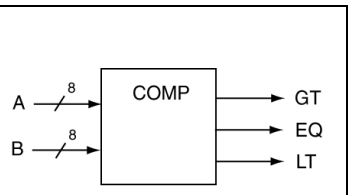


Figure 20.13: Block diagram for Error! Reference source not found.**.**

The next consideration is dealing with the "comparator-type" circuit. What we'll need to do is modify the standard comparator circuit for this problem. We did this in a previous example so we'll not need to say too much about it here. Figure 20.14 shows the VHDL model and block diagram for the required comparator. The comparator we design for this problem does not actually have all three of the outputs shown in Figure 20.14 but we include it in the modified comparator design because it will be useful for other problems.

```
entity comp is
   port (  A,B : in std_logic_vector(7 downto 0);
           GT,LT,EQ : out std_logic);
end comp;

architecture my_comp of comp is
begin
   process(A,B)
   begin
      -- pre-assign output values
      EQ <= '0';  LT <= '0';  GT <= '0';

      if (A = B) then EQ <= '1';  end if;
      if (A < B) then LT <= '1';  end if;
      if (A > B) then GT <= '1';  end if;
   end process;
end my_comp;
```

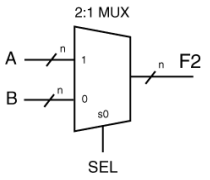


(a) (b)

Figure 20.14: Black box diagram for out new comparator (b), and VHDL model for new full-featured comparator circuit (a).

The next consideration is the data selection portion of the circuit. Once again, the key here is the fact that we used the word "selection" to roughly describe the solution. This heavily implies a MUX of some type. Since the circuit we're trying to design has two outputs, and the MUXes we know about have only one output, it looks as if we're going to need two MUXes. This is exactly the type of MUX used on the second example and is shown again in Figure 20.15.

```
entity mux2t1 is
   port (  A,B : in std_logic_vector(7 downto 0);
           SEL : in std_logic;
            F2 : out std_logic_vector(7 downto 0));
end mux2t1;

architecture mux2t1 of mux2t1 is
begin
   F2 <= A when (SEL = '1') else
         B when (SEL = '0') else
         '0';
end mux2t1;
```



(a) (b)

Figure 20.15: 2:1 MUX with bundle inputs and outputs: VHDL model (a) and block diagram (b)

So we're to the point where we know that we'll need two 2:1 bundle-type MUXes. The only other issue we need to contend with is how we are going to control the two MUXes. What we are interested in is the condition where the A input is greater than or equal to the B input. What we could do for the final circuit is control the data selection function of the two MUXes with an ANDing of the comparator's GT and EQ signals. But a more clever way to do this would be to use the LT signal on the comparator to directly control the MUX. What we need from the MUXes is always to have them choose different outputs. We could do this by connecting the circuit's inputs identically to the MUXes and complementing the MUX control signal from one of the MUXes. A better solution would be to skip the inverter and connect the inputs to the MUXes differently. Figure 20.16 shows a diagram of the final circuit. The final trick in this problem was to be careful when setting up the MUX select signals: it would be easy to get them backwards in this case.

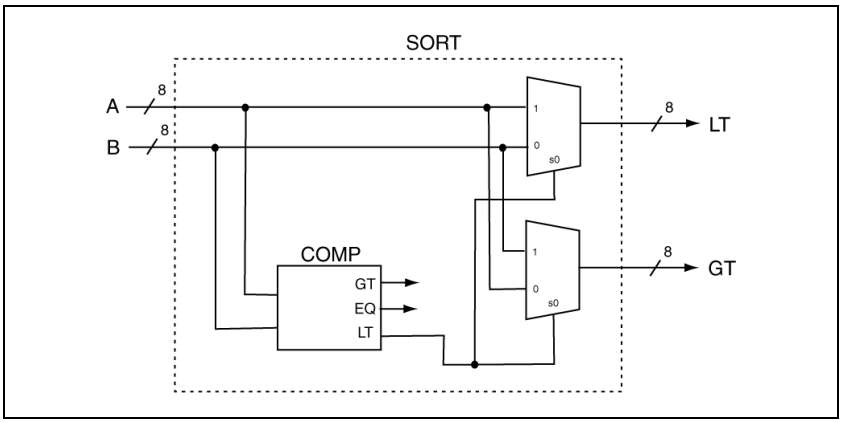


Figure 20.16: The diagram of the final circuit.

Finally, as you may have noticed, this entire problem is easily describable with a simple VHDL model. Figure 20.17 shows the final solution as a single VHDL entity architecture pair. Note that this solution advertises the power of modeling circuits using VHDL in that only a few lines of VHDL code can model the entire circuit. The ">=" operator has been used (greater than or equal to) which is one of the many operators in VHDL. Another thing to note here is that the "less than or equal to" operator is the now infamous "<=" symbol.

```
entity sort is
   port (    A,B : in std_logic_vector(7 downto 0);
          GT,LT : out std_logic_vector(7 downto 0));
end sort;

architecture my_sort of sort is
begin
   process(A,B)
   begin

      if (A >= B) then
         GT <= A;  LT <= B;
      else
         GT <= B;  LT <= A;
      end if;

   end process;
end my_sort;
```

Figure 20.17: The entire solution as a VHDL model.

Chapter Summary

- The multiplexor, or MUX, is a standard digital circuit used to "select" a value. In general, the output of the MUX is one of the data inputs as chosen by the selector inputs. Simple MUX designs are possible using gate-level implementations. VHDL can model MUXes VHDL using many different styles of modeling.

- The MUX has a distinctive shape when it appears in circuit diagram; this shape is always used in circuit diagrams in order to let the reader know a "selection" operation is taking place.

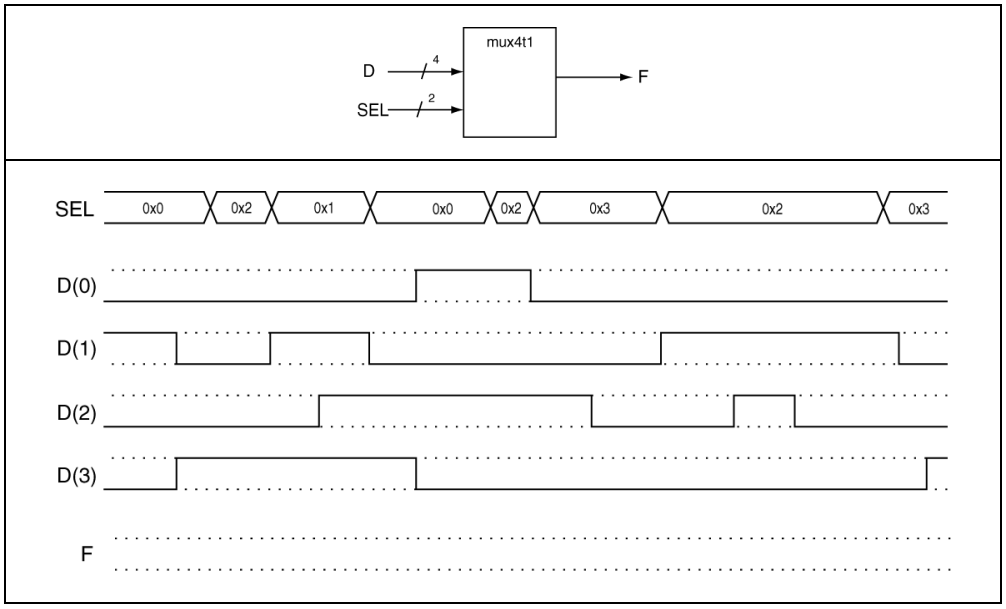
- When modeling MUXes with VHDL, you should always include every possible case in the model and complete the model with some type of catchall statement. Shortcuts in coding do not lead to more efficient modeling and can hamper the debugging process.

- Digital design generally uses MUXes as selection devices. Contrary to computer programming, digital design typically uses hardware to generate all possible results for a given problem and then "selects" the correct result (via a MUX) based on the value of the signal connected to the MUX's data selection inputs.

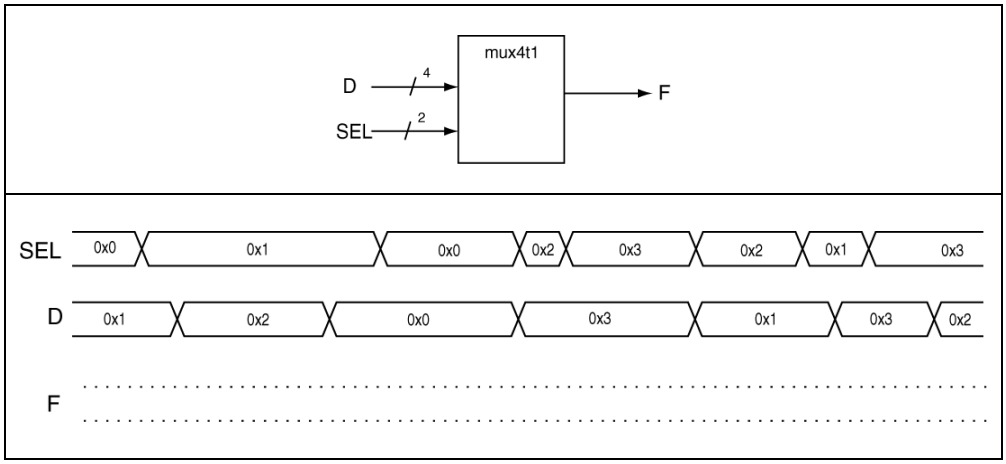
Chapter Problems

1) Briefly describe the special relationship between a MUX and a standard decoder.

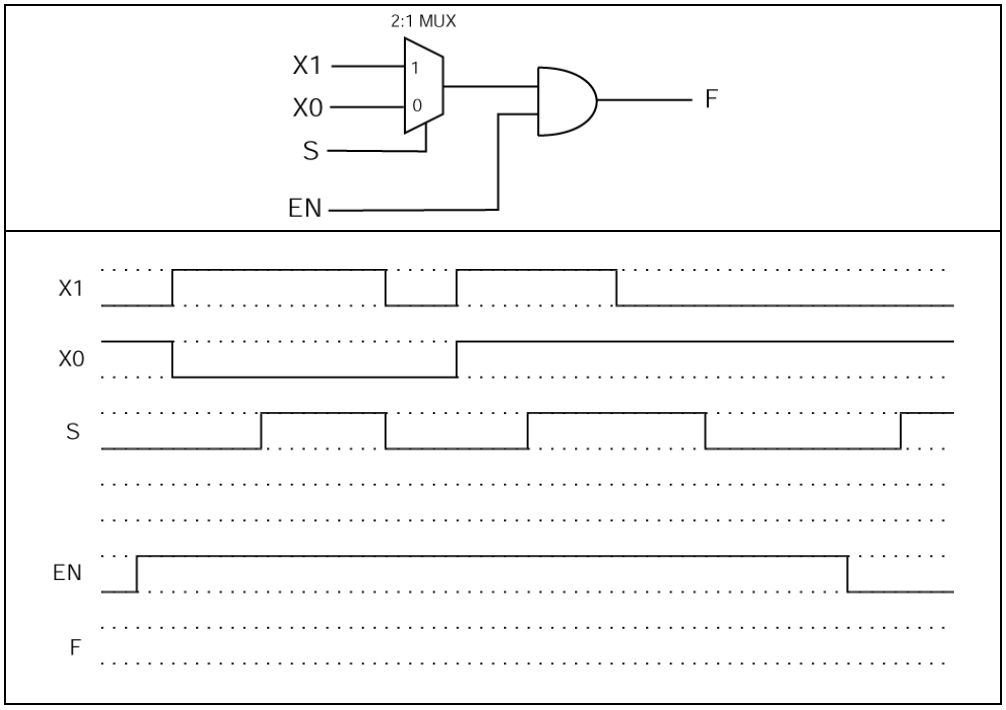
2) Use the following block diagram to complete the provided timing diagram. For this problem, consider the block diagram to represent a basic 4:1 MUX.



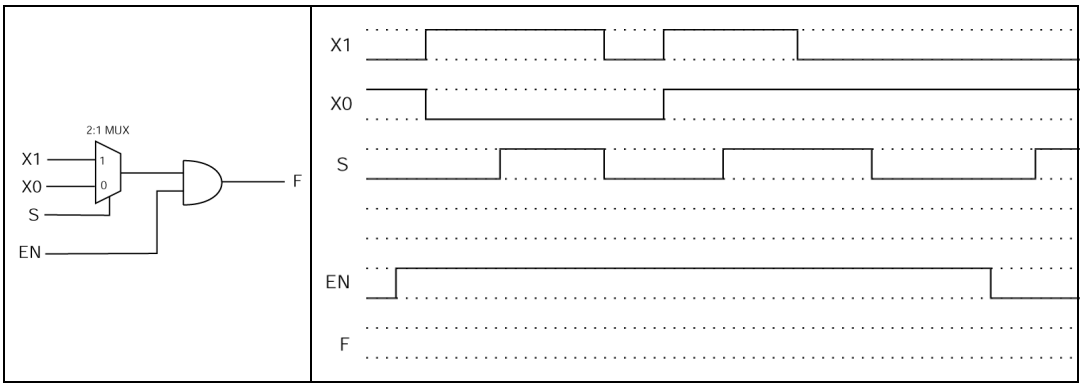
3) Use the following block diagram to complete the provided timing diagram. For this problem, consider the block diagram to represent a basic 4:1 MUX.



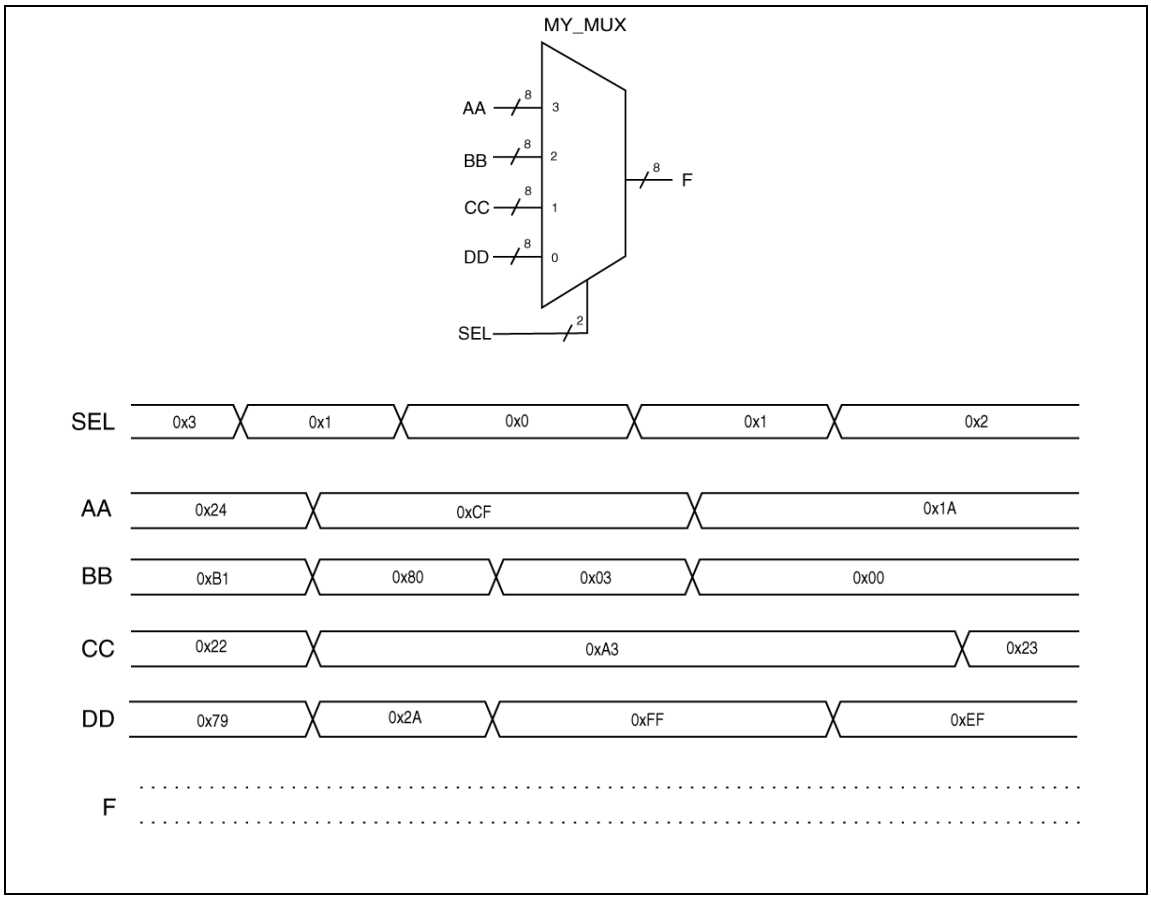
4) Use the listed circuit to complete signal **F** in the following timing diagram.



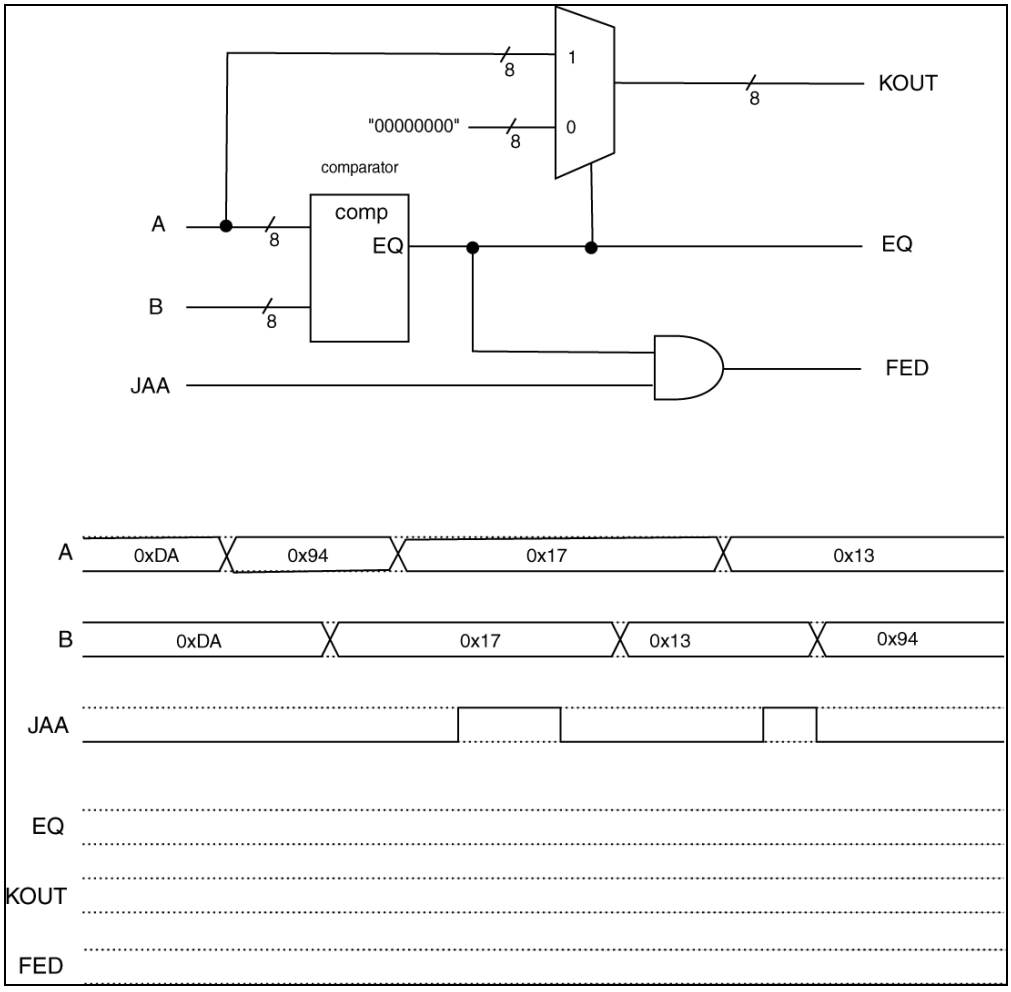
5) Use the listed circuit to fill signal **F** in the following timing diagram.



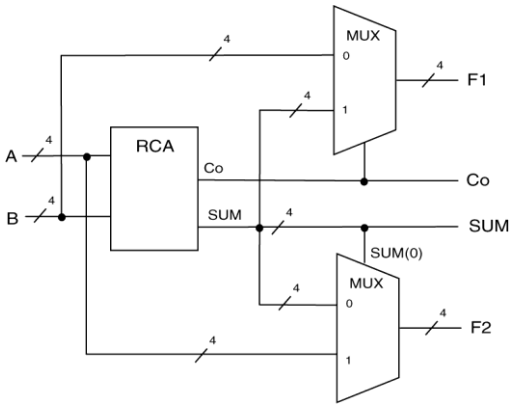
6) Using the following diagram of a 4:1 MUX, complete the provided timing diagram. Also provide a VHDL model that implements the 4:1 MUX.

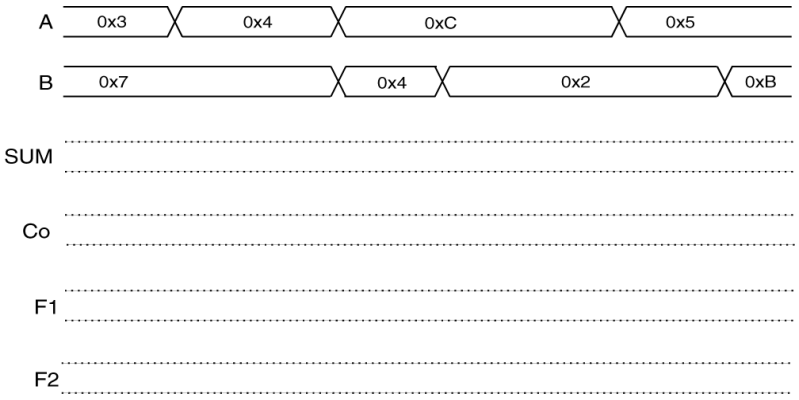


6) Use the following circuit to complete the unlisted signals in the timing diagram. For this problem, assume there are no propagation delays.



7) Use the following circuit to complete the unlisted signals in the timing diagUse the following circuit diagram to complete the empty rows on the accompanying timing diagram. Use bus notation for all bundles (Co is the only non-bundle signal; 0x indicates hexadecimal).


MUX
0
1
F1
RCA
A
B
Co
SUM
SUM(0)
MUX
0
1
F2
4


A
0x3
0x4
0xC
0x5
B
0x7
0x4
0x2
0xB
SUM
Co
F1
F2

Design Problems

1) Design a circuit that outputs one 8-bit value. If the sum of the circuit's two 8-bit inputs, A & B, generate a carry out and are equal, the value of 2A is output; otherwise the sum of 2B is output. Consider the output value to be an 8-bit number also; don't worry about carryouts on the output operations. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

2) Design a circuit that does the following. If the circuit's two 2-bit values (A & B) are not equivalent, then the 8-bit value AA will show up on the circuit's output; otherwise, the 8-bit value BB will show up on the output. Consider AA and BB to be inputs to the circuit. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

3) Design a circuit that outputs one 8-bit value. If the circuits two 8-bit inputs, A & B are equivalent, then the sum of A & B are output; otherwise, the value of A + A is output. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

4) Design a circuit on a block diagram level that performs one of several mathematical operations. Your design should use the standard circuits you've learned about thus far in digital design. Minimize the use of hardware in your design; **use no more than one adder**. Be sure to label everything! The circuit operates as follows:

- Depending on the value of the ***two*** select inputs, the single output should reflect the result of one of the following operations. It does not matter which select values select which operation but make sure the combinations associated with the select inputs can generate each of the following operations:

  RES = A + A
  RES = A + C
  RES = A + B
  RES = B + C

- For this problem, make the following assumptions:

  - Assume inputs A, B, C and the output are all 12-bit values
  - Assume there will no issues or problems with carry out values

5) Design a circuit on a block diagram level with an output that represents either a mathematical operation or another input. Use only standard digital modules in your design. Minimize your use of hardware. The circuit operates as follows:

- if input SEL equals '1', then the circuit outputs the result of the operations A + B + C
- if input SEL equals '0', then the circuit outputs the value of D directly.

o For this problem, make the following assumptions:

- Assume inputs A, B, C, D, and the output are all 12-bit values
- Assume there will no issues or problems with carry out values

6) Design a circuit that has one 8-bit input **A**, a single bit input **BTN3**, and one 8-bit output **F**. Both 8-bit input and output are signed binary numbers in radix complement form. If the value of A is equal to zero, the circuit outputs zero. Otherwise the circuit outputs **A** if **BTN3** is pressed or **–A** if **BTN3** is not pressed. Assume that a button press generates a '1' value for the input. Use only standard digital modules in your design. Minimize your use of hardware.

7) Design the following digital circuit: if the two 8-bit binary numbers (RC) are both positive, they are added and the result of the addition becomes the 8-bit output of the circuit. Otherwise, the circuit's 8-bit output is set to 0. Your design should be on a block level using standard digital module. Be sure to completely label your diagram. Use only standard digital modules in your design. Minimize your use of hardware.

8) Design the following digital circuit; consider all inputs to be 12-bit unsigned binary numbers. If the A and B inputs are equal, ***and*** the C and D inputs are equal, the 12-bit output of the circuit is the sum of A and B. Otherwise, the 12-bit circuit output is the sum of C and D. Your design should be on a block level using standard digital modules. Be sure to completely label your diagram; minimize your use of hardware modules for this design. Don't worry about overflow in this design.

9) Design a circuit that has two 8-bit unsigned binary inputs (**A** & **B**) and one 8-bit unsigned binary output. If both inputs are represent even numbers, are not equal, and the sum of **A** + **B** does not generate a carry-out, then the sum of **A** + **B** is output; otherwise, the value of **B** is output. For this problem, disregard the carry-out on the final sum output of the circuit. If you use anything other than a standard digital circuit, be sure to adequately describe the circuit, but do not use VHDL. Minimize your use of hardware in this design. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

10) Design a circuit that does the following. If the sum of the A input added to the B input is less than or equal to the C input, then the circuit outputs the value of A + C; otherwise, the circuit outputs the value of B + C. Assume all input and output values are 8-bits. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

11) Design a circuit that outputs one 8-bit value. If the sum of the circuit's two 8-bit inputs, A & B, generate a carry out and are not equal, the value of 2A is output; otherwise the sum of 2B is output. Consider the output value to be an 8-bit number also. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

12) Design a circuit that does the following. If the sum of the A input added to the B input is less than or equal to the C input, then the circuit outputs the value of A + C; otherwise, the circuit outputs the value of B + C. Assume all input and output values are 8-bits. Don't worry about carry-outs form the addition operations. Provide a block box diagram for this circuit and label everything with great care and attention. Use only standard digital modules in your design. Minimize your use of hardware.

13) Design a circuit that performs as follows: If both **A** and **B** inputs are both positive or both negative, the circuit outputs a -1 (in signed binary radix complement form); otherwise the circuit outputs the sum of **A** + **B**. Consider both the inputs and outputs to be 8-bit signed binary numbers in radix compliment form. For this problem, disregard any issues having to do with a carry-out. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

14) Design a circuit that has one 8-bit input and three 8-bit outputs. Both the inputs and outputs are signed binary numbers in radix complement form. The circuit's three outputs represent two less than, two greater than, and four greater than the circuit's input, respectively. For this problem, assume the input value is always between 20_{10} and 120_{10}. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

15) Design a circuit that has one 8-bit input, **A**, and two 8-bit outputs. Both the inputs and outputs are signed binary numbers in radix complement form. The circuit's two outputs, **POS_A** and **NEG_A**, represent the negative and positive version of the input value, respectively. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

16) Design a circuit that has two 8-bit signed binary inputs and one 8-bit signed binary output. If both inputs are negative, and the sum of **A** + **B** generates a carry-out, then the sum of **A** + **B** is output; otherwise, the value of **B** is output. For this problem, disregard the carry-out on the final sum output of the circuit. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

17) Design a circuit that performs as follows: If the sum of the circuit's two 10-bit unsigned binary inputs (A, B) generates a carry-out, and both of the two 10-bit inputs are odd, the then the circuit outputs the A input; otherwise, the circuit outputs B input. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

18) Design a circuit that performs as follows: If the circuit's two 10-bit signed binary inputs (A,B) are equivalent, the circuit changes the sign of each number before they are output; otherwise, the circuit outputs the two inputs without changing them. For this problem, you can use a box labeled (2_COMP) which inputs a 10-bit number and outputs the 10-bit 2-s complement representation of that number. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

19) Design a circuit that performs as follows: The circuit has two 10-bit unsigned binary inputs (A,B). If the value of A + 2 (addition) is greater than or equal to B + 5 (addition), the circuit outputs the unchanged A value; otherwise, the circuit outputs the unchanged B value. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

20) Design a circuit that performs as follows: The circuit contains three 5-bit binary inputs and one 5-bit binary output; both inputs and output are in RC form. The circuit outputs the input value that has the largest magnitude of the three inputs. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

21) Design a circuit that performs as follows: The circuit contains three 5-bit binary inputs and one 5-bit binary output; both inputs and output are in RC form. The circuit outputs the input value that has the largest magnitude of the three inputs. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

22) Design a circuit that adds the magnitude of the three 4-bit signed binary numbers (RC form). The circuit's output should be in unsigned binary form with a sufficient amount of bits to accurately represent the required summation. For this problem, assume that -8 will never be included an input value.

23) Design a circuit that performs as follows: The circuit contains a single button input (BTN) and a single 4-bit binary input. The circuit contains one single-bit output. When the button is pressed (input value is a '1'), the circuit treats the 4-bit inputs as an unsigned binary number; the output indicates when the 4-bit input is greater than 8. When the button is not pressed, the circuit treats the 4-bit input as a signed binary number in RC form and the circuit output indicates when this number is negative. Use only standard digital modules in your design. Minimize your use of hardware. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.

24) Design a circuit that adds two unsigned 10-bit numbers (which generates an 11-bit result including the carryout) and is then "scaled" by removing the three least significant bits to form an 8-bit result. Regarding the three least significant bits removed, an input to this circuit decides whether the 8-bit output is the result of a rounding up or truncation operation (for example 31.5 rounds up to 32 and truncates to 31). HINT: $0.1_2 = 0.5_{10}$. Minimize your use of hardware. Use primarily black box models in this design; in other words, severely limit your use of VHDL. Include a black box diagram for both the entire circuit and the underlying circuitry. Fully describe any non-standard sub-modules modules you use in this design.

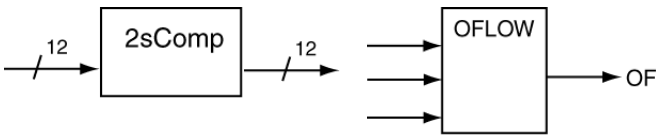
25) Design a circuit that adds four unsigned 10-bit numbers (A, B, C, D). The result should have the minimum number of bits while generating the correct result (including number of bits) of the addition operations. Use no more than three 10-bit RCAs in your design. Minimize your use of hardware. Use primarily black box models in this design; in other words, severely limit your use of VHDL. Include a black box diagram for both the entire circuit and the underlying circuitry. Fully describe any non-standard sub-modules modules you use in this design.

26) Design a circuit that adds two signed 12-bit numbers A & B. If this operation generates no carry and no overflow, then the circuit outputs the result of the operation (A + B). If only a carry is generated without an overflow, the circuit outputs $\overline{A}$; if only an overflow is generated with no carry generated, the circuit outputs $\overline{B}$; if the operation generates both an overflow and carry, the circuit outputs 0x000 (hex). The circuit has an output NO_ERR that indicates when no overflow and no carry is generated. Use the overflow generator model listed below (be sure to connect it properly; you don't need to describe it at a low level). Minimize your use of hardware. Use primarily black box models in this design; in other words, severely limit your use of VHDL. Include a black box diagram for both the entire circuit and the underlying circuitry. Fully describe any non-standard sub-modules modules you use in this design.

27) Design a circuit that adds two signed 12-bit numbers A & B in radix complement form.

- if (A + B) generates no carry and no overflow, then the circuit outputs (A + B)
- if (A + B) generates a carry without an overflow, the circuit outputs $\overline{A}$
- if (A + B) generates an overflow without a carry, the circuit outputs $\overline{B}$
- if (A + B) generates both an overflow and a carry, the circuit outputs (A – B)

The also circuit has an output NO_ERR that indicates when no overflow and no carry is generated. Feel free to use the overflow generator (OFLOW) and/or 2's complement (2sComp) models listed below in your design (you don't need to describe it at a low level). If you use anything other than a standard digital circuit, be sure to adequately describe that circuit, but do not use VHDL. Minimize your use of hardware in this design. Include a black box diagram for both the top-level circuit as well as the underlying circuitry.



21 Chapter Twenty-One

(Bryan Mealy 2012 ©)

21.1 Chapter Overview

The previous chapters were primarily concerned with digital circuits based idealized gate-models. Although we expended considerable effort describing the operation and representation of many different types of circuits, we never really considered some of the physical attributes associated with the actual devices themselves. In this chapter, we'll be dealing with these issues in the context of implementing functions with the techniques we've learned thus far. Timing diagrams are the best approach to representing these physical attributes, which turns out to be an art form of its own.

Main Chapter Topics

- **GATE-LEVEL MODEL:** This chapter introduces the concept of switching times in digital circuits. While previous chapters modeled gates as ideal devices, this chapter presents some of the non-idealized characteristics of gates.
- **CIRCUIT GLITCHES:** The chapter describes glitches and outlines their effect on digital circuit. Although many different circuit conditions can cause glitches, this chapter deals primarily with glitches resulting from static logic hazards. This chapter presents techniques to remove static logic hazards from circuits.
- **TIMING DIAGRAMS:** This chapter revisits timing diagrams and introduces a method for providing useful annotations to increase the readability and understandability of timing diagram.

Why This Chapter is Important

This chapter is important because it introduces non-idealized circuit models. Real digital circuits contain propagation delays, which can cause unwanted characteristics such as glitches.

21.2 Real Digital Devices

Our approach to modeling digital circuits has thus far omitted some of the most important aspects of digital reality in order to smooth-out the learning curve. Although most introductory digital circuit design courses deal with idealized models (ignoring timing considerations), most digital circuit design generally deals with as assortment of timing considerations. Robust digital design requires that the digital designer take into account the actual device parameters that will affect the real world application

of the circuit. These considerations are particularly critical when the timing characteristics associated with physical devices start pushing the speed limits of the devices. As you know, speed, or how fast your digital circuit operates, is an important attribute out there in digital design-land.

A digital circuit is comprised of logic devices that can be modeled at many different levels. Courses in semiconductor physics may model these devices at the molecular level. A course in semiconductor devices may model these devices at the transistor level. A course in world history would most likely not need to model these devices at all. In the previous chapters, you preformed a gate-level modeling of these devices when you applied the iterative-based design approach. You've also modeled circuits on a module level and implemented them using VHDL structural modeling using the iterative-modular and modular-based approaches. Your final approach to circuit design was all the way up at the block level.

The common approach in these methods is the use of *models*. The model essentially refers to the characteristics of the devices that are important to your particular need or application. A model is nothing more than a convenient description of something (as opposed to the real thing). The circuits you have modeled up to this point have ignored most timing considerations. In reality, the circuits you've designed have the ability to operate at speeds in the nano-second range (10^{-9} seconds) yet you were considering them from a functional level only. Although this approach works well for many applications, it won't work in all cases. To put all of this in other words, you have been considering the gates you've been using to be idealized models. We'll view them slightly different starting with this chapter.

Timing issues are a critical in the design of most meaningful digital circuits. The general theme in most areas of digital design is to create circuits that are able to operate as quickly as possible while generating the correct result. The general thought here is to make the circuit "fast" so that you can increase the amount of information your circuit can process.

The underlying goal of practically any digital circuit is to increase the amount of useful information any circuit can process (referred to as *throughput*). The throughput increases as the amount of time it takes inputs to generate the correct output decreases. The problem is that digital logic gates are physical devices and they require a finite amount of time for changes in circuit inputs to effect circuit outputs. This *delay* from the input to the output is the main topic of this chapter.

Another main topic in this chapter is the extended use of timing diagrams. These timing diagrams provide a visual representation of circuit delays. Knowledge of circuit delays and proper use of timing diagrams forms the foundation of proper digital circuit design.

21.3 Timing Diagrams Yet Again

We discussed timing diagrams in a previous chapter so a quick overview won't be too painful. Creating and analyzing timing diagrams is an important area of digital design. The usefulness of timing diagrams ranges from producing a visual explanation of how a circuit operates[257] to providing a valuable debugging tool. Timing diagrams are able to highlight circuit operation beyond a circuit schematic in that timing considerations such as propagation delays, set-up and hold time[258], etc. are easily indicated by the visual nature of the timing diagram. The use of timing diagrams becomes increasingly important as the operating speed of the circuit increases. Moreover, the timing diagram is the primary output of many standard digital design and test tools such as simulators and logic analyzers.

Timing diagrams show the values of signals as a function of time. Time is therefore the independent variable (horizontal axis) while the signal value is the dependent variable (vertical axis). Since we are

[257] Recall that a timing diagram is a viable method to model digital circuits.
[258] You'll learn about these later.

working with digital signals, the output is either "high" or "low". In addition, because signals are true functions in the mathematical sense of the word, a signal can't simultaneously be high and low.

21.4 Gate Delays and Gate Delay Modeling

Up until now, you've used an idealized model for the logic gates you've worked with. The concept of a gate modeled as "ideal" and its comparison to a non-idealized model provides a great vehicle to introduce timing diagrams and the concept of low-level device modeling in general.

Figure 21.1: shows the schematic symbols for an inverter and NAND gate. Figure 21.2 shows two example timing diagrams associated with the schematic symbols of Figure 21.1: using an idealized model for the devices. There are several important things to note about the timing diagrams in Figure 21.2.

- Timing diagrams generally show both the inputs to the device and the outputs. The timing diagrams for these simple devices show all inputs and outputs; timing diagrams may not show all the inputs and outputs for more complicated circuits. Timing diagrams become less readable as the number of listed inputs and outputs increase. In the interest of readability, you'll only list the most interesting signals in the timing diagram.[259]

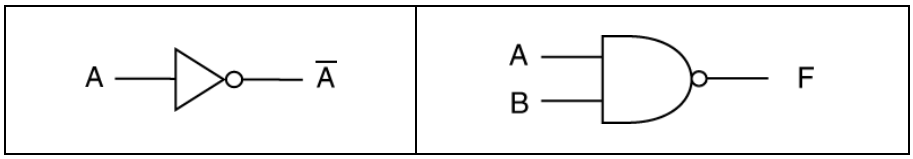
- The horizontal axis represents time. Moving from left to right on this axis represents the passing of time. Time is understood to be the horizontal axis and is rarely labeled in timing diagrams.

- Each of the listed signals is modeled as being either "high" or "low". In reality, signals require a finite amount of time to transition from high to low and thus are sometimes somewhere in-between high and low. Modeling these signals as only high or low is arbitrary but matches the definition of digital. The actual circuit requires a finite amount of time to switch from '0' to '1' and from '1' to '0'. Modeling devices with an infinite slope is generally represents the functionality of a digital circuit and not its true operating characteristics.

- The concept of '1' and '0' is a model too; timing diagrams rarely list the exact values for high and low. The high and low values generally represent voltage levels; these levels will differ depending on the family of ICs used. Once again, the model we're using opts to represent voltage levels as high and low which correspond to high and low positions on the given signals.

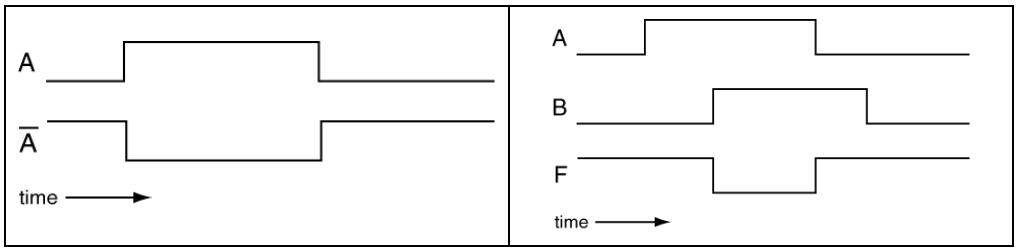
- The ramifications of using an idealized model are that there are no delays shown in the timing diagrams. In other words, the outputs respond immediately to the inputs. Ideal models are nice to work with but not always appropriate especially in the case where the operating speed of the circuit increases to the point where idealized models don't accurately represent the functionality of the circuit.

[259] Timing diagrams represent a mechanism to show the input and output relationship between particular signals of interest: uninteresting signals should be, and usually are, omitted.



(a) (b)

Figure 21.1: Models of the standard inverter (a), and NAND gate (b).



(a) (b)

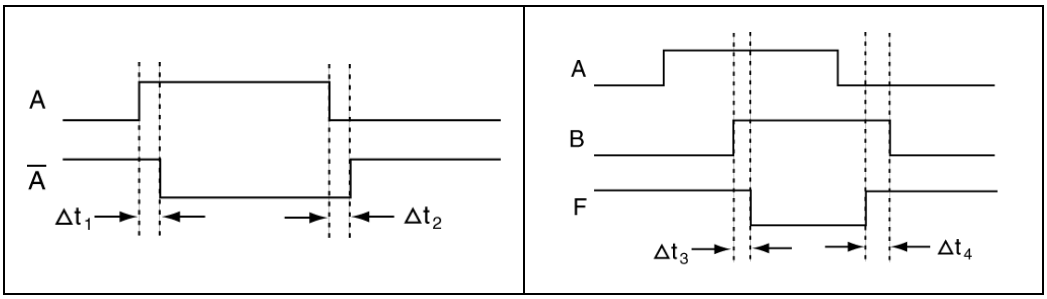
Figure 21.2: Timing diagrams associated with idealized models for the inverter and NAND gate.

Figure 21.3 show timing diagrams that use non-idealized models for the circuit elements. This means that there are delays associated with the signal transitions; the timing diagram clearly shows these delays. These delays represent the amount of time required for an output signal to respond to a change in an input signal. In official digital terms, changes in the input signals require a finite amount of time to propagate to the output. These delay times are referred to as *propagation delays*, or *prop delays*. There are several important things to note about the timing diagrams in Figure 21.3.

- The prop delay times are further broken down into high-to-low transitions and low-to-high transitions. These times are often labeled as t_{phl} and t_{plh}, respectively (or something similar). In Figure 21.3, Δt_1 and Δt_3 are examples of t_{phl} while Δt_2 and Δt_4 are examples of t_{plh}.

- Figure 21.3(b) shows that the values for t_{phl} and t_{plh} are not necessarily equal for a given digital device. In particular, Figure 21.3(b) shows different delay times for Δt_3 and Δt_4.

- The timing diagrams in Figure 21.3 model delay times, but the actual value of these delays is not given. In actuality, datasheets associated with a device you are using provide this information. Different flavors of digital devices will have different delay times. In other words, these devices are implemented using different flavors of transistors; the device characteristics such as propagation delays and voltage characteristics are primarily dependent upon the underlying transistor implementations of the gates.



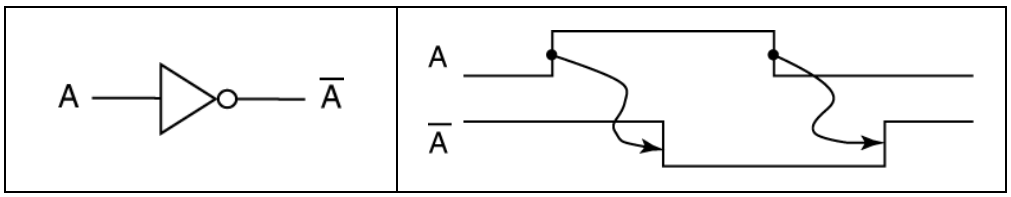
(a) (b)

Figure 21.3: Timing diagrams for inverter and NAND gates that includes delays.

21.4.1 Timing Diagram Annotation

By their nature, timing diagrams provide an abundance of information for a given circuit. The state of each signal listed in the timing diagram is generally present for the entire listed time span in the given timing diagram. In actuality, interesting and important features that are present in the timing diagram often only occur in relatively small areas of the provided time span. In order to draw the viewer's attention to the important portions of the timing diagram, good timing diagrams[260] always use a special type of timing diagram *annotation*.

Figure 21.4 shows the symbology used to indicate causality in timing diagrams. This annotation style shows the relationships between signal transitions throughout the timing diagram. The non-timing lines and arrows drawn in Figure 21.4(b) indicate a relationship between the two timing events. The first arrow shows the low-to-high transition on signal **A** causes the subsequent high-to-low transition on the output of the inverter. Similarly, Figure 21.4(b) also shows that the high-to-low transitions on signal **A** causes the low-to-high transition on the output of the inverter.



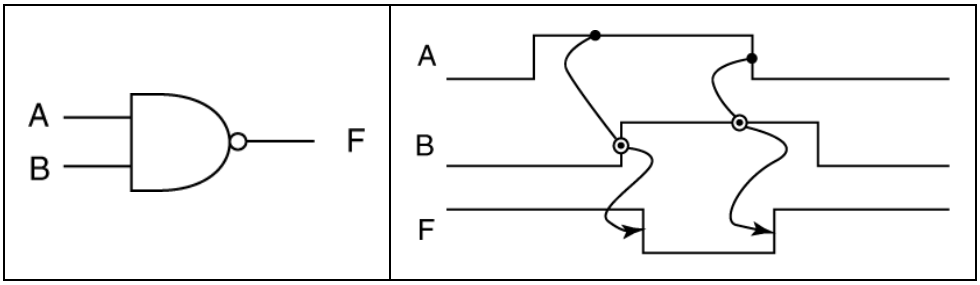
(a) (b)

Figure 21.4: Timing diagram notation for a single input device.

Figure 21.5 shows a slightly more complicated timing diagram. In this case, the *target* symbol indicates when two or more signals are required to cause the switching of another signal. The annotation symbols on the left of Figure 21.5(b) show that the high-to-low transition of signal **F** is caused by the state of **A** and the low-to-high transition of signal **B**. The right annotation symbol in Figure 21.5(b) indicates that the current state of **B** and the low-to-high transition of signal **A** causes the low-to-high transition of signal **F**. This symbol roughly represents a logical AND relationship between the two signals in that both the signals are involved in the listed transition. In other words, signal **A** "and" signal "**B**" must both be high in order for the output to be high. The output transition from high-to-low occurs when both

[260] In this context, "good" means clear and easy to understand.

the input signals are high (plus some time delay as indicated in the timing diagram). Don't confuse this with a bit-wise AND operation.



(a) (b)

Figure 21.5: Timing diagram notation for a multiple input device.

21.4.2 The Simulation Process

The simulation of a digital circuit involves "guessing" how an actual circuit would operate if it were actually implemented. Though "guessing" is not much of a technical term, it's basically what the simulation process is doing. The probability that the guesses made by the simulator are correct generally increase if the devices used in the circuit are accurately *modeled* by the simulator. In this case, the model of a device is essentially a description of the device that the simulator can use when the device is in a circuit. The models used for circuit elements can range from simple to sophisticated depending on your particular requirements.

As you probably can imagine, the more sophisticated circuit models require more processing power by the device performing the simulation and thus the simulation requires more time to complete. On the other hand, devices using simpler models generally see a reduction in simulation time. The basic rule of thumb when using a simulator is that the simulation is only as good as the models used for the simulated devices.

Circuit simulation serves two main functions. First, it is a valuable design tool in that circuits can be designed and tested before the circuit is actually implemented. Problems can be then be detected and corrected before implementing the actual circuit, thus saving valuable engineering costs and irreplaceable party time. Secondly, currently implemented circuits can be modeled in order to debug portions of the circuit that may not be directly available to the outside world. The timing diagrams generated from circuit simulations can be compared against the output from other test devices such as Logic Analyzers for the verification of proper circuit models and the correct circuit outputs.

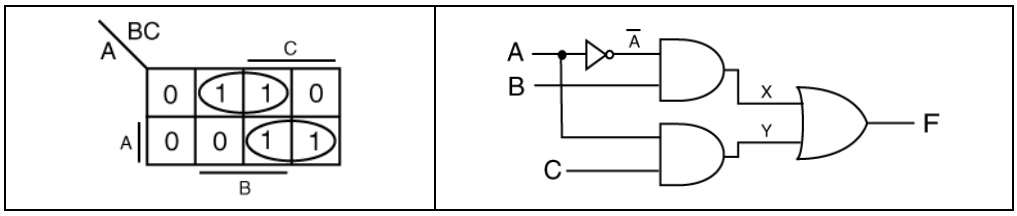
21.5 Glitches in Digital Circuits

A *glitch* in a circuit is defined as a momentary error condition. In other words, the digital output of some circuit element is momentarily high when it should be low (or vice versa). Glitches in circuits can be caused by many different conditions such as switch bounce, electro-magnetic interference, sun spots, demonic possession, and apathetic lab partners. The presence of glitches in your circuit can be of grave concern in that these unwanted signal transitions can cause incorrect and intermittent errors in the operation of your circuit. We'll only be considering glitches caused by *static logic hazards*.

21.5.1 Static Logic Hazards

Static logic hazards fit in nicely with the other material discussed so far in this chapter. The presence of a *hazard* in your circuit is an indication that a glitch *may* occur in your circuit under certain circuit conditions. The root cause of static logic hazards is unequal circuit delays[261] in the devices used to implement the circuit. In this section, you'll learn to detect and correct for those conditions. The approach you'll use also serves as a good review of K-map techniques and a bolstering of your K-map skills.

Figure 21.6(a) shows a K-map with an arbitrary function. This K-map shows the groupings associated with standard K-map reduction techniques. Figure 21.6(b) shows the circuit associated with the reduced equation generated from the K-map of Figure 21.6(a). The interior signals of the circuit contain labels that are used the timing diagram of Figure 21.7. The timing diagram in Figure 21.7 highlights two input signal transitions and the effect these transitions have on the output and interior signals of the circuit. In particular, the timing diagram of Figure 21.7 shows the input transitions of **ABC** = "111" to **ABC** = 011" and **ABC** = "011" to **ABC** = 111". In other words, we're only interested in signal **A** changing while signals **B** and **C** remain constant at their high levels.



(a) (b)

Figure 21.6: The standard K-map approach to function reduction and the resulting circuit.

The two transitions shown in Figure 21.8(a) are generated when the **A** input changes while the **B** and **C** inputs remain at their high values. The numbers in the cells of the K-map represent the output value of the function. As indicated in Figure 21.6(b), **F** is the output of the given function. Since the output does not change (remains at '1') during the two transitions listed, you would expect the output of the circuit to remain in its high state. As you can see in Figure 21.7, the output temporarily goes to its low value before returning to the expected high state. This unexpected signal transition in the circuit output is the glitch we've been talking about.

Unequal circuit delays in the circuit of Figure 21.6(b) cause a glitch in the output of the circuit. The annotations labeled in Figure 21.7 provide the explanation of the undesirable event. In the timing diagram, the prop delays for the various circuit elements are model as being equivalent. The numbers listed below reference the circled numbers shown in Figure 21.7.

1) This signal is the inversion of signal **A**. The high-to-low transition of signal **A** causes the low-to-high transition of the $\overline{A}$ signal after an appropriate delay.

2) The high-to-low transition of signal **A** causes the high-to-low transition of signal **Y**. Signal **Y** is an ANDing of signal **A** and **C**.

3) The signal labeled **X** is an ANDing of signals $\overline{A}$ and **B** The low to high transition of signal $\overline{A}$ causes the low-to-high transition of signal **X**.

[261] Note that this statement is "unequal circuit delays"; claiming that glitches are caused by "circuit delays" only is not accurate and somewhat misleading.

4) The high-to-low transition of signal **Y** causes the high to low transition of the output signal **F**. This is because the output signal **F** is an ORing of signal **X** and **Y**.

5) The low-to-high transition of signal **X** causes the low-to-high transition of the output signal **F**. This is once again due to the fact the **F** is a result of ORing signals **X** and **Y**.

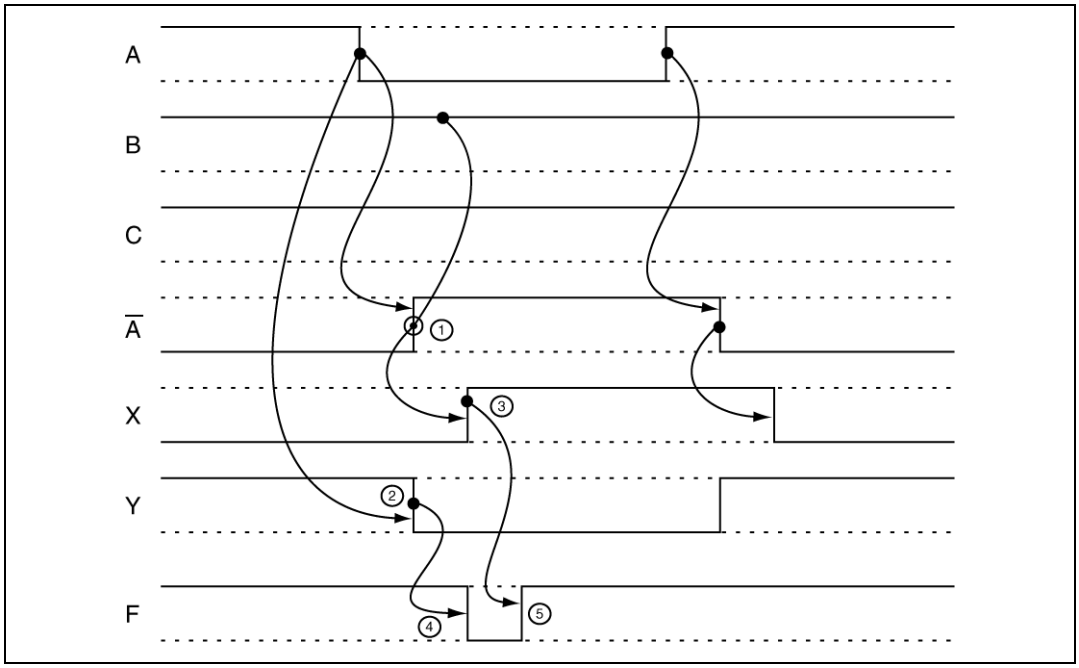
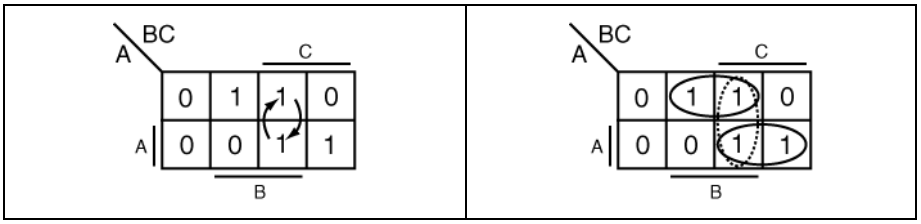


Figure 21.7: The timing diagram generated from the circuit of Figure 21.6.



(a) (b)

Figure 21.8: The transition of interest and the associated cover term for the function.

The glitch shown in signal **F** of Figure 21.7 is referred to as a static logic hazard. This particular hazard is referred to as a static '1' hazard; there are also static '0' hazards which are associated with '0' to '0' transitions in the POS circuit implementations. There are two major types of hazards: *function hazards* and *logic hazards*. These two types of hazards differ by the number of input changes that cause them.

A change in a single input caused the hazard in the previous example, which is what makes it a logic hazard. If more than one input changed simultaneously and caused a hazard, it is referred to as a *function* hazard. As you'll soon see, it's not a big deal locate and correct logic hazards because of the

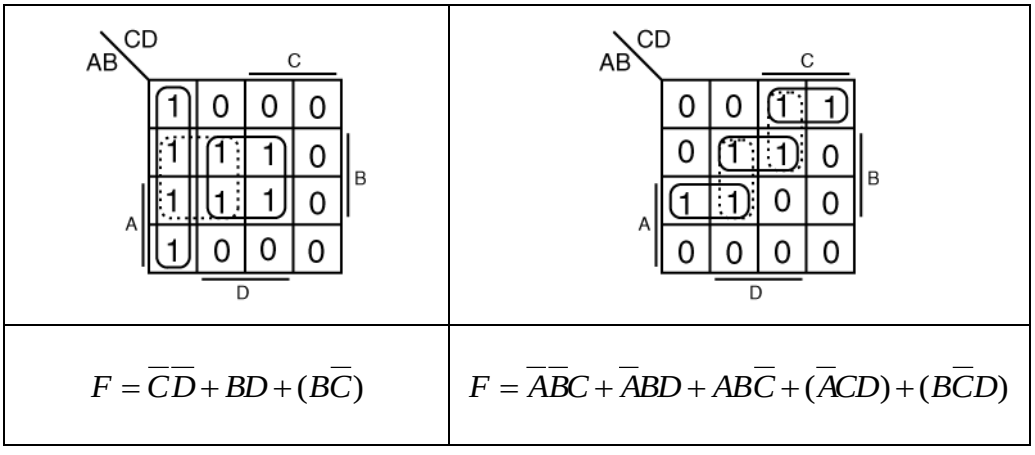
unit distance property of K-maps. A unit distance move in a K-map is (a move in the compass directions: north, south, east, or west) is defined to be caused by a change in a single input variable. Any move in the K-map that is not from one cell to an adjacent cell is associated with a changing of more than one input variable.

Logic hazards can be broken into two sub-types: static hazards and dynamic hazards. These terms describe the desired state of the output when an input change occurs. The previous example is referred to as a static logic hazard because the output was not expected to change because of the input change (in other words, it was supposed to remain static). The previous example was a static '1' hazard because the output should have remained high. Glitching could also occur in when the output is expected to change as a result of the input transitions. In this case, the hazard would be referred to as a dynamic logic hazard.

The key to removing static logic hazards is to ensure that all input transitions between the circled variables terms in a K-map remain in the same grouping. This essentially requires that you include other groupings in addition to the groupings that provide the best possible minimization of the function. Although your function will no longer be in reduced form, it will be free of static logic hazards.

Another way to view this approach is that you must include some *nonessential prime implicants* in your grouping. This approach ensures that each change in input variable is contained in a grouping of its own. The reason this approach works is that the each of the input variable transitions are "covered" by a product term due to the way the variables were grouped. This ensures that each input transition is associated with a gate that is not affected by the change in that input variable. You can remove the static logic hazard in the above example by including the grouping indicated with dotted lines in Figure 21.8.

Figure 21.9 shows two examples of removing potential glitches with the inclusion of cover terms in the K-map reduction. The associated expression lists the normal product terms without parenthesis while parentheses surround the product terms associated with the cover terms. Note that by the inclusion of the cover terms, every '1' to '1' transition is included in a K-map grouping. While the resulting equation is not maximally reduced when the cover terms are included, the resulting circuit will be free of static '1' logic hazards.



$F = \overline{C}\,\overline{D} + BD + (B\overline{C})$

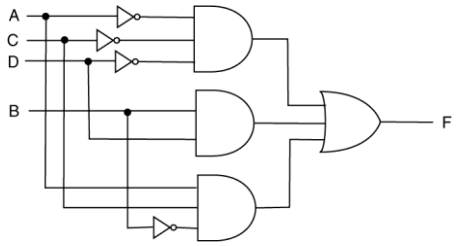
$F = \overline{A}\,\overline{B}C + \overline{A}BD + AB\overline{C} + (\overline{A}CD) + (B\overline{C}D)$

(a) (b)

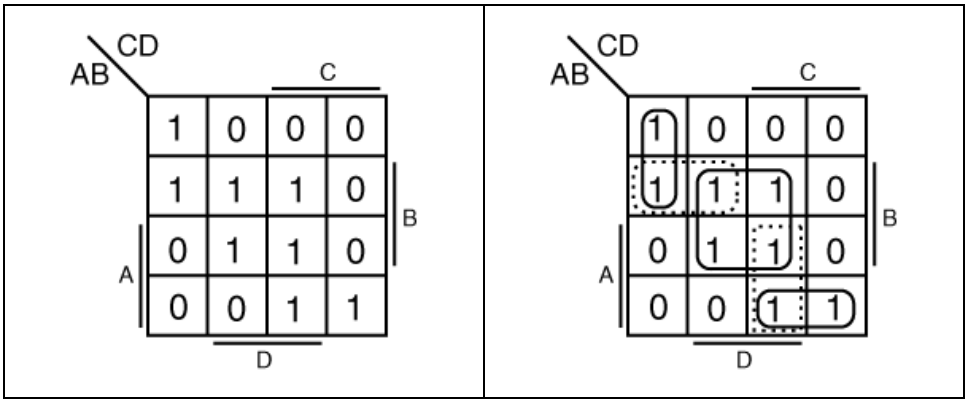
Figure 21.9: Two more examples of removing potential glitches with cover terms (in parentheses).

Example 21-1

List the product terms that need to be included in the following circuit that remove all static logic hazards.



Solution: The only tool you currently know of to detect static logic hazards is the K-map. The mode of attack for this problem is to first translate the circuit model provided in the problem to a K-map representation. Figure 21.10(a) shows the K-map model corresponding to the circuit model shown in the problem description. Figure 21.10(b) shows the K-map with the standard groupings (solid lines) and the groupings required to remove the static logic hazards. Equation 21-1 shows the product terms required to remove the static logic hazards from the original circuit. These two product terms are now included in the original circuit.



(a) (b)

Figure 21.10: The initial (a) and completed (b) K-maps for Example 21-1.

$$\overline{A}B\overline{C} + ACD$$

Equation 21-1: Product terms required to remove static logic hazards.

Chapter Summary

- Timing diagrams show the state of digital signals as a function of time. Timing diagrams are useful for describing circuit behavior and verify correct circuit operation in simulation and actual circuit testing. Circuit operation in general cal be modeled using a timing diagram.

- Timing diagrams can quickly become large and messy. Special annotation techniques are available to increase the readability and understandability of timing diagrams. The key to annotating timing diagrams is to direct the reader's eye to the important portions of the timing diagram.

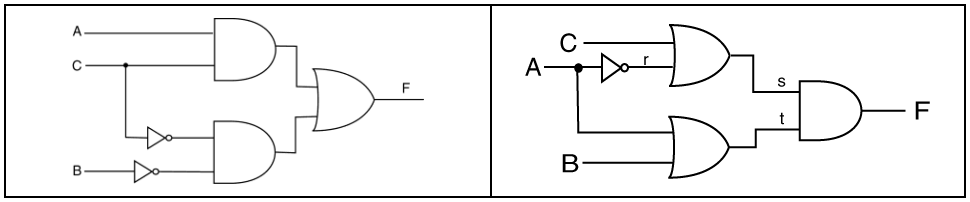
- Real digital circuit elements necessarily have delays associated with their operation. Although some applications do not need to consider delay characteristics of real devices, these factors become more important as the operating speed of circuits increase. Digital device delays can be clearly modeled using timing diagrams.

- Glitches are momentary error states in the outputs of digital circuits. Glitches are caused by many different conditions present in digital circuits.

- Static logic hazards are a *potential* causes of glitches in digital circuits. Static logic hazards can be detected in K-map representations of circuits. Extra hardware can be strategically added to a circuit to remove static logic hazard; this extra hardware is generally referred to as adding cover terms.

Chapter Exercises

1) Using timing diagrams to explicitly show whether the given circuit contains glitches due to static logic hazards. If glitches are present, re-implement the circuit in such a way as to remove the static logic hazards.



(a) (b)

2) Generate a timing diagram that shows the static-1 logic hazard present in the maximally reduced SOP implementation of the following function:

$$F = (A,B,C,D) = \sum(1,5,9,13,14,15)$$

3) List the cover terms that would be required to eliminate the static-1 logic hazards from the following function if it was implemented in reduced SOP form.

$$F(A,B,C,D) = \sum(0,3,4,5,7,10,11,13)$$

4) Write an expression for the following function in reduced SOP form. Write the final expression such that the implemented circuit will contain no static logic hazards.

$$F = (A,B,C,D) = \sum(1,3,4,6,8,9,11,14)$$

5) Reduce the following function. Then, show what term(s) need to be included to remove any static logic hazards. Don't use XOR gates.

$$F(A,B,C,D) = \sum(0,2,6,7,8,10,14,15)$$

6) Reduce the following function. Add product terms to prevent glitches caused by static logic hazards. Don't use XOR type gates.

$$F(A,B,C,D) = \sum(1,3,5,7,10,11,12,13)$$

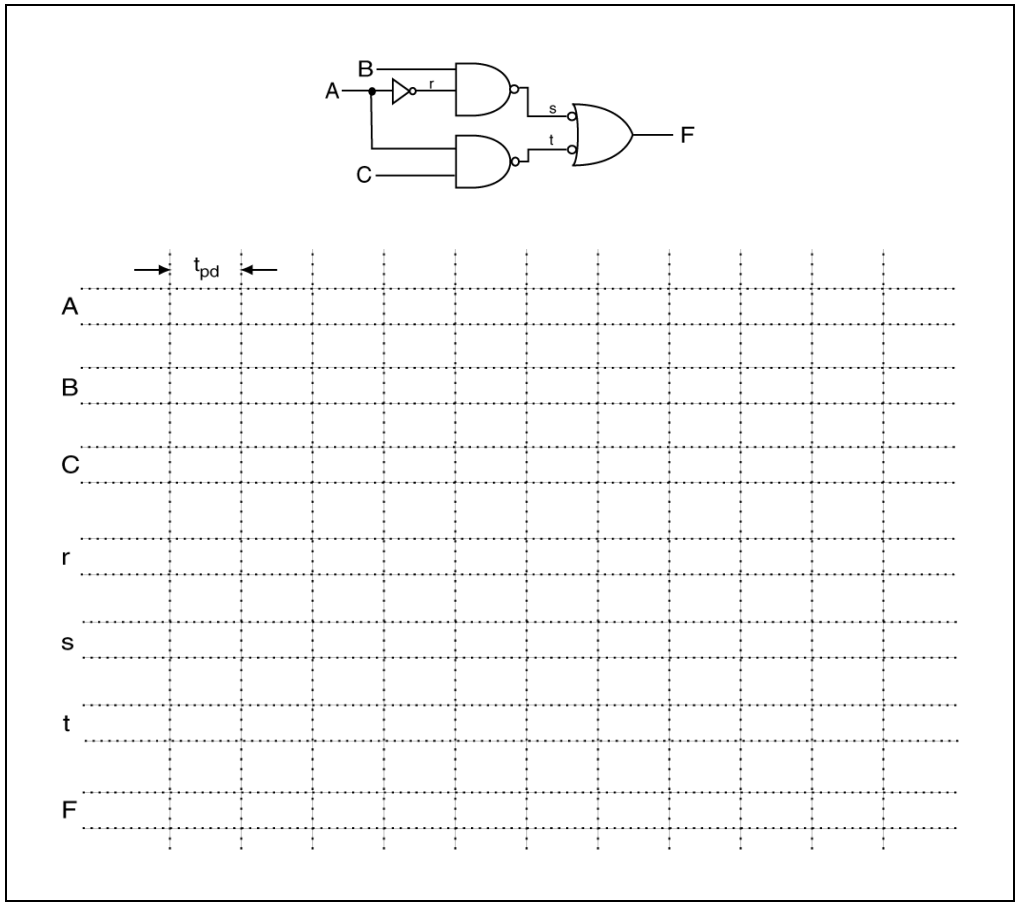
7) Write an expression for the following function in reduced SOP form. Make sure the function contains no static logic hazards.

$$F = (A,B,C,D) = \prod(1,2,3,6,8,9,10,11,12)$$

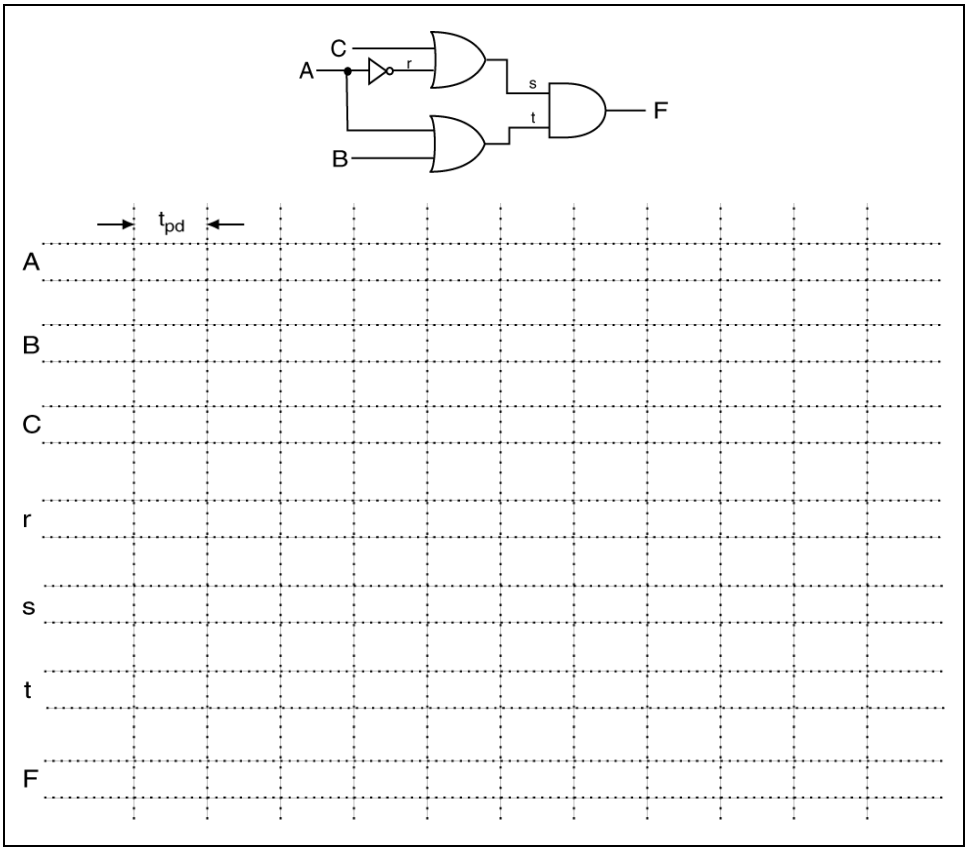
8) Write an expression for the following function in reduced SOP form. Make sure the function contains no static logic hazards.

$$F(A,B,C,D) = \sum(2,6,7,8,13,15) + \sum m_d(0,10)$$

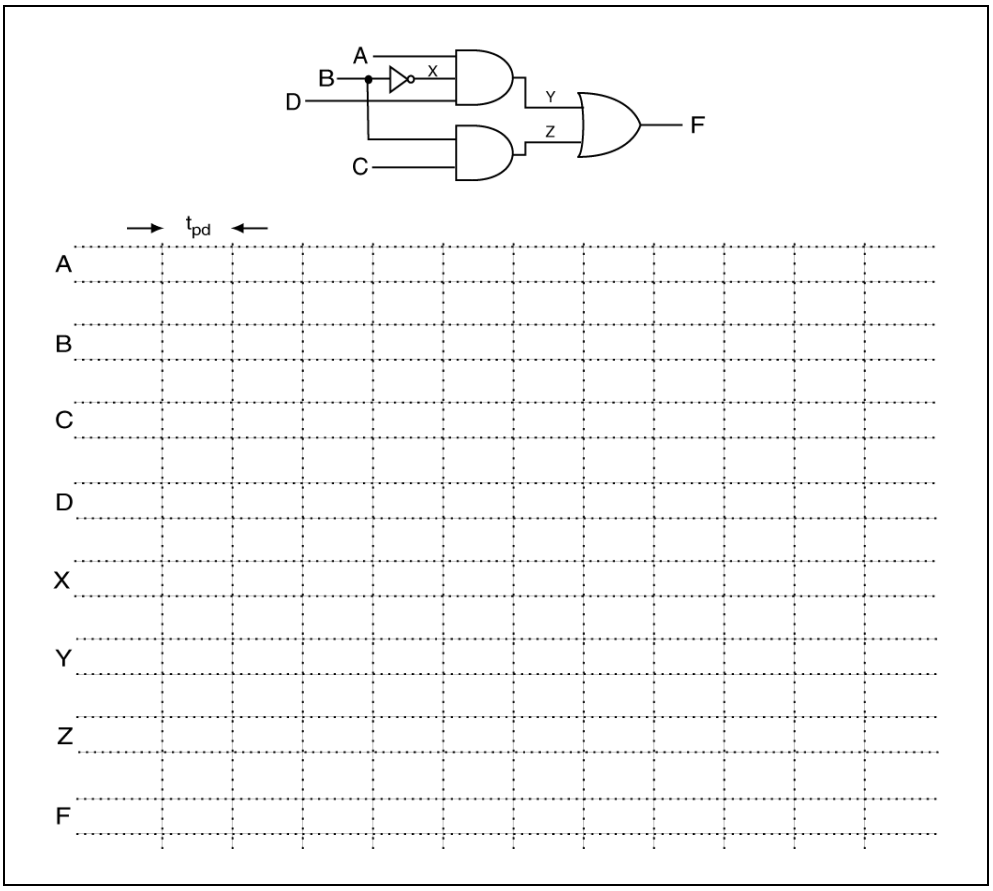
9) Using the timing diagram provided below, show whether the listed circuit contains a static logic hazard. If the circuit does contain a logic hazard, use the provided timing diagram to explicitly show the associated glitch. Assume each of the circuit elements contain equivalent propagation delays, t_{pd}, and are indicated with vertical dotted lines in the provided circuit diagram.



10) Using the timing diagram provided below, show whether the listed circuit contains a static logic hazard. If the circuit does contain a logic hazard, use the provided timing diagram to explicitly show the associated glitch. Assume each of the circuit elements contain equivalent propagation delays, t_{pd}, and are indicated with vertical dotted lines in the provided circuit diagram.



11) Using the timing diagram provided below, show whether the listed circuit contains a static logic hazard. If the circuit does contain a logic hazard, use the provided timing diagram to explicitly show the associated glitch. Assume each of the circuit elements contain equivalent propagation delays, t_{pd}, and are indicated with vertical dotted lines in the provided circuit diagram.



22 Chapter Twenty-Two

(Bryan Mealy 2012 ©)

22.1 Chapter Overview

If you truly want to be a digital design goddess or god, you need to have a complete understanding of digital logic. While it is true you can go a long way by only pretending you understand mixed logic, you'll eventually run into it and be really bummed that you don't really understand it. It's highly unlikely that any system you work with will only use one type of logic; so you really need to be able to face the dilemma of designing and/or interfacing digital circuits in a mixed logic environment.

During the previous several chapters, we've covered a lot of digital design topics. However, the chapter verbage has not thus far presented the entire story regarding the ins and outs of digital logic design. This chapter aims to provide you with the final piece of the basic logic puzzle. Your knowledge and experience with digital design is such that you can finally digest this new material.

Main Chapter Topics

- **MIXED LOGIC:** This chapter provides an in-depth summary of mixed logic digital design. This introduction includes a description of the underlying theory which is later applied in both circuit design and circuit analysis problems.

Why This Chapter is Important

This chapter is important because it provides the theoretical foundation for designing and analyzing mixed logic circuits.

22.2 Mixed Logic Overview

The one thing that stands out most in my mind regarding mixed logic was a comment one of my teachers made when I was taking digital design. The comment was: "*nobody really understands mixed logic*"[262]. Although it has been a struggle for me, I feel I'm making headway into understanding mixed logic. My latest spin on the topic is that, although the stuff is strange, it's becoming more and more doable each time I look at it.

What I've come to realize is that the reason that "nobody understands" this stuff is two-fold. First, I've never run into a text that explains the topic in a manner that I could understand. This is particularly a problem when you're seeing the material for the first time as is the case with beginning digital design

[262] Spoken sometime in the haze of the late 1980's.

students. Secondly, it's a topic that is not overly used in modern digital design. In reality, you generally only deal with mixed logic if you really need to and expend a significant amount of effort attempting to avoid dealing with it.

Although mixed logic is a topic present in just about any digital logic circuit, you generally build up a repertoire of techniques to deal with it when you run into it so that you really don't need to understand it. These techniques are generally good enough because you rarely run into mixed logic in any great depth, but odds are that you'll eventually run into it. More likely than not, you'll act like most digital designers and find yourself hoping that you can avoid dealing with mixed logic issues in all of your digital circuit designs.

22.3 Chapter Overview

The underlying theme of all digital logic deals with the basic interpretation of signals. A signal in a digital circuit is either at a high or low voltage level[263]. We've been modeling these high and low voltage levels thus far with a '1' or a '0'. A given signal is generally the output of one gate or device in the circuit; this given signal is generally the input to another device in the circuit.

In truth, digital circuits are extraordinarily dumb: the gates in a digital circuit do nothing more interesting than having outputs that react to the gate inputs. Here's the whole story in a few sentences: the way we've been modeling our circuits so far is that a '1' represented the *action state* or *active state* of things while '0' represented the *non-action state* or *inactive state*. In other words, when the circuit inputs represented a combination that did something we were looking for, we assigned a '1' to the output. In yet other words, the '1' or the high state, was generally taken to mean something affirmative or positive.

The entire strangeness that encircles mixed logic is that fact that sometimes '1' does not represent the active state. Sometimes the '0' state is the active state and '1' represents the inactive state. While you have a choice of designing your circuits anyway you want (with either '1' or '0' being the active state), sometimes you need to design your circuit to interface with another circuit that is interpreting the 1's in 0's in a different manner than your circuit. Thus, you're faced with not just coming up with a digital design, your faced with coming up with a *mixed logic design*.

The same argument is applicable to the outputs of your circuit: sometimes the outputs of your circuit drives a circuit that is interpreting the 1's in 0's different from your design. You need to be able to both understand and handle both of these cases. Remember, digital logic gates are really dumb; they only react to voltage levels. Your digital circuit designs are based on logic levels, and not necessarily voltage levels; this is a really important distinction. It's your mission as a digital designer to ensure that your circuits are reacting in a way that lands you a promotion after your company has finished its next design. The choice is up to you.

Let's use a few short definitions to sum up the concepts presented in the previous paragraphs. We're presenting these definitions here so that we can use them throughout this chapter. The terminology is quite common out in digital-land so being familiar with them is considered a good thing: a real good thing. You'll for sure want to refer back to them as you read on in this section.

- **Positive logic:** positive logic is when the '1' state of a signal represents the active state.
- **Negative logic:** negative logic is when the '0' state of signal represents the active state.

[263] Note that we stay general here by not mentioning the exact voltage levels; it's suffice to leave it at high and low voltage. The notion here is that some external entity has pre-decided what the voltage levels are.

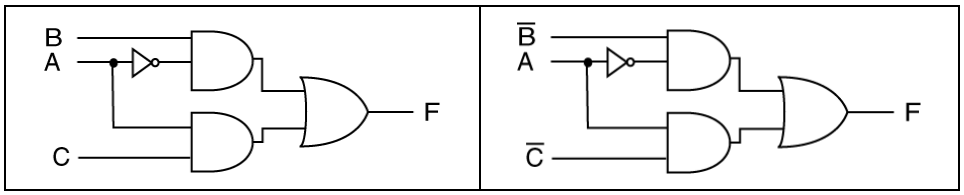
- **Mixed logic:** a term referring to the use of both negative and positive logic in a digital circuit or system.
- **Assertation levels:** assertation levels are an indirect reference to the form of logic used in a circuit. These definitions lead to a common digital vernacular in referring to a signal as being "asserted" or "not asserted" (defined below).
- **Asserted high:** Another way of referring to a positive logic signal
- **Asserted low:** Another way of referring to a negative logic signal
- **Logic levels:** same thing as assertation levels
- **Asserted signal:** a signal that is currently in its active state (independent of the logic levels). A positive logic signal is asserted when it is in a high state while a negative logic signal is asserted with it is in a low state.
- **Not-asserted signal:** a signal that is currently in its non-active state (independent of logic levels). A positive logic signal is not-asserted when it is low while a negative logic signal not asserted when it is high.

The first thing we need to do here is to convince you that the circuits you've been working with thus far have all been positive logic circuits. Figure 22.1(a) shows a circuit that appears to be a typical circuit you've been working with for awhile now. What you may not realize is that by the way the circuit appears in Figure 22.1(a), the inputs to the circuit and the output of the circuit are all positive logic. In other words, a '1' appearing on the circuit inputs and/or the circuit output indicates an active state. When a '1' appears on the output of the circuit, the circuit is indicating some positive condition (vice versa for a '0' appearing on the circuit outputs).

The question that arises is how exactly should we represent negative and positive logic in a circuit? There are actually two ways: the Positive Logic Convention (PLC) and Direct Polarity Indicators (DPI). For this discussion, we'll only be use DPI since it is easier to deal with while learning mixed logic.

The reality is that you've been dealing with PLC ever since your introduction to digital logic. The PLC uses overbars on signals to indicate that they are negative logic (with positive logic represented by not having an overbar). For example, the circuit in Figure 22.1(a) contains three input variables and one output variable. Since none of these variables has overbars on them (and they don't have direct polarity indicator either), they are interpreted as being positive logic.

Figure 22.1(b) shows an example of a similar circuit that uses mixed logic; note that in this circuit two of the inputs contain overbars. This indicates that while signal **A** is a positive logic signal, signals **B** and **C** are negative logic signals. The important point here is that all the signals listed in both of these diagrams are represented using Boolean variables: **A**, **B**, **C**, and **F**. Because these variables are Boolean variables, they can represent either 1's or 0's. Mixed logic once again refers to the fact that sometimes a '1' and a '0' have different meanings. The confusing aspect of mixed logic design lies in the fact that the logic gates only react to voltage levels and know nothing of the logic levels intended by the circuit designers. Although the two circuits shown in Figure 22.1 look similar, *they perform different logic functions*. What exact logic functions they perform is what we'll try and figure out in the remainder of this section.



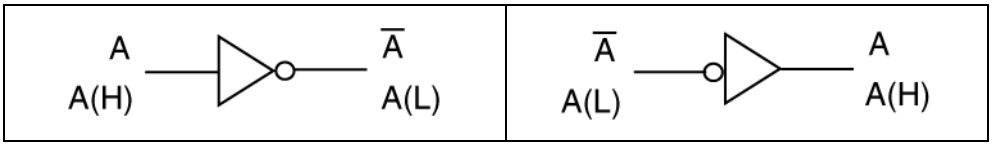
(a) (b)

Figure 22.1: Some generic circuits just for the heck of it.

Assuming there is a good place to start in mixed-logic land, looking at the inverter would be that place. The approach we've taken to inverters up to this point is to think of them as devices that change 1's to 0's and 0's to 1's. While this model of an inverter is valid, we need to model it in a different form in order to gives us a foundation for understanding mixed logic. Figure 22.2 shows our new approach to modeling an inverter.

Figure 22.2(a) shows an inverter drawn as you're used to seeing it. The thing that is somewhat new about this diagram is the PLC and DPI indicators provided above and below the signals, respectively. These two forms of notation are used to indicate that we're no longer thinking of the inverter as a device that toggles a signal value. The new view of an inverter is that it *changes the logic level* of a signal. In other words, if the input to an inverter is a positive logic signal, the output of the inverter is a negative logic signal (and vice versa). We use the notation included in Figure 22.2(a) to indicate this notion.

With the PLC convention (the notation above the signal lines), the **A** without the overbar indicates the signal is positive logic. On the output of the inverter, the **A** has an overbar, which indicates it is a negative logic signal. We can also express the same model using DPI notation, which we list under the signal in Figure 22.2(a). With the DPI notation, the **A** signal is clearly indicated with a directly polarity indicator of H (indicating positive logic) on the input of the inverter. Once it passes through the inverter, the direct polarity indicator changes to L (indicating negative logic). These are important points; you may want to read them again or have some friends read them to you as you contemplate the vastness of the concept.



(a) (b)

Figure 22.2: A different approach to modeling an inverter.

That was not too painful, was it? Now we need to pass a slight amount of terminology by you. For the sake of this discussion, we'll only discuss the DPI convention due to the fact that the polarity indicators are somewhat easier to working with when you're first struggling with these concepts. Once you understand the DPI convention, using either DPI or PLC (or both) should not be a problem. Figure 22.3 shows the terminology we need to work with.

What we're trying to do here is present some tools to effectively deal with the more complex mixed logic circuits that arise later. The intentions here are good even though it all may seem strange at this point. What the equations in Figure 22.3 are saying is that there is more than one way to represent both negative and positive logic using the DPI convention. In other words, you can indicate a positive logic signal as an equivalent negative logic signal (Figure 22.3(a)) and you can write a negative logic signal

as an equivalent positive logic signal (Figure 22.3(b)). These probably don't seem to useful now, but they become worth dying for later. The equations of Figure 22.3 are *equivalent forms* of the signals.

$A(H) = \overline{A}(L)$	$A(L) = \overline{A}(H)$
(a)	(b)

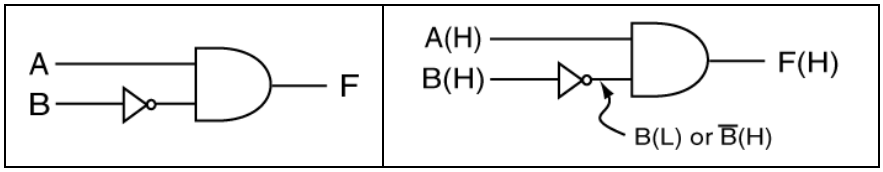
Figure 22.3: Equivalent signals relating to inversion.

Now let's apply these concepts in a manner that you've already seen. You've drawn bunches of circuit diagrams already and many of them used inverters. A two-input AND gate with an inverter sitting in front of one of the inputs as is shown in Figure 22.4(a) is therefore nothing new. But let's reanalyze it using our budding love of mixed logic.

Figure 22.4 shows a simple circuit implementing a product term. We've attached a DPI convention to the inputs and outputs of this device; both inputs and the single output are positive logic signals. The inverter changes the logic level of the B signal before it enters the AND gate. In the end, as you're used to thinking about it, the logic expression $A\overline{B}$ is implemented. Note that we suddenly magically switched to PLC notation, which is what you're used to dealing with.

In reality, the AND gate used in Figure 22.4 has an output that only becomes a '1' when both inputs are '1'. The question that arises is this: what is the relation between the product term $A\overline{B}$ and having both inputs being a '1' in order for the output to be a '1'? What we need to do in this product term is have the output be a '1' when both inputs are in their active state. This means that we need to have **A** be positive logic and **B** to be negative logic as they are input to the AND gate.

The problem is that **B** is a positive logic signal. The solution here is to change the logic level of the **B** signal. Once we change the logic level of **B** from the original positive logic to negative logic, '0' will then be the active level of the signal; or to use our new terminology, the signal is active low. To sum this up in one statement, the output of the gate is asserted when the both the **A** and **B** inputs are asserted. It just so happens that the **B** input is asserted low as it enters the AND gate because its logic level was changed by the inverter in the circuit. Figure 22.4(b) shows the logic levels of the signal after it passes through the inverter written in with equivalent forms as presented in Figure 22.3.



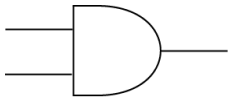
(a) (b)

Figure 22.4: A mixed logic approach to analyzing familiar functions.

OK… we've driven the inverter analysis into the ground; now let's go back and look at the logic gates we've dealt with up to this point: (AND, OR, NAND, NOR). Using a strange mixture of mixed logic concepts and Boolean algebra, we'll be generating some alternative forms of these gates. We'll be using these alternative forms later in our foray into mixed logic concepts. The theme of the following circuits is to take what we know about the gates we know and start to look at them differently and particularly in the concept of mixed logic.

The simplest approach to understanding mixed logic is to examine the most basic form of logic: the logic implemented with the basic gates used in digital design. Until now we've implemented our gate-

level designs using primarily AND, NAND, OR, NOR gates and inverters[264]. We've also implemented designs on the block diagram level but we won't be dealing with that in this section. Remember those bubbles on the outputs of the NAND and NOR gates (and inverters too)? They're actually significantly important, and if you understand their actual purpose, you'll be on your way to understanding mixed logic. The simplest digital device is the inverter, which is why we started the discussion there. The following figures describe mixed logic concepts at the gate level

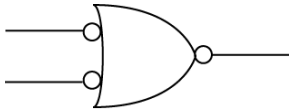


$$F = A \cdot B$$

$$F(H) = A(H) \cdot B(H)$$

This is the ever-so friendly AND gate. You've grown to love this gate as a device that has a high output when both of its inputs are high. This is all good and fine but now we want to re-examine it from another angle. What you see now is an AND gate; this is a device that provides an AND function with a positive logic output. As you can see from the equations shown on the left, the AND gate performs an AND function on the two positive logic inputs. Since there are no bubbles on the back of the gate (you'll see some bubbles in next gate described), this AND gate expects positive logic inputs. This will make more sense as we look at the next gate. To be consistent with the following diagrams, the gate on the left is the AND form of an AND gate.

Figure 22.5: A mixed logic view of an AND gate.



$$F = A \cdot B$$

$$\overline{\overline{F}} = \overline{\overline{A \cdot B}}$$

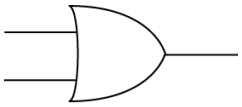
$$F = \overline{\overline{A} + \overline{B}}$$

$$F(L) = A(L) + B(L)$$

The gate shown on the left is also an AND gate. Why is it an AND gate? Because you can use DeMorgan's theorem to generate a different equation (though functionally equivalent) describing the gate; once you do this, you can alter the form of the gate based on the equations you derived. The new form of this gate is derived from double complementing the equation describing an AND function and DeMorganizing the resulting equation. The distinctive symbol results from the two equations on the left; these equations are listed in PLC and DPI forms. The key to understanding this gate is to interpret both the bubbles and the gate form. It sort of looks like an OR gate, doesn't it? The truth is that if you feed this gate two negative logic input, it performs an OR function on those inputs and generates a negative logic output. We use bubbles to indicate the negative logic inputs (as indicated by the (L) polarity indicators) and negative logic output of the final equation on the left. The reality here is that you can use an AND gate to perform an OR function. In other words, this is an AND gate that performs an OR function. This gate is officially know as the OR form of an AND gate.

Figure 22.6: A different mixed logic view of an AND gate

[264] In reality, an inverter is not really a logic gate; but it's a cool and useful device anyway.

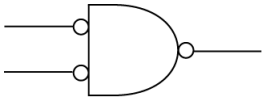


$$F = A + B$$

$$F(H) = A(H) + B(H)$$

This is the ever-so familiar OR gate. This gate provides a high output when either of the gates two inputs is at a high state. Another way of looking at this gate is if you provide it with positive logic inputs, it performs an OR function and generate a positive logic output. The equations on the left show this characteristic; we write the equations in both PLC and DPI form. Since there are no bubbles on the back of the gate, this OR gate expects positive logic inputs. Since there is not a bubble on the gate outputs, this gate delivers a positive logic output. In summary, the gate on the left is the OR form of an OR gate.

Figure 22.7: A mixed logic view of an OR gate.



$$F = A + B$$

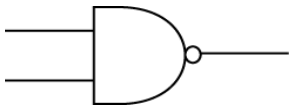
$$\overline{\overline{F}} = \overline{\overline{A + B}}$$

$$F = \overline{\overline{A} \cdot \overline{B}}$$

$$F(L) = A(L) \cdot B(L)$$

The gate shown on the left is also an OR gate. We derive this new gate form from double complementing the equation describing the OR function and DeMorganizing the resulting equation. We derive this distinctive symbol from the bottom two equations; we list these equations in both PLC and DPI forms. The key to understanding this gate is to interpret both the bubbles and the gate form. It sort of looks like an AND gate but the reality is this: if you feed this gate two negative logic inputs, it performs an AND function on those inputs and generates a negative logic output. We use bubbles on the resulting gate to indicate the negative logic inputs (as indicated by the (L) polarity indicators) and negative logic output of the final equation. Not surprisingly, you can use an OR gate to perform an AND function (as the equations show). This gate is officially know as the AND form of an OR gate. In other words, this is an OR gate that performs an AND function.

Figure 22.8: A different mixed logic view of OR gate.

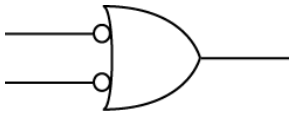


$$F = \overline{A \cdot B}$$

$$F(L) = A(H) \cdot B(H)$$

The gate shown on the left is a NAND gate. You may have come to know this gate as a AND gate with an inverted output; this definition is not too far from a mixed logic view of this gate. In a mixed logic sense, this gate performs an AND function on the two positive logic inputs and provides a negative logic output. The inputs are positive logic due to the absence of bubbles on the inputs; the output is a negative logic output since there is a bubble on the output. One way to look at this circuit is that the output of '0' is now the active state rather than the '1' output that is the active state from a normal AND gate. This gate is officially known as the AND form of a NAND gate; this gate performs an AND function on its positive logic inputs and returns a negative logic output.

Figure 22.9: A mixed logic view of an NAND gate.



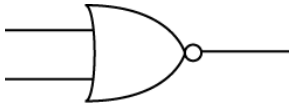
$$F = \overline{A \cdot B}$$

$$F = \overline{A} + \overline{B}$$

$$F(H) = A(L) + B(L)$$

The gate shown on the left is also a NAND gate. Note that if we apply DeMorgan's theorem to the gate we can arrive at a slightly different looking equation describing the gate. The final two equations on the left describe this gate in the context of mixed logic: this gate performs an OR function on its two negative logic inputs and returns a positive logic output. The fact that an OR function will be performed is indicated by the distinctive OR symbol; this gate only performs an OR function if the two input values are provided in negative logic format. The bubbles indicate the negative logic input format; the absence of a bubble on the output indicates positive logic. The polarity indicators in the final equation on the left show the logic level of this gate's inputs and output. This gate is officially known as the OR form of a NAND gate; this gate performs an OR function on its negative logic inputs and returns a positive logic result.

Figure 22.10: Yet another mixed logic view of an NAND gate.

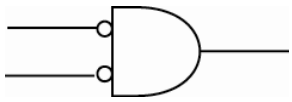


$$F = \overline{A + B}$$

$$F(L) = A(H) + B(H)$$

The gate shown on the left is a NOR gate. You're probably used to thinking of this gate as an OR gate with an inverted output. In a mixed logic context, this gates actually performs an OR function on its two positive logic inputs and outputs a negative logic result. The equation on the left nicely describe this gate's attributes. Note that absence of bubbles on the inputs indicate that the inputs are positive logic; the gate's output is a positive logic output due to the presence of the bubble on the output. In other words, if this gate receives two positive logic signals on the input, the gate performs an OR function and returns a negative logic result. This gate is officially known as the OR form of a NOR gate. In yet other words, the gate performs an OR function with an asserted low result.

Figure 22.11: A mixed logic view of an AND gate.



$$F = \overline{A + B}$$

$$F = \overline{A} \cdot \overline{B}$$

$$F(H) = A(L) \cdot B(L)$$

The gate shown on the left is also a NOR gate. Note that we can apply DeMorgan's theorem to the equation describing the NOR gates and arrive at a new and even more wonderful equation. The final two equations on the left describe the operation of this gate in the magical world of mixed logic. To state this magic directly: this gate performs an AND operation (note the AND symbol being used here) if we provide two negative logic signals as inputs; the resulting output of the AND operation is positive logic. The bubble placement provides another way of looking at this gate; since there are bubbles on the inputs, this gate only performs the AND operation if the two inputs are negative logic. Since the output contains no bubble, the output of the gate is a positive logic signal. This gate is officially known as the AND form of the NOR gate. In other words, this gate performs an AND function if it receives a negative logic input; it subsequently provides a positive logic output.

Figure 22.12: A mixed logic view of an AND gate.

Figure 22.13 shows a summary of all the standard gates forms. At this point, you may be wondering why there are some many different forms of gates out there. It rather seems like we were doing OK with

the few gates forms we knew about before breeching the topic of mixed logic. The short answer to this question is that in some situations, we need a certain amount of flexibility in implementing logic functions. We've done OK up to now, but there are some situations where a mixed logic approach is not avoidable. If you understand the basics of mixed logic, you will have nothing to fear.

The real reason we have so many different and sometimes equivalent gates is that we need to always choose the gate that most appropriately represents the actual logic function we are performing. This desire, however, becomes trickier in a mixed logic environment. In reality, there are still only AND and OR functions out there; we need to draw our circuits such that they express whether we are performing an AND function or an OR function. The relatively large set of gates guarantees that we'll be able to accurately display the actual logic functions we're performing in a mixed logic environment. Then again, if you don't use the proper gate in your design, you may have a working circuit but no one will know what the heck you're really doing[265].

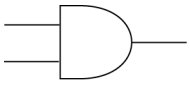
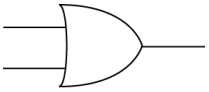
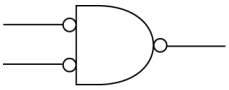
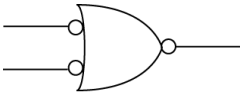
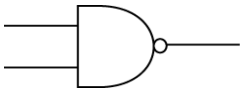
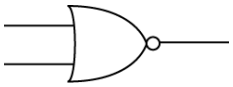
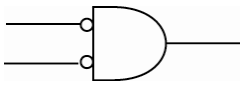
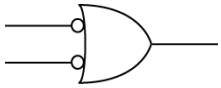
Standard Gate Forms	
AND functions	**OR functions**
	
AND form of **AND** gate	**OR** form of **OR** gate
	
OR form of **AND** gate	**AND** form of **OR** gate
	
AND form of **NAND** gate	**OR** form of **NOR** gate
	
AND form of **NOR** gate	**OR** form of **NAND** gate

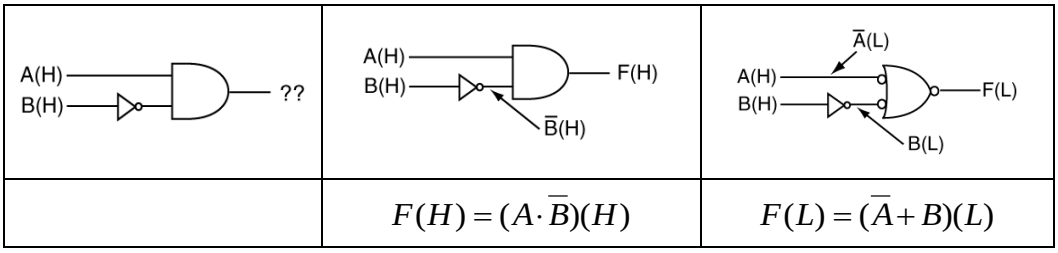
Figure 22.13: The giant summary of the strange new gate forms.

For an example in mixed logic analysis, look at Figure 22.14. For this example, you must analyze the circuit shown in Figure 22.14(a). The first thing you should note is that the both inputs in this circuit are positive logic. The second thing you should notice is that there is no indication of the output logic level. Because the diagram does not list the output logic level, you can assume it is either negative or positive logic[266]. For this example, we'll examine both cases.

[265] It's well known that such mystery designs establish job security of sorts for the circuit designer.
[266] Not listing the output logic level is a horrendously bad thing.

Figure 22.14(b) shows the case where the output is positive logic; the direct polarity indicator shows the logic level of the output. The important thing to note here is that the polarity indicator on the output of the gates matches what the gate states it is providing: since there is no bubble on the gate, we consider the output logic level as positive logic. This gate is an AND gate and performs an AND function on the two inputs if they are both provided as positive logic (note the absence of bubbles on the gate inputs).

The first thing we need to do is to write the inputs such that they indicate a positive logic signal as this is what the AND gate is expecting. The **A** input is in correct form already because it is a positive logic signal. The **B** signal, however, passes through an inverter before entering the AND gate. Although the inverter changes the logic level from positive to negative, the AND gate is still expecting a positive logic input. In other words, if you were to input a B(L) signal to the AND gate, it would not look correct would lead to mass confusion and hysteria. The solution to this dilemma is to use an equivalent signal representation for the B(L) signal. We can officially list this signal as a positive logic signal by listing the signal name in complemented form. Once both inputs are written in positive logic form, we can write the down the resulting equation (shown under the circuit in Figure 22.14(b)). The important thing to note about this equation is that the polarity indicators on both sides of the equation match. If they did not match, the equation would make no sense.



$F(H) = (A \cdot \overline{B})(H)$

$F(L) = (\overline{A} + B)(L)$

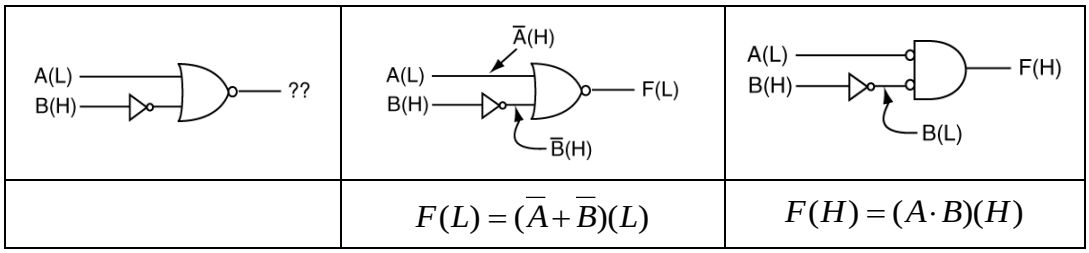
(a) (b) (c)

Figure 22.14: An example of mixed logic analysis where (a) shows the original circuit and (b) and (c) show positive and negative logic interpretations of the output, respectively.

OK, now let's re-analyze this circuit as having a negative logic output. In other words, we want to know what logic function is being executed if the output is interpreted as negative logic. The first step in doing this is to redraw the gate such that there is a bubble on the output. We need to replace the original AND gate with an equivalent gate; when we replace it with an equivalent gate, we are officially not changing the circuit in any way.

Figure 22.14(c) shows that the equivalent gate form for an AND gate is the OR form of an AND. Note that once we replace the gate and the bubble appears on the output, we rewrite the output of the gate to show that it is negative logic. The inputs to the new gate form need some modification also. The new gate form is going to perform an OR function when both of the two gate inputs are provided in negative logic. This requires that we rewrite the input logic levels in forms that reflect the negative logic levels. The **B** input is positive logic and the inverter changes it to negative logic; for this input, we do not need to perform any modifications. The **A** input is also positive logic but needs to be in negative logic as directed by the bubbled input to the gate. Since there is no inverter on this input, the approach we'll take is to rewrite the signal with an equivalent signal name as shown in Figure 22.14(c). The equivalent signal names contains a polarity indicator that indicates the gate receives a negative logic signal as requested. Figure 22.14(c) shows the resulting equation below the circuit diagram.

Figure 22.15 shows another example of mixed logic analysis. This example is slightly different in that the inputs are in a true mixed logic form: the **A** and **B** inputs are in negative and positive logic forms, respectively. Once again, there is a question of the output level of the circuit so this example will once again examine both cases of output logic levels.



(a) (b) (c)

Figure 22.15: An example of mixed logic analysis.

The circuit in Figure 22.15(a) has two inputs, one input both positive and negative logic. Since the output level of the device is not stated, let's analyze this problem assuming the output logic level is asserted high and then re-analyze it as asserted low. The circuit shown in Figure 22.15(b) assumes the output is asserted low, which is the implication from the original drawing of the gate (because of the bubbled output of the gates). In this case, the gate provides an OR function with an asserted low output under the conditions that the two inputs are positive logic. Since the **A** input is negative logic, we must re-write it in positive logic form in order for us to know what logic function the gate is performing; this is done by using an equivalent signal for the **A** input. The **B** input is in positive logic but has its logic level changed to negative logic by the inverter. Once again, we rewrite the negative logic signal for **B** in positive logic form using equivalent signals. Once the two inputs are both in positive logic forms, we satisfy the gate inputs and we can then write the equation for the circuit (shown under the circuit in Figure 22.15(b)).

But what if the output was actually a positive logic output? We first need to represent that condition with a gate that has no bubble on the output; we do this by using an equivalent gate form for the NOR gate as listed in Figure 22.15(a). In this case, the equivalent gate is the AND form of a NOR gate as shown in Figure 22.15(c). This gate performs an AND function with a positive logic output if the two inputs are in a negative logic format. The **A** input requires no modification because it is already in negative logic format. The **B** input is originally in positive logic format but the inverter changes the logic level to negative logic. Once the two inputs to the circuit are in negative logic forms, we can write the equation performed by the gate.

Note that in both of the two previous examples we were given the choice of how to interpret the output of the circuit. The analysis of the circuit entailed using equivalent gates and equivalent signals in order to discern the logic function performed by the gate. Here are a few key things to notice about this form of analysis.

- The output logic level always matched the gate output level. In other words, if there is a bubble on the output of the gate, the gate is providing a negative logic signal. If there is not bubble on the gate output, the gate is providing a positive logic signal.

- We only used the polarity indicators in the final equation for the output; we did not carry around the polarity indicators for the internal signals. The assumption made here is that we matched all the interior logic levels so there is no need to include them in the final equation.

- In the final equation, the polarity indicators match. If they did not match, the equation would not make sense; it would not really be an equation. It would be evil confusion.

- Although we only had one circuit, we seemed to have generated two equations from it. This is true because we base the two final equations for these circuits on our interpretation of the circuit's output. In other words, depending on how we interpreted the logic level of the

circuit output, we were able to consider the function as implementing two different functions. The reality is that the two equations have sort of a complementary relationship (think DeMorgan's theorem).

Figure 22.16 shows a slightly more complicated example that is similar to the previous example. Have no fear; the analysis approach is the same as the simple gate: match the logic levels to the gate inputs and outputs using equivalent gates and equivalent signals. The explanation of this example is also similar to the previous example, so we omit the bloviated description.

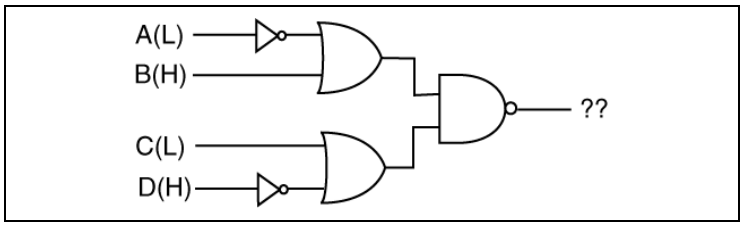
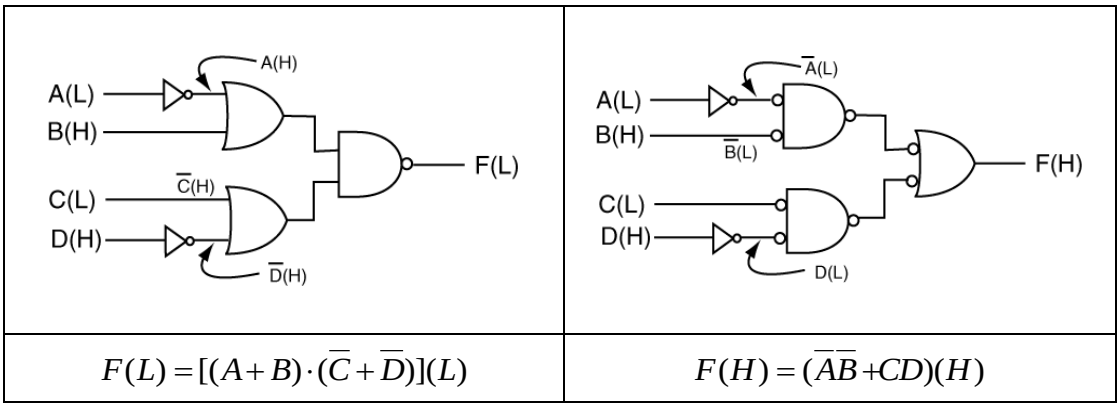


Figure 22.16: A slightly more complicated mixed logic example.



(a)	(b)
$F(L) = [(A+B)\cdot(\overline{C}+\overline{D})](L)$	$F(H) = (\overline{A}\overline{B}+CD)(H)$

Figure 22.17: The total mixed logic analysis approach.

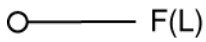
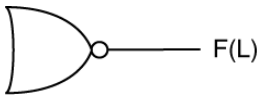
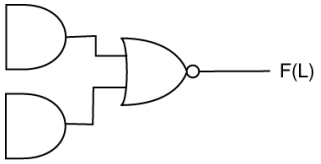
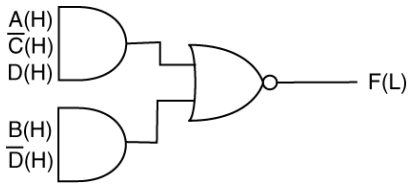
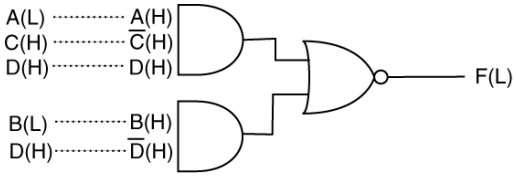
Up to now, we have been analyzing mixed logic circuits. Let's switch to the opposite approach and design some circuits based on mixed logic. The following examples provide such a design problem.

Example 22-1:

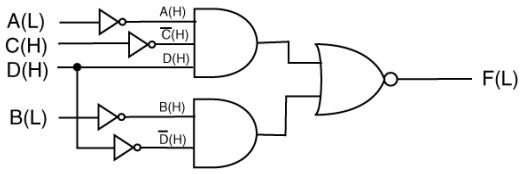
Design a circuit that implement the following function: $F(A,B,C,D) = A\overline{C}D+B\overline{D}$

For this problem consider the **A** and **B** inputs and the output as asserted low; all other inputs are positive logic. Implement this function using any type of gates.

Solution: The first thing to do with this solution is to list the parameters in DPI form. We represent the negative logic signals by A(L), B(L); we represent the **F** output as F(L). We represent the two positive logic signals by C(H) and D(H). The best approach to problems such as these is to start at the output and work backwards. The following verbage shows this step-by-step approach.

Step 1: Draw and label the output. Since we know the output is asserted low, draw a bubble and label it in such a way as it support the original problem description.	
Step 2: Draw a gate such that it satisfies the logic function requested by the equation given in the problem description. In this example, the required logic function is an OR function so we draw a gate that looks something like an OR gate. Since this example does not specify which type of gates to use, we'll use a NOR gate which provides an OR function with a negative logic output.	
Step 3: The other part of the equation includes two product terms. For this problem, the product terms can be implemented with some form of AND gates. Since this problem did not restrict the type of gates, choosing an AND form of an AND gate is sufficient. What makes an AND gate appropriate is that the input to the OR gate is expecting positive logic inputs (note the absence of bubbles on the input). The AND gates provide an AND function with positive logic outputs. The final word is that the bubbles (or lack there of in this case) match and every thing is happy up to now.	
Step 4: We're now ready to assign some logic for the inputs to the AND gates. We've written the logic that the AND gates are expecting based on the original equation provided by the problem. Note that since the AND gates are expecting positive logic inputs, we listed all of the inputs in positive logic form.	
Step 5: In this step, we've included the input signals with the logic levels as stated in the input problem. Recall that **A** and **B** are provided in negative logic form while **C** and **D** were positive logic. The dotted lines mean nothing in particular; we draw them to refer to what each AND gate needs relative to the original equation and the logic levels of the inputs from the outside world.	

Step 6: From the previous step, you can see that some of the input signals are not in the correct logic form as required by the AND gates. For these cases, an inverter is required in order to switch the logic levels. Note that in some cases, we have switched the logic levels directly and in other cases, we have rewritten the signals using equivalent signal forms.



One of the key elements in the previous problem is that we had the luxury of using any type of gate we could in the implementation. Let's redo the previous problem, but this time restrict our gate usage to NOR gates and inverters. As you'll see in the section on circuit forms, we usually need to implement functions using only one type of gate.

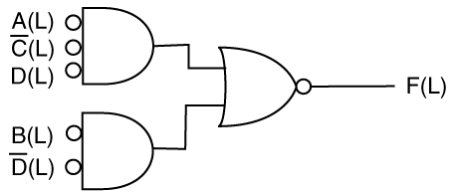
Example 22-2:

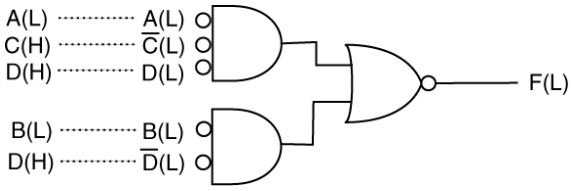
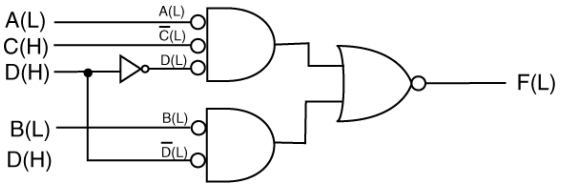
Design a circuit that implement the following function: $F(A,B,C,D) = A\overline{C}D + B\overline{D}$

For this problem consider the **A** and **B** inputs and the output as asserted; all other inputs are positive logic. Implement this function using only NOR gates and inverters.

Solution: We'll take a few short cuts in this problem since we already choose a NOR gate for the output stage of this circuit in the previous problem.

Step 4: For this problem, we jump in at Step 4 because the first three steps are the same as the previous problem. The difference in this problem is that we need to choose NOR gates for the input gates rather than the AND gates of the previous problem. The key here is that we need to choose a NOR gate that performs an AND function. Lucky for us, we can choose the AND form of an NOR gate. This gate performs an AND function if the inputs are provided in a negative logic format. The diagram shows the signal requirements as they relate to the original problem. Note that all of the polarity indicators on the signals have been listed as L as required by the gate inputs.



Step 5: This step is mostly a bookkeeping step. We need to make sure that we align the provided signals and their logic levels to the function we're implementing. We list the input requirements of the signal we're implementing on the inputs of the NOR gates.	
Step 6: The last step is matching the logic levels of the provided signals to those of the require function.	

There is something massively important to note in these previous two examples. Although we essentially did the same problem twice, we ended up with two different solutions. One of the good things about the second solution is that it uses fewer devices than the first solution. One of the distant morals of this story is that you can tweak the gates around and end up with many different equivalent solutions but some of the solutions can be implemented at a lower cost than others[267]. This is somewhat cool. Well, at least some people think it's cool.

Example 22-3: Generic Switch Controller

Design a circuit that controls an unspecified output according to the following description. If the MASTER_OVERRIDE switch is asserted, the output is always asserted. Otherwise, if the LOCAL_OVERRIDE switch is asserted, the output is also asserted. If both the override switches are not asserted, the output is only asserted when SW1 and SW2 are both asserted. For this problem, consider the output to be active low. The two override switches are active low also; SW1 and SW2 are active high. Specify the solution using POS form.

Solution: This problem is similar to other switch problems you've done but with the twist added of working with both negative and positive logic. Since there are not too many inputs, you can take the truth table approach to designing this problem. Figure 22.18 shows the empty truth table.

[267] It another version of the minimum cost concept idea as previously discussed

MO	LO	S1	S2	F
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Figure 22.18: Truth table for design example.

The problem with this problem is that we need to deal with mixed logic. Although there are many approaches to dealing with mixed logic, the approach we'll take here is somewhat more straight-forward than other approaches. Since we're more used to dealing with positive logic, let's convert the negative logic signals to positive logic before we assign the output values. We'll also convert the negative logic output to positive logic. Once we've specified the output, we'll complement it before we generate the subsequent logic. We'll not have to do anything with the inputs at that point since the inputs still reflect the order (but not the numbering) required to use a truth table.

!MO	!LO	S1	S2	!F
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1

(a)

!MO	!LO	S1	S2	F
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0

(b)

Figure 22.19: The modified truth tables with negative (a) and positive (b) logic outputs.

The final logic we're looking for is specified in **Error! Reference source not found.**(b). Once you toss the column for F into a truth table, you'll arrive at the POS equation shown in **Error! Reference source not found.**.

$$F = (MO \cdot LO)(\overline{S1} + \overline{S2})$$

Figure 22.20: The final equation for this problem.

Chapter Summary

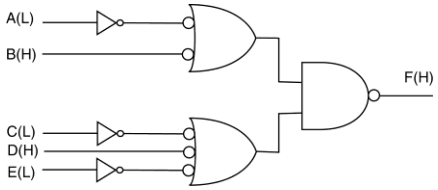
- The concept of mixed logic is based upon the "action" state of a digital signal. In all cases, either the '1' or '0' state is considered to be the action state if a digital signal. If a '1' is considered the action state, the design is considered to be positive logic while if the '0' is the action state, then the design is considered to be negative logic. A mixed logic system is a digital system that uses both negative a positive logic in the design.

- Most gate-level circuits deal with mixed-logic concepts at some level. Although mixed logic concepts are often initially confusing the digital designers, having a basic understanding of the mixed logic is generally enough for survival in digital design land.

- Logic levels in digital circuits are represented by either the Positive Logic Convention (PLC) or Direct Polarity Indicators (DPI). Logic levels in a circuit are often referred to as assertation levels.

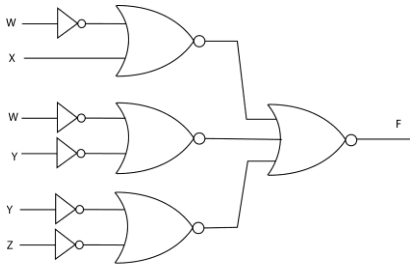
Chapter Exercises

1) Write an equation for **F(H)** that describes the following circuit. Put your answer in DPI form.

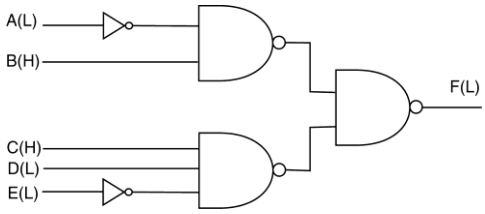


2) Implement the following equation using any type of gate and inverters. Minimize the device count in your implementation. $F(L) = [(\overline{A} + B + C)(\overline{D} + E)](L)$. Consider the inputs and outputs to be: A(H), B(L), C(H), D(H), E(L), F(L).

3) Write an equation for **F(W,X,Y,Z)** in NAND/NAND form.

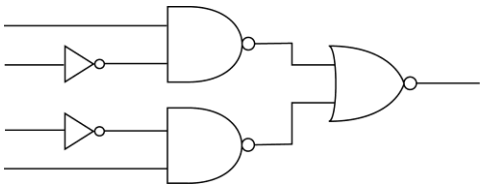


4) Write an equation for F(L) that describes the following circuit using DPI.



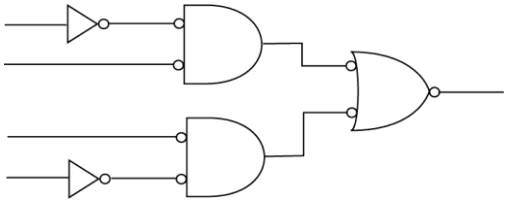
5) Draw a circuit that implements the following function: $F(H) = [(A + B + \overline{D})(\overline{A} + \overline{C})](H)$. Consider the inputs to be: A(L), B(L), C(H), D(L); use only standard gates and inverters in your solution.

6) Without altering the function implemented by the circuit below, redraw the circuit using only **OR** gates and inverters. Minimize device count where possible.



7) Using only NAND gates and inverters, draw a circuit that implements $F(H) = (AB + \overline{B}CD)(H)$. Consider the inputs and outputs to be: A(L), B(L), C(L), D(H), F(H).

8) Without altering the function implemented by the circuit below, redraw the circuit using only **NAND** gates and inverters. Minimize device count where possible.



9) Using only NOR gates and inverters, draw a circuit that implements $F(L) = (\overline{A}B + \overline{B}CD)(L)$. Consider the inputs and outputs to be: A(L), B(H), C(L), D(H), F(L).

10) Draw a circuit that implements the following function: $F(L) = (A\overline{B}D + BC)(L)$. Consider the inputs to be: A(H), B(L), C(L), D(L); use only standard gates and inverters in your solution.

Design Problems

1) A logic network is to be designed to implement a seat belt alarm that is required on all new cars. A set of senor switches is available to supply the inputs to the network. One switch will be turned on if the gear shift is engaged (not in neutral). A switch is placed under each front seat and each will turn of if someone sits in the corresponding sear. Finally, a switch is attached to each front seat which will turn on if and only if the seat below is fastened. An alarm buzzer is to sound (LED display light) when the ignition is turned on and the gear shift is engaged, provided that either of the front seats is occupied and the corresponding seat belt is not fastened.

Alarm (sound) - A(H)
Ignition (on) – I(L)
Gearshift (engages) – G(L)
Left Front Seat (occupied) – LFS(H)
Right Front Seat (occupied) – RFS(H)
Left Seat Belt (fastened) – SBL(H)
Right Seat Belt (fastened) – SBR(H)

2) There are four parking slots in the Acme Inc. executive parking area. Each slot is equipped with a special sensor whose output is active low when a car is occupying the slot. Otherwise, the sensor's output is at a high voltage. You are to design and draw schematics for a decoding system that will generate a low output voltage if and only if there are two (or more) adjacent vacant slots.

23 Chapter Twenty-Three

(Bryan Mealy 2012 ©)

23.1 Chapter Overview

Back in the dark ages, the K-map was the primary method used for reducing Boolean equations. Although the practical use of K-maps is limited by the number of input variables in the function, the use of map entered variables (MEVs) could partially bypass this limitation under certain conditions. The use of map entered variables does have some interesting uses and they are still commonly used to describe the operation of many digital integrated circuits. While map entered variables are not too to hard figure out by staring at them, an appropriate background is rather nice. This chapter provides that background.

Main Chapter Topics

- **MAPPED ENTERED VARIABLES:** This chapter also introduces the notion of map entered variables. Although some of the topics associated with map entered variables are obsolesced, they do form the underlying theme for several other subtopics discussed in this chapter including K-map compression and implementing functions using K-maps.
- **MUX-BASED FUNCTION IMPLEMENTATION:** Boolean functions can be implemented in many different ways and with many different devices. A MUX can also be used to implement functions; the method used to implement these functions is somewhat related to the notion of mapped entered variables.

Why This Chapter is Important

This chapter is important because it provides the theoretical foundation for using and understanding map entered variables.

23.2 Map Entered Variables

Without doubt, *map entered variables (MEVs)* used to be an important topic in digital design. Back in the days when a digital design course was mostly paper designs, MEVs were more important because they provided a topic that was easy to test students on. In current digital design courses where students are actually implementing real circuits (using VHDL modeling), the topic of MEVs is less applicable. But it's important enough to require that you have a few MEV skills. Also, it builds your familiarity and

skills with K-maps. In summary, this is an important topic; you'll realize this one day if you continue onwards into digital design.

MEVs are, as the name implies, variables that are entered directly into a K-map. Up until now, we have only entered 1's and 0's into K-maps. One of the major drawbacks of a K-map is the fact that once your function in question has more than four variables, your K-map becomes real ugly and you start regretting ever studying digital design. Using MEVs somewhat mitigates this problem. This is of course not a great justification because in reality, a computer with the proper software will do a better job of reducing functions than you and your K-maps.

To develop these ideas, consider a three variable K-map with input variables **A**, **B**, and **C**. A general expression for such a K-map in standard SOP form can be written as shown in Equation (a) of Figure 23.1. If we AND both sides of the function with the Boolean variable F, no algebra rules are violated and we preserve the inequality. Figure 23.1 shows the resulting equation. From the Equation in Figure 23.1(b), we can now substitute in the actual values of a given function for the variables F that appear as part of the product terms on the left-hand side of the equation. For this example, we'll use the function: $F = \sum(2,3,5,6)$. The result of these substitutions appears in Equation (c) of Figure 23.1.

From this point, we can massage the resulting equation to the new look of Equation Figure 23.1(d). Note that in Equation (d) every value of F is represented but we've factored out the A and B terms. Using this notation, the AB-type terms are referred to as sub-minterms and include every possible combination of the two most significant variables from the associated truth table. Equations Figure 23.1(e) and Figure 23.1(f) are derived by grinding out the math associated with the non-sub-minterm expression. The final equation presented in Figure 23.1(f) presents a combination of both the sub-minterms and the MEV terms (the MEV terms are shown in the square brackets).

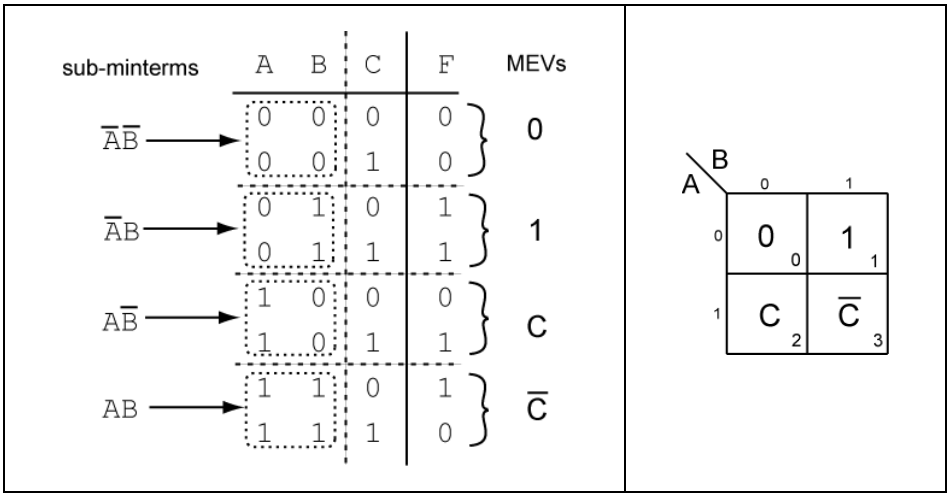
(a)	$F = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C} + ABC$
(b)	$F \cdot F = F = \overline{A}\overline{B}\overline{C}F + \overline{A}\overline{B}CF + \overline{A}B\overline{C}F + \overline{A}BCF + A\overline{B}\overline{C}F + A\overline{B}CF + AB\overline{C}F + ABCF$
(c)	$F = \overline{A}\overline{B}\overline{C}(0) + \overline{A}\overline{B}C(0) + \overline{A}B\overline{C}(1) + \overline{A}BCF(1) + A\overline{B}\overline{C}(0) + A\overline{B}C(1) + AB\overline{C}(1) + ABC(0)$
(d)	$F = \overline{A}\overline{B} \cdot [\overline{C}(0) + C(0)] + \overline{A}B \cdot [\overline{C}(1) + C(1)] + A\overline{B}[C(0) + C(1)] + AB[\overline{C}(1) + C(0)]$
(e)	$F = \overline{A}\overline{B} \cdot [0 + 0] + \overline{A}B \cdot [\overline{C} + C] + A\overline{B}[0 + C] + AB[\overline{C} + 0]$
(f)	$F = \overline{A}\overline{B} \cdot [0] + \overline{A}B \cdot [1] + A\overline{B}[C] + AB[\overline{C}]$

Figure 23.1: The derivation of the sub-minterm and MEVs.

The results from the derivation shown in Figure 23.1 are typically not the approach we use to generate MEVs. Figure 23.2 shows a better way to generate MEVs, although this is still not the best way. Note that Figure 23.2 was originally the truth table associated with the function in the previous derivation. This truth table is then divided into two-row pairs; each of the two-row pairs in the truth table is associated with a sub-minterm. Note that for each of the two-row pairs, the **A** and **B** values are identical. Be sure to note that these values are associated with a 2-bit binary count. The column labeled "MEV" shows the MEVs associated with this example.

The example of Figure 23.2 was carefully construction to show each of the four possibilities for the MEVs. The approach we're taking in this truth table is to make the **C** variable into an MEV. This requires that we draw upon the relationship between the **C** and **F** columns in the truth table. In this manner, there are only four possible MEVs as shown in the MEV column of Figure 23.2: "0", "1", "C",

and a complemented "C". From this point, it should be clear that we can enter the MEVs into a two-variable K-map. Figure 23.2 (b) shows the resulting K-map.



(a) (b)

Figure 23.2: A truth table shown MEVs and sub-minterms (a) and the associated 2-variable truth table.

The technique of generating MEVs from a truth table as shown in Figure 23.2 works fine for the least significant variable in the truth table. We can use a better technique to convert an input variable into a MEV. There is a one interesting thing about the K-map shown in Figure 23.2(b) that is worth mentioning here. Although this K-map appears the same as any other 2-variable K-map that you've worked with, it is actually significantly different.

With a normal K-map, each cell represents one output value. Each cell in the K-map of Figure 23.2(b) now represents two output values as you can see from Figure 23.2(a). Another way to look at this is that every cell in the K-map of Figure 23.2(b) is now a miniature K-map all its own; each of the mini-K-maps that are represented by these cells are one-variable K-maps. While we have never had the need to work with a one-variable K-map, such a K-map would contain two cells. In other words, the K-map of Figure 23.2(b) is actually comprised of four two-variable K-maps. This is somewhat strange but rather cool at the same time.

23.2.1 Karnaugh Map Compression

The term K-map compression comes from the fact that when generating MEVs, the resulting K-map shrinks by at least one variable. Shrinking the K-map is one of the great advantages of using K-maps in that you suddenly have the ability to reduce a six-variable K-map down to a manageable size (such as a four-variable K-map). You saw this compression in the example in Figure 23.2. What we want to do now is present a technique that we can use to compress a K-map for any variable instead of just the least significant variable as was done in the Figure 23.2 example.

For this example, let's compress the function shown in Figure 23.3. There is nothing special about this function; it's just another generic function used for the sake of this example. The approach we'll take in

this example is to compress this function for each of the input variables. Keep in mind that although we present this technique for a 3-variabl K-map, it works great for K-maps of any reasonable size[268].

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 23.3: A generic function we'll use to describe the K-map compressing technique.

The heart of this technique is to use the K-map to identify the sub-minterms. Once we identify the sub-minterms, they can be easily placed into a K-map. The one thing to keep in mind when using this technique is that there are only the four possibilities presented in the example of Figure 23.2: '0', '1', the MEV, and a complemented MEV.

Example 23-1

Compress the K-map of Figure 23.3 for the **C** variable.

Solution: The first step in the solution is to identify the sub-minterms. Figure 23.4(a) shows these sub-minterms using dotted lines. For the **C** variable, the sub-minterms are associated with the **A** and **B** variables. Another way to look at this is that the dotted lines are locating all the product terms associated with standard K-map groupings of the sub-minterm variables. This means that in Figure 23.4(a), the dotted lines represent the four possible AB-type K-map grouping. Yet another way to view this approach is identify groupings that cut the boundaries of the variable you're compressing. In Figure 23.4(a), the dotted lines include the decimal equivalent of the binary number associated with the sub-minterm variables. These decimal numbers function as indexes into the compressed K-map. Figure 23.4(b) shows the compressed K-map with MEVs entered.

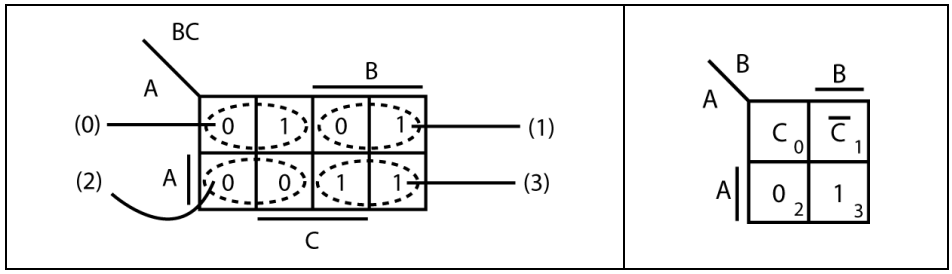


Figure 23.4: A truth table with sub-minterms (a) and the associated 2-variable truth table.

[268] We still fight hard to avoid working with K-maps of more than four variables.

The important thing to keep in mind in this problem is that each of the sub-minterms in Figure 23.4(a) generates a Boolean equation. Table 23.1 shows these four equations. Once again, these equations represent the four possibilities. Keep in mind that the outputs associated with both the complemented and uncomplimented **C** terms comes from the K-map of Figure 23.4(a). The truth is that you may want to generate these equations for the first couple of times you compress a K-map. After that, you'll probably develop your own technique for employing this approach that is more direct than writing out the equations for every sub-minterm.

Additionally, this technique works for the notion of compressing K-maps for more than one variable at the same time. You may actually want to do this someday; it used to be popular, but you don't see it much anymore. Keep in mind that if you're compressing a K-map for two variables, you'll be generating two-variable K-maps for each sub-minterm. It's fun stuff, sort of.

Sub minterm #	Sub minterm	Sub-minterm Equation	Final Term
0	$\overline{A}\,\overline{B}$	$\overline{C}\cdot 0 + C\cdot 1$	C
1	$\overline{A}B$	$\overline{C}\cdot 1 + C\cdot 0$	$\overline{C}$
2	$\overline{A}B$	$\overline{C}\cdot 0 + C\cdot 0$	0
3	AB	$\overline{C}\cdot 1 + C\cdot 1$	1

Table 23.1: Boolean equation explanation of Figure 23.4**.**

Example 23-2

Compress the K-map of **Figure 23.3** for the B variable.

Solution: The first step is to identify the sub-minterms. In this example, the sub-minters are associated with the A and C input variables. Figure 23.5 shows the associated K-map-type groupings that are associated with these sub-minterm variables. The transferring of the MEVs from the sub-minterm grouping to the compressed K-map is slightly more complicated in this example than it was for compressing the K-map for the C variable in the previous example, but still doable. Since some of the sub-minterm groupings are broken off the K-map in Figure 23.5, confusion may set in; try hard to prevent this. We can also generate Boolean equations similar to those of Table 23.1 for this example.

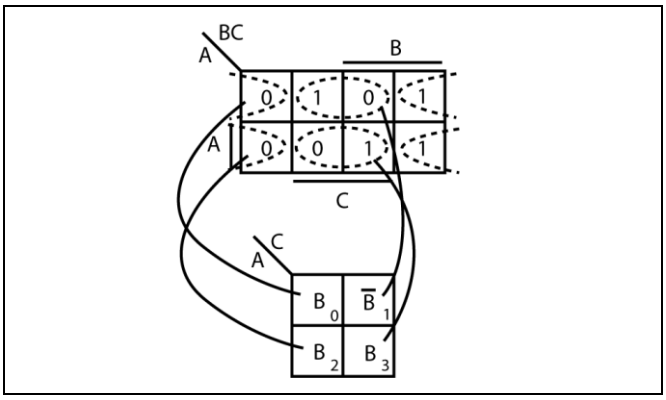


Figure 23.5: K-map compression for the B variable of a 3-input K-map.

Example 23-3

Compress the K-map of **Figure 23.3** for the **A** variable.

Solution: This example is once again similar to the previous two examples. Once again, the key to performing this type of compression is the use of the sub-minterms indexes to generate positional information into the compressed K-map. Figure 23.6 shows two flavors of K-maps for the solution of this problem. The overall approach here is to do what you need to do to compress the K-map. There are many ways to approach this problem; choose the way that makes the most sense to you.

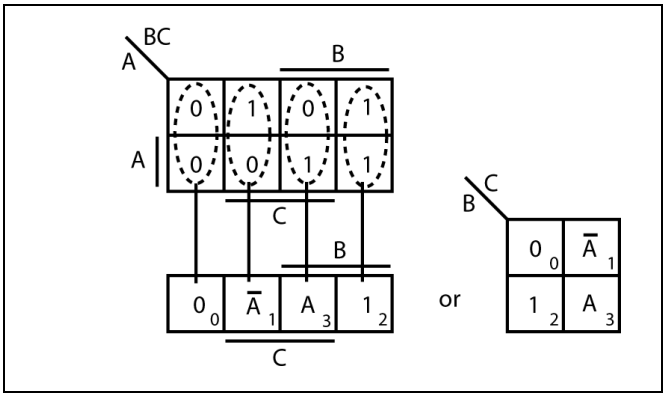


Figure 23.6: A truth table shown MEVs and sub-minterms (a) and the associated 2-variable truth table.

You can use this K-map compressing technique to compress four-variable K-maps also. There are some exercise problems that do this but no examples are provided here. Another thing worth noting is that you can compress a K-map that was previously compressed by also applying this technique. In this case, each of the K-map cells represents more than two non-compressed K-map cells. This is not a complicated topic but it does not have much use in modern digital design.

23.3 Implementing Functions Using MUXes

Implementing functions using actual digital devices has been the name of the game for many years now. Back in the old days, before there were programmable logic devices that could do just about anything, people implemented functions using discrete logic. If you ever have had the opportunity to rip apart old electronic devices, you'll generally come across boards that contained an exceeding number of integrated circuits, or ICs. These ICs were generally digital devices of various sorts; and often times they were simple devices such as discrete logic gates. The truth was that this flavor of design was tedious and time consuming and the resulting circuits ate up a lot of power. A partial solution for this problem was to implement functions using devices such as MUXes. Since some of the required logic was already built into the MUX (recall the underlying circuit diagram for a MUX), these circuit implementations required less external circuitry.

However, things have changed these days. Programmable logic devices are massively powerful and can easily implement even the most complicated digital devices. The need to implement functions on devices such as MUXes is obsolete. We'll take a brief look at it here because the topic does arise occasionally out there in digital design land, particularly during a job interview with dinosaur interviewers. It does have some relation to the compressed K-map, which provides an opportunity to practice those types of problems.

A MUX is a selection device; under the control of the selector inputs, one of the several inputs appears on the single output. The approach for having a MUX implement a function is to have the function's independent variables connected to the selector inputs of the MUX and to have the output variable connected to the other inputs of the MUX. Connecting the MUX in this way allows the selector variables to select one of the inputs (which are associated with the function's output) to appear on the output of the MUX. The best way to see this is through a simple example. We'll borrow the example we were working with in the section on compressed K-maps.

Example 23-4

Implement the following function on a 8:1 MUX: $F(A,B,C) = \sum(2,3,5,6)$

Solution: The compact minterm form of the function indicates where the 1's of the circuit live. For this problem, we simple need to connect power to the 2, 3, 5, and 6 inputs on the 8:1 MUX. Inputs that are not connected to power are the 0's of the circuit and are connected to ground. Figure 23.7 shows the final circuit. The triangular symbol on the bottom is a ground signal which is generally taken to be the '0' value in digital circuit land. The bent-T is the power connection and in this case is labeled "Vcc" for mostly historical reasons.

The way this circuit works is that the independent variables (A, B, & C) are used to control which of the MUX's eight data inputs will appear on the MUX's single output. For each of the minterms that "implicate" the function (meaning the rows in the associated truth table that contain a '1'), the MUX input with the same numeric index as the truth table row is connected to a '1'; all the other MUX data inputs are connected to '0'. The thought here is that I can now use one digital device, namely a MUX, to implement this function as opposed to a bunch of logic equations and all that stuff. They really used to do this in real life; I'm not kidding.

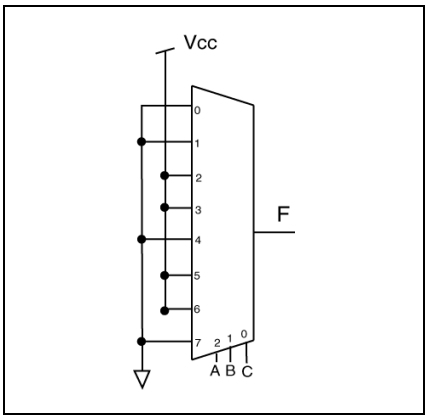
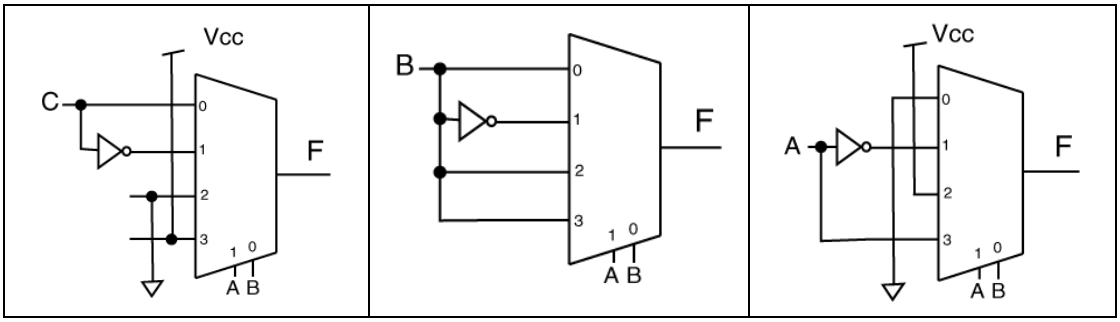


Figure 23.7: A MUX-based implementation of a function.

Example 23-5

Implement the following function on a 4:1 MUX: $F(A,B,C) = \sum(2,3,5,6)$. Provide three different solutions to this problem by using AB, AC, and BC and the selector inputs to the 4:1 MUX.

Solution: The trick to this flavor of problem is to realize that there are three independent variables and only two control variables on the 4:1 MUX. The way around this is to compress the K-map before implementing it on the MUX: the compressed variable then appears as a MEV on the inputs to the MUX. Herein lays the connection between K-map compression and implementing functions on MUXes. Instead of actually redoing this problem, we'll pull the solutions directly from Figure 23.4, Figure 23.5, and Figure 23.6. Figure 23.8 show the final MUX-based function implementations for this problem.



(a) (b) (c)

Figure 23.8: A MUX-based implementations for three different sets of control variables as specified in Example 23-5.

OK, this stuff is somewhat interesting. This is just a brief view of the material. In reality, it is possible to configure MUXes in this way in order to implement functions of any number of variables. One notion that we don't cover in this text is the fact that you can create a 256:1 MUX using smaller

MUXes (such as 4:1 and 8:1 MUXes). If there ever was a textbook definition of academic exercises, this use of MUXes could actually be it[269].

Finally, this stuff can be useful in special situations. The notion you may be getting from reading this text is that functions are similar to each other in the sense that the independent variables show up relatively often in the final expression. This is not always true. In some cases, you may have a function of ten variables, which sounds rather complicated. In reality, six of those variables may appear only once in the associated truth table. In this case, it is relatively easy to represent the function with a 4-input truth table and/or K-map with the other six variables entered as MEVs. It happens, though, we'll not go into it in this chapter.

[269] This may not be 100% true. Because PLDs often implement functions via look-up tables, there is a strong possibility that the writers of PLD design tools may be interested in this knowledge. From a digital designer's standpoint, you'll probably never even hear mention of the topic ever again.

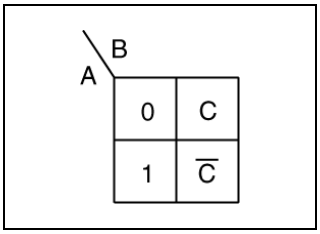
Chapter Summary

- Map entered variables, or MEVs, provide another method of representing functions and describing the operation of digital hardware. A MEV is a variable that associated with more than one cell in a K-map, or equivalently, more than one row in a standard truth table. One main use of MEVs is to effectively reduce the number of independent variables in for a given function which increases the possibility that a K-map reduction can be applied.

- MUXes can be used to implement functions. In these cases, the independent variables act as the control inputs to the MUX while the MUX output represents the function output. K-map compression facilitates the use of MUXes to implement functions. In real digital circuits, functions are rarely implemented on MUXes but the topic serves as yet another aid to understanding K-maps.

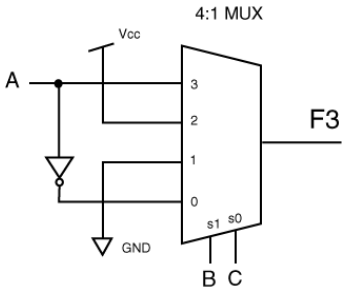
Chapter Exercises

1) Compress the K-map on the left for variable **B.**

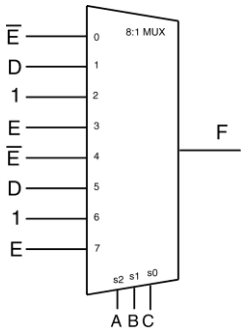


2) Compress variable **B** and draw the corresponding Karnaugh map for the following function:
$F(A,B,C) = \sum(1,2,4,5)$.

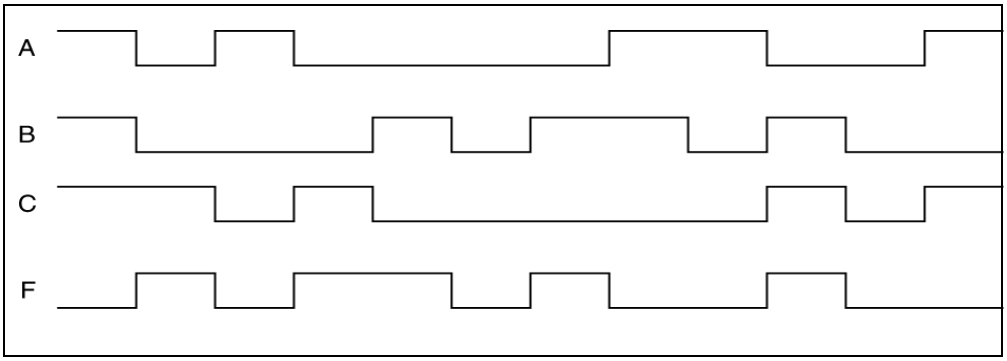
3) Draw a circuit that implements the following function using only NAND gates and inverters.



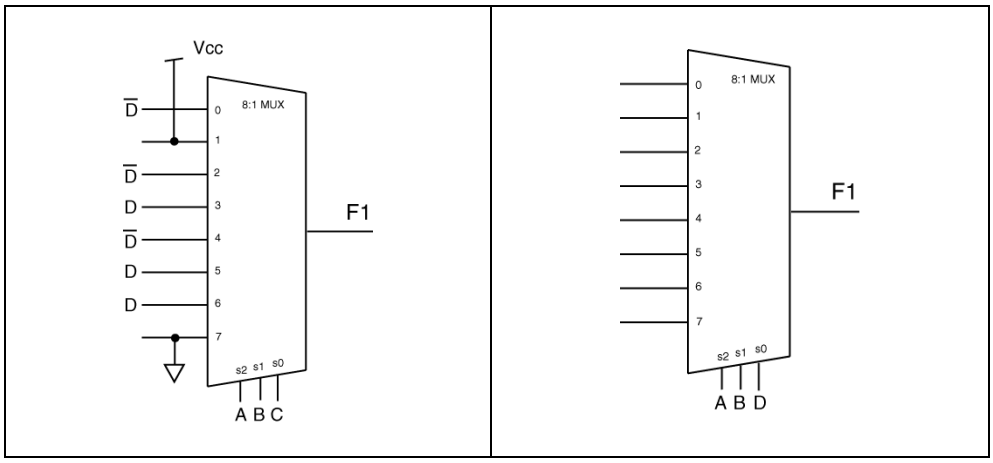
4) Write a reduced equation in NAND/NAND form for the following circuit.



5) The following timing diagram completely defines a function F(A,B,C) that has been implemented on an 8:1 MUX. The control variables are A, B, and C (A is the most significant bit and C is the least significant bit) and the output is F. Write an expression for this function in reduced NAND/NAND form. Assume propagation delays are negligible.



6) The diagram on the lefts shows an implementation of a function F1(A,B,C,D) on an 8:1 MUX using control variables of A, B, and C. Re-implement the F1(A,B,C,D) on the right MUX using control variables A, B, and C.



7) Compress the K-maps represented by each of the following four functions. Compress each of the four variables for each K-map.

a) $F = (A,B,C) = \sum(1,2,4,5)$

b) $F = (A,B,C) = \prod(0,2,5,6,7)$

c) $F = (A,B,C,D) = \sum(0,2,6,8,9,15)$

d) $\bar{F} = (A,B,C,D) = \prod(5,6,8,9,10,11,15)$

8) Implement $F(A,B,C,D) = \sum(1,3,4,5,7,10,11,12,15)$ on a 8:1 MUX with control variables of:

a) ABC
b) ACD
c) BCD
d) ABD.

9) Implement $F(A,B,C,D) = \sum(0,1,7,9,11,14)$ on a 4:1 MUX with control variables of:

a) AB
b) BD

25 Chapter Twenty-Five

(Bryan Mealy 2012 ©)

25.1 Chapter Overview

The previous chapter introduced the concept of state in digital circuits. The concept of "state" is important when dealing with sequential circuits since the notion of state is refers to the bits a given circuit is remembering. The previous chapter's introduction to digital circuits was rather basic and not overly useful in the real world, as you rarely see latches in modern digital design.

This chapter presents the notion of flip-flops, which are nothing more than latches with an added sense of control. The notion of flip-flops is somewhat "dated" as modern digital design no longer uses all flavors of flip-flops. However, the few flavors of flip-flops out there are arguably both interesting and instructive so we happily delve into them in this chapter. Included in this said delving is the inclusion of timing diagrams, which are probably more helpful to sequential circuits than they are to combinatorial circuits. Finally, continuing on our path towards usefulness is the notion of using VHDL to model memory. While not overly intuitive, the concept of inducing digital memory elements using VHDL is not overly complicated.

Main Chapter Topics

- **FLIP-FLOPS:** This circuit describes the three basic types of edge-sensitive latches, referred to as flip-flops.
- **MEMORY REPRESENTATION USING VHDL:** This chapter introduces and concept of modeling storage elements in using VHDL. Modeling memory in VHDL is a unique but a straightforward process.
- **MODELING SYNCHRONOUS AND ASYNCHRONOUS SEQUENTIAL CIRCUIT INPUTS:** This chapter introduces the notion of modeling various flavors of inputs in the context of modeling flip-flops using VHDL.

Why This Chapter is Important

This chapter is important because it describes the methods used to generate sequential circuits using VHDL models.

25.2 Flip-Flops

While the latches we've been working with are inherently level-sensitive devices, a flip-flop is essentially an *edge-sensitive*. Most of what you learned about latches transfers easily to flip-flops as

flips-flops are nothing more than edge-sensitive latches. This means that the outputs of the flip-flop can only change on an *active-edge*; so when you use the term "flip-flop" everyone knows it's an edge-sensitive device and not a level sensitive device.

When we derived the circuitry for the gated latch, we did so using an input referred to as the **C** input. The reality is that in most sequential devices, there is an input that is commonly referred to as a clock input. The edge-sensitivity in flip-flops is based on a clock edge. Some type of clock signal is thus an input to all flip-flops, or at least the standard flip-flops that we'll be dealing with.

Most digital logic texts include the derivation of the actual circuitry that creates the "edge sensitivity", so take a look if you're totally bored. The circuitry that achieves the edge triggering is somewhat complicated but is something we can skip over[270] without too much worry, as we prefer to keep many things abstracted to higher levels. The term edge-sensitivity in context of a clock signal means that the output can only change on a *rising edge* or *falling edge* of the signal. Let's now move into the land of flip-flops.

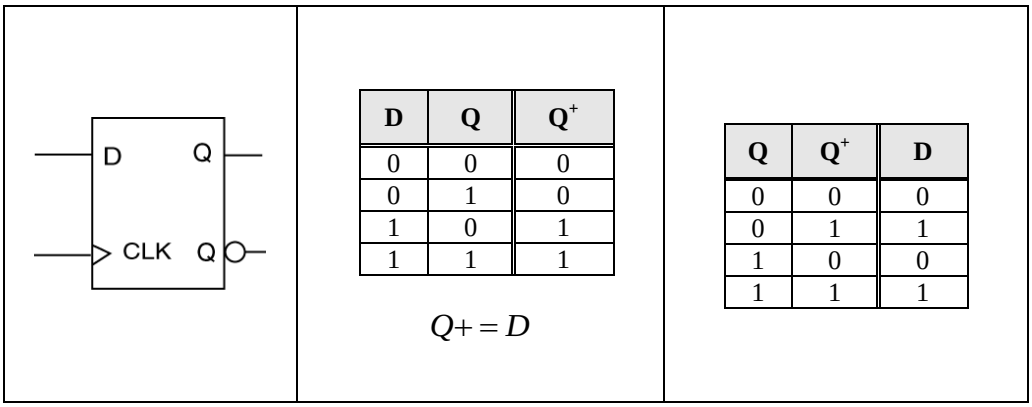
There are three main types of flip-flops out there in digital-land: the D, T, and JK flip-flops. We'll provide some short definitions and derivations of each of these flip-flops in this section. It's somewhat instructive to see where these devices come from although you may never need to go there again.

25.2.1 The D Flip-Flop

The D flip-flop is probably the most commonly used flip-flop. The D stands for *Data*, so this is a data flip-flop. The characteristic of a D flip-flop is that the output of the flip-flop follows the D inputs. Figure 25.1(a) shows a schematic symbol for a simple D flip-flop. The new and possibly shocking thing to notice about this symbol is the triangular shape on the CLK inputs. This triangle means that the device is edge-triggered. More importantly, since there is no bubble attached to this triangle, the device is a *rising-edge-triggered* (RET) D flip-flop. Had there been a bubble on the CLK input, this device would have been a *falling-edge-triggered* (FET) device.

Figure 25.1(b) shows the characteristic table of the D flip-flop. What the characteristic table shows is that the next state ($\mathbf{Q}^+$) of the flip-flop follows the D input to the flip-flop. By inspection of the characteristic table, you can generate the characteristic equation shown in Figure 25.1(b). Figure 25.1(c) shows the excitation table for the D flip-flop. From this table, you can see what the value of the D input needs to be in order to force the listed state change to occur. Since this is a D flip-flop, the output follows the D input.

[270] We're not skipping it… we're abstracting the concept to a higher level.



D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

$$Q+ = D$$

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

(a) (b) (c)

Figure 25.1: The schematic symbol (a), characteristic table and characteristic equation (b), and excitation table for the D flip-flop.

Figure 25.2 shows a timing diagram that demonstrates the operation of the RET D flip-flop shown in Figure 25.1(a). The only times the outputs of this device can possibly change are on the rising edge of the clock signal. Figure 25.2 uses dotted lines to show the rising edges of the clock across each listed signal. The timing diagram must provide the initial state of the D flip-flop (**Q**) otherwise you would not know the initial state of the device.

For the case of Figure 25.2, the initial state of the flip-flop is '0'. Since the D flip-flop is a sequential circuit, the timing diagram must provide the initial value of the output. At the first rising edge, the D input is a '1' and the value of this input transfers to the output and becomes the official "state" of the flip-flop. At the second rising edge, the D input is high once again so no state change in the flip-flop occurs.

Note in Figure 25.2 that during the time interval between the first and second rising edges, the D input changes twice. Changes such as these are ignored because the output can only change on the active edge (rising-edge) of the clock. At the third rising edge, the D input is in a low state, which causes the output of the flip-flop to change from high to low. At the fourth rising clock edge, the output is low again and the flip-flop remains in a low state. At the fifth clock edge, the D input is high which in-turn causes the state of the flip-flop to change from low to high.

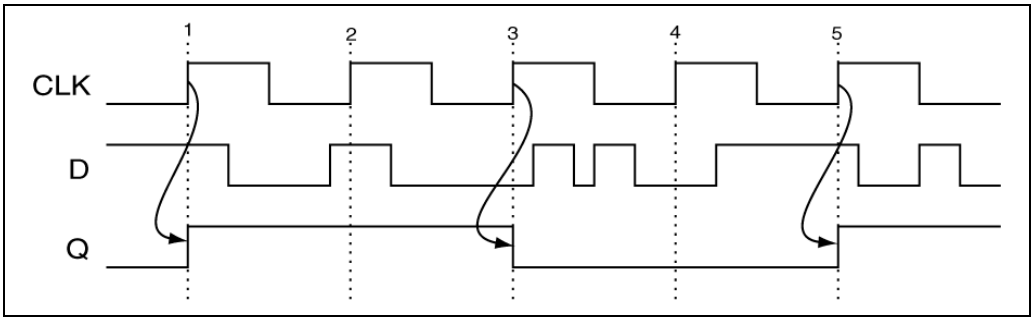
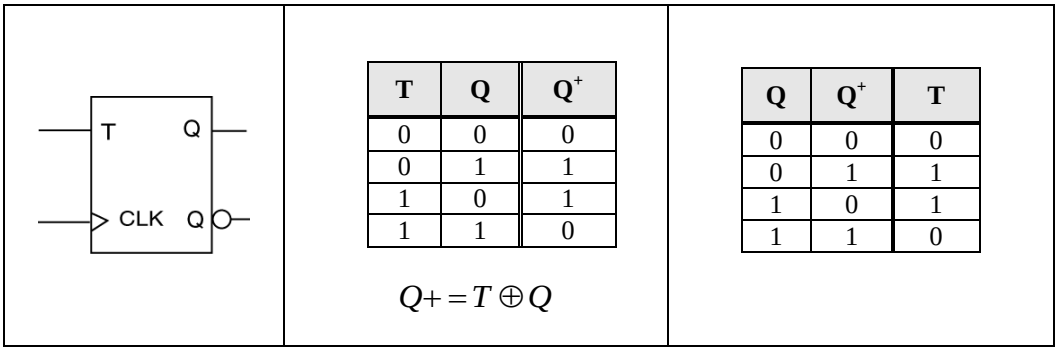


Figure 25.2: An example timing diagram for the D flip-flop.

25.2.2 The T Flip-Flop

The T flip-flop is another standard flip-flop out there in digital land. The T in flip-flop is referred to as a *toggle* flip-flop because when the T input is a '1', the output of the T flip-flop toggles state on the active edge of the clock. If the T input is a '0', the state of the flip-flop does not change. You should definitely verify this verbal description of the T flip-flop with both the characteristic and excitation tables shown in Figure 25.3.

Figure 25.3(a) shows the schematic diagram of the T flip-flop while Figure 25.3(b) shows the characteristic table for the T flip-flop. Note that in the characteristic table, the output only changes state when the T input is a '1'. Note that by inspection of the characteristic table, you can write the characteristic equation associated with the T flip-flop (it sure does look like an exclusive OR function). Figure 25.3(c) provides the excitation table of for the T flip-flop, which is a simple rearrangement of the T flip-flop's characteristic table.



T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

$$Q+ = T \oplus Q$$

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

(a) (b) (c)

Figure 25.3: The schematic symbol (a), characteristic table and characteristic equation (b), and excitation table for the T flip-flop (c).

Figure 25.4 shows a timing diagram that demonstrates the operation of the RET T flip-flop of Figure 25.3(a). Once again, the output of this device can only change on the rising clock edge, which are nicely delineated in Figure 25.4. The initial state of the T flip-flop output Q is a '1' as shown in the timing diagram. At the first rising clock edge, the state of the T input is a '1'; this causes the output of the T flip-flop to toggle state (thus the name toggle flip-flop). Note that the output of flip-flop transitions from '1' to '0' as a result of the rising clock edge and the fact that the T input is in a high state. On the second clock edge, the output toggles again because the T input is once again at a high state. The changes in value of the T input between the first and second clock edges have no affect on the state of the flip-flop because the T flip-flop is only active on the rising edge of the clock. Since the T input is a '0' at the third clock edge, the output of the flip-flop does not change state. On the third and fourth clock edges, the output once again changes state because in both of these instances, the T input is asserted.

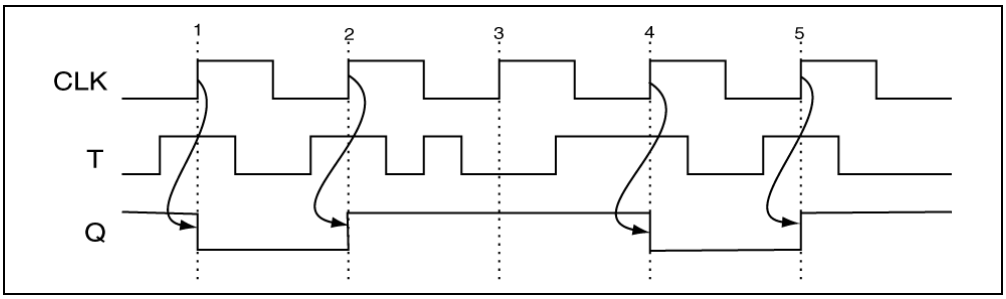


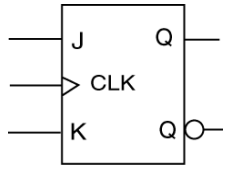
Figure 25.4: An example timing diagram showing the operation of the T flip-flop.

25.2.3 The JK Flip-Flop

The JK flip-flop is the final standard flip-flop that we'll examine. No one really knows what exactly the JK stands for but as you will see, the JK flip-flop shares many of the same operating characteristics as the SR latch. Figure 25.5(a) shows the schematic symbol for the JK flip-flop while Figure 25.5(b) shows the associated characteristic. We can generate the accompanying characteristic equation by dropping the $\mathbf{Q}^+$ column of the characteristic table into a K-map.

The JK flip-flop operates as follows: there are four possible input combinations of the J and K variables. For JK = "00", the output of the flip-flop does not change state (hold condition). For JK = "01", the output of the flip-flop always resets (clear condition). For the JK = "10" condition, the output of the flip-flop always sets (set condition). For the JK = "11" condition, the output of the flip-flop toggles its current state. You should be able to see these actions from examining the characteristic table of Figure 25.5(b).

You should also note from Figure 25.5(b) that the first three JK conditions are the same as the first three conditions for the SR latch. The main difference here is that the JK flip-flop uses the JK = "11" input condition to toggle the current output state of the flip-flop. Figure 25.5(c) show the excitation table for the JK flip-flop. The important thing to note in this table is that the each of the four possible state changes can be caused by two different input conditions on the JK inputs in a way that was similar to the SR latch. For example, a state change of $(0 \rightarrow 0)$ occurs as a result of either a JK = "01" (reset condition) or a JK = "00" (hold condition). The first column of Figure 25.5(c) lists these two conditions.



J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$Q+ = J\overline{Q} + \overline{K}Q$$

Q	Q^+	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

(a) (b) (c)

Figure 25.5: The schematic symbol (a), characteristic table and characteristic equation (b), and excitation table for the JK flip-flop.

A timing diagram shows the true operation of the JK flip-flop. The JK flip-flop of Figure 25.5(a) is a RET device which means the JK flip-flop's outputs can only change on the rising edge of the clock. Another way of saying this is that the output transitions of this device are synchronized to the rising edge of the clock. In order to provide a **Q** output waveform, you must be given the initial state of the **Q** output. In the case of the Figure 25.6, the output is in a low state. At the first rising clock edge, the output of the JK flip-flop toggles due to the fact that both the J and K inputs are '1' (the toggle condition for the JK flip-flop). At the second clock edge, JK = "00" which is the hold condition for the flip-flop and thus no output conditions occur. At the third clock edge, the output is reset because of the clear condition (JK = "01") on the flip-flop inputs. The fourth clock edge finds that JK = "11" which is yet another toggle condition and cause the flip-flop to change state. The fifth clock edge has no effect on the state of the flip-flop due to the fact that despite the presence of a set condition (JK = "10"), the flip-flop's state is already at '1'.

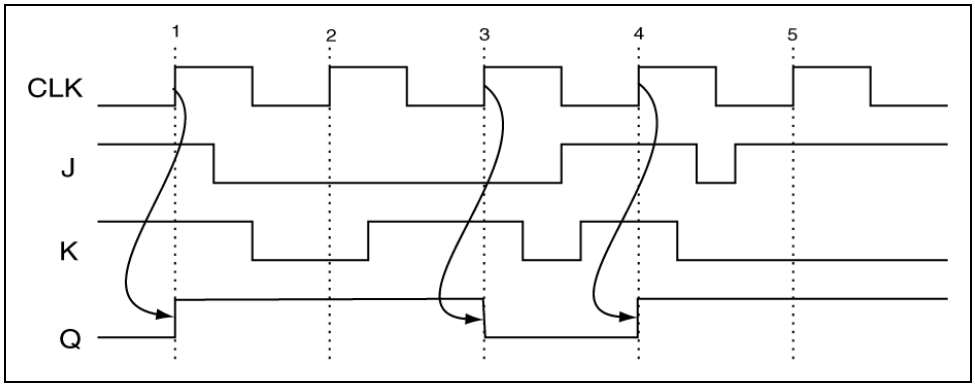


Figure 25.6: An example timing diagram for the JK flip-flop.

25.2.4 The Big D, T, and JK Flip-Flop Summary

Table 25.1 shows everything you were hoping not to know about flip-flops. In theory, it would not be too tough for you to memorize this stuff. Such an endeavor would not be overly taxing in that these devices simply make sense if you stare at them for a few minutes. Moreover, understanding the basic operation of the standard flip-flops provides you with a solid foundation in sequential circuit design.

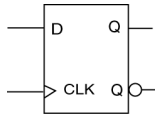
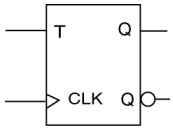
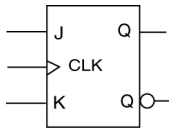
<table>
<tr><th>Type</th><th>Symbol</th><th>Characteristic Table</th><th>Characteristic Equation</th><th>Excitation Table</th></tr>
<tr>
<td>D</td>
<td> 
</td>
<td>
<table>
<tr><th>D</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td>
<td>$Q+ = D$</td>
<td>
<table>
<tr><th>Q</th><th>Q+</th><th>D</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td>
</tr>
<tr>
<td>T</td>
<td> 
</td>
<td>
<table>
<tr><th>T</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>$Q+ = T \oplus Q$</td>
<td>
<table>
<tr><th>Q</th><th>Q+</th><th>T</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
</tr>
<tr>
<td>JK</td>
<td> 
</td>
<td>
<table>
<tr><th>J</th><th>K</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>$Q+ = J\overline{Q} + \overline{K}Q$</td>
<td>
<table>
<tr><th>Q</th><th>Q+</th><th>J</th><th>K</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>-</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>-</td></tr>
<tr><td>1</td><td>0</td><td>-</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>-</td><td>0</td></tr>
</table>
</td>
</tr>
</table>

Table 25.1: The major characteristics of the D, T, and JK flip-flops.

25.3 VHDL Models for Basic Sequential Circuits

This section shows the some of the various methods used to model basic sequential circuits using VHDL. This discussion is somewhat strange because we toss out a bunch of the ideas we've just been working on as we switch over to modeling sequential circuits with VHDL. The truth is that, in general, VHDL models sequential circuits at a higher level than the level we used to describe those circuits in the previous sections. This is good because you'll never need to use VHDL to model a simple latch on the gate-level (as presented in the previous chapter). Although you're hopefully well on your way to understanding each of the three standard types of flip-flops, the approach we'll take now is to limit our discussion primarily to VHDL models for D flip-flops.

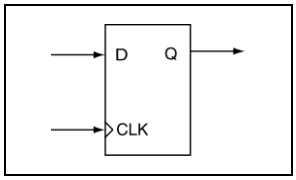
As you for sure know by now, VHDL is a massively versatile language based on its ability to describe digital circuits. While it is possible and in some cases desirable to use dataflow models to describe storage elements in VHDL, clocked storage elements such as flip-flops are best described using behavior models. This fact should become more obvious as we examine a few VHDL models.

25.3.1 Simple Storage Elements Using VHDL

The general approach to learning about simple storage elements in digital design is to study the properties of a basic cross-coupled cell. Examining a simple model of a D flip-flop is the best approach to learning how VHDL models simple storage elements. In other words, the study of VHDL descriptions of storage elements starts at the D flip-flop. The VHDL examples presented are the basic edge-triggered D flip-flop.

Example 25-1

Write the VHDL code that describes a D flip-flop shown on the right. Use a behavioral model in your description.



Solution: Figure 25.7 shows the solution to Example 25-1. Listed below are a few interesting things to note about the solution.

- The given architecture body describes the *my_d_ff* version of the *d_ff* entity.
- Because example requested the use of a behavioral model, the architecture body is comprised primarily of a *process* statement. The statements within the *process* execute sequentially. The *process* executes each time a change occurs in any of the signals in the *process*'s *sensitivity list.* In this case, the statements within the *process* execute each time there is a change in logic level of the *D* or *CLK* signals.
- The *if* statement uses the *rising_edge()* construct to indicate that changes in the circuit output only on the rising edge of the *CLK* input. The *rising_edge()* construct is actually an example of a VHDL function which has been defined in one of the included libraries. The circuit is synchronous based on the given VHDL model; this means that changes in the circuit's output are synchronized to the rising edge of the clock signal. In this case, the action is a transfer of the logic level on the *D* input to the *Q* output.
- The process has a label: *dff*. The VHDL language does not require this but including process labels promotes self-commenting code and increases its readability and understandability.

```
------------------------------------------------------------
-- Model of a simple D Flip-Flop
------------------------------------------------------------
entity d_ff is
    port ( D, CLK : in std_logic;
                Q : out std_logic);
end d_ff;

architecture my_d_ff of d_ff is
begin
   dff:  process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process dff;
end my_d_ff;
```

Figure 25.7: Solution to Example 25-1.

The D flip-flop is best known and loved for its ability to store (save, remember) a single bit. The way that the VHDL code listed in Figure 25.7 is able to store a bit is not obvious, however. The bit-storage capability in the VHDL is *implied* by the both the VHDL code and the way the VHDL code is interpreted. The implied storage comes about as a result of not providing a condition that indicates what should happen if the listed **`if`** condition is not met. We've been referring to this condition as the "catch-all" condition. In other words, if the **`if`** condition is not met, the device does not change the current value of **Q** and therefore must "remember" that current value.

The "remembering" of the current value, or state, constitutes the famous bit storage quality of a flip-flop. If you have not specified what the outputs should be for every possible set of input conditions, there will be conditions where the changes in the output are not defined. In these cases, the option taken by VHDL is not to change the current output. By definition, if the inputs change to an unspecified state, the outputs remain unchanged. In this case, the outputs associated with the previous set of inputs are thought of as being "remembered". VHDL uses this mechanism, as strange and interesting as it sounds, to *induce* memory in VHDL. This is also why we were so careful to always provide a "catch-all" condition for our previous VHDL models. It was the inclusion of the catch-all condition in our models that assured us that we were not inducing memory elements[271].

In terms of the D flip-flop shown in Example 25-1, the only time the output is specified is for that delta time associated with the rising edge of the clock. The typical method used to provide a catch-all condition in case the **`if`** condition is not met is with an **`else`** clause. A quick way to tell if you've induced a memory element is to look for the presence of an **`else`** clauses associated with the **`if`** statement. Once again, it is the **else** statement that provides the "catch-all" characteristic of the model.

The previous two paragraphs are vastly important to understanding VHDL; the concept of inducing memory in VHDL is massively important to digital circuit design. By definition, the modeling of all sequential circuits is dependent on this concept. This somewhat cryptic method used by VHDL to induce memory elements is a byproduct of behavioral modeling based solely on the interpretation of the VHDL source code. Even if you'll only be using VHDL to design combinatorial circuits, you need to understanding these concepts.

[271] Recall that we have mostly dealt with combinatorial circuits until now. Since the circuits are combinatorial, provided a catch-all statement assures that we will not generate a latch which ensures our combinatorial circuit is actually combinatorial.

One of the classic warnings generated by the VHDL synthesizer is notification that your VHDL code has generated a "latch". Despite the fact that this is "only a warning", if you did not intend to generate a latch, you should strive modify your VHDL code in such as way as to make this warning go away. Assuming you did not intend to generate a latch, the cause of your problem is that you've not explicitly provided an output state for all the possible input conditions. Because of this, your circuit will need to remember the previous output state so that it can provide an output in the case where you've not explicitly listed the current input condition.

25.3.2 Synchronous and Asynchronous Flip-Flop Inputs

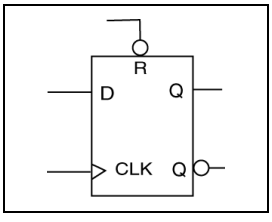
The flip-flops we've described up to this point have been what are considered *synchronous* circuits. In the context of flip-flops, "synchronous" refers to the fact that the changes in the state of the flip-flop are synchronized to the active clock edge. In the case of the flip-flops we've been developing, changes in the state of the flip-flop were synchronized to the rising clock edge. In reality, most flip-flops out there in digital land have the ability to change state either synchronously (generally based on the clock input) or *asynchronously*. For the asynchronous case, some inputs can cause state changes that are not synchronized with the clock. In this section, we want to look at a few of those cases.

Dealing with asynchronous flip-flop inputs is somewhat troubling because the actual circuitry that implements these asynchronous features is beyond the scope of an introductory digital design course. This is OK though because we can then abstract the concept to a higher level and deal with it there. The D, T, and JK flip-flops we're dealing have inputs that force the state of the flip-flop to change at a time other than on the active clock edge. In other words, the effect that these asynchronous signals have on the flip-flops occurs immediately, regardless of whether a clock edge is present or not.

As you would probably guess, since there are two different things you can do to a flip-flop's output, namely make it a '1' or make it a '0' ("set" or "clear", respectively). Not surprisingly, there are usually two different asynchronous inputs to a flip-flop: the *set* and *reset* input. These inputs are usually active low which means when the asynchronous input signal is low, some action occurs on the output of the flip-flop. Flip-flop diagrams use a bubble to indicate the logic level of the input. Most often, flip-flops use an "S" to represent the input that asynchronously sets the state of the flip-flop and use an "R" to represent the input that resets the state of the flip-flop. For the record, flop-flops sometimes list the set input as a "preset"; the reset input is sometimes listed as a "clear" input. Let's take a look at a few examples of flip-flops with asynchronous inputs.

Example 25-2

Write the VHDL code that describes a D flip-flop shown on the right. Use a behavioral model in your description. Consider the **R** input to be an active-low, asynchronous input that clears the D flip-flop outputs when asserted.



Solution: Figure 25.8 shows the VHDL model for this solution. The first thing you should notice about this solution is that it is amazing similar to the VHDL model for the standard D flip-flop. There are a few important items worth noting about this solution.

- The R input is included in the process sensitivity list. If R was not included here, the flip-flop would not operate properly (but it probably would synthesize).

- The associated VHDL code evaluates the reset input (R) before the CLK input. Because of the sequential nature of the statements inside of process statements, if the first ***if*** clause evaluates are true, the ***elsif*** clause is not evaluated and the statement associated with the ***if*** clause is executed. What makes the R input of the flip-flop asynchronous is the fact that the model evaluates the condition of the R signal before the CLK signal. In other words, the R input has precedence of the CLK input in this case.

- The active low nature of the flip-flop is modeled by making the action state of R to be '0' as shown in the conditional portion of the ***if*** clause. In other words, when R is '0', the output of the flip-flop resets if it is currently set.

```
-----------------------------------------------------------------
-- RET D Flip-flop model with active-low asynchronous reset input.
-----------------------------------------------------------------
entity d_ff_nr is
    port (D,R,CLK : in  std_logic;
                Q : out std_logic);
end d_ff_nr;

architecture my_d_ff_nr of d_ff_nr is
begin
   dff: process (D,R,CLK)
   begin
      if (R = '0') then
         Q <= '0';
      elsif (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process dff;
end my_d_ff_nr;
```

Figure 25.8: VHDL model of D flip-flop with active low asynchronous clear.

We can use our friend the timing diagram to model the operation of the flop-flop in Figure 25.8. Figure 25.9 shows a timing diagram associated with this example. Of course, we need to list a few fantastically interesting things to note regarding this timing diagram.

- Since the R input is low at the start of the timing diagram, the output of the flip-flop is in the reset state. Using the R input in this manner is typical in timing diagrams and should be feature you always look for when asked to deal with timing diagrams.

- On the first rising clock edge, the D flip-flop acts as you expect. In this case, the model ignores the R input because it is a '1'; the model then evaluates the other two inputs.

- Between the second and third clock edges, the R input goes low. Once this occurs, the output immediately resets[272]. The R input returns to its non-active state (the '1' state) soon afterwards. Returning to the non-active state has no effect on the state of the flip-flop. This is typically how synchronous set and reset inputs act on flip-flops.

[272] There is actually an associated propagation delay associated with this state transition but we're still modeling these flip-flops using an ideal model.

- The timing diagram of Figure 25.9 has two reset "pulses". Both of these pulses are negative pulses (high-to-low-to-high) and both pulses cause the output of the flip-flop to reset.

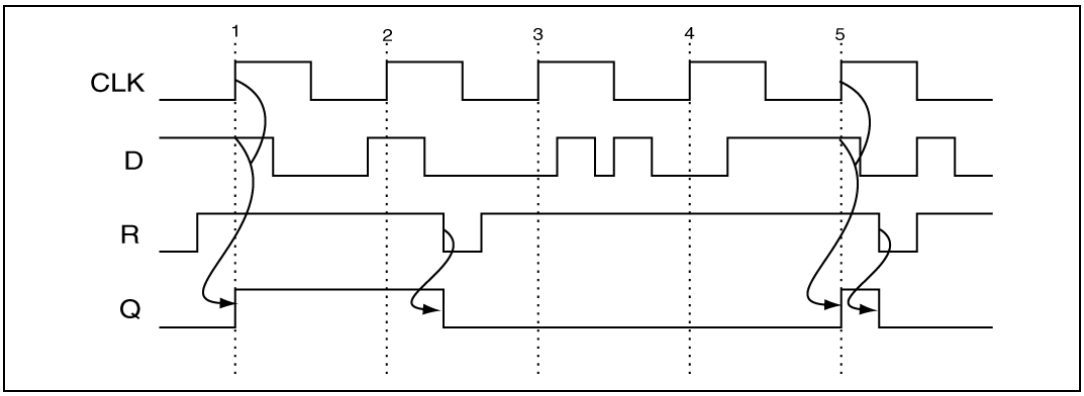
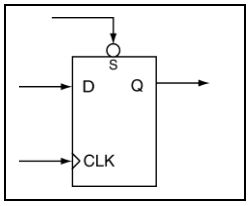


Figure 25.9: Timing diagram associated D Flip-flop with asynchronous active low clear.

Example 25-3

Write the VHDL code that describes a D flip-flop shown on the right. Use a behavioral model in your description. Consider the **S** input to be an active-low, synchronous input that sets the D flip-flop outputs when asserted.



Solution: Figure 25.10 shows the solution to Example 25-3. There are a few things of interest regarding this solution.

- You would not know from the block diagram that the **S** input is synchronous; the problem (or datasheet or whatever) needs to state it directly.

- The **S** input to the flip-flop becomes synchronous by only allowing it to affect the operation of the flip-flop on the rising edge of the clock. In other words, the **S** input only acts on the flip-flop outputs on the active clock edge.

- On the rising edge of the clock, the **S** input takes precedence over the D input because the model evaluates the state of the **S** input prior to examining the state of the D input. In an **`if-else`** statement, once one condition evaluates as true, none of the other conditions are checked. In other words, the D input transfers to the output only the rising edge of the clock and only if the **S** input is not asserted. This once again emphasizes the sequential nature of statements appearing inside process statements.

```
-----------------------------------------------------------------
-- RET D Flip-flop model with active-low synchronous set input.
-----------------------------------------------------------------
entity d_ff_ns is
    port (D,S,CLK : in  std_logic;
                  Q : out std_logic);
end d_ff_ns;

architecture my_d_ff_ns of d_ff_ns is
begin
   dff:  process (D,S,CLK)
   begin
      if (rising_edge(CLK)) then
         if (S = '0') then
            Q <= '1';
         else
            Q <= D;
         end if;
      end if;
   end process dff;
end my_d_ff_ns;
```

Figure 25.10: The VHDL code solving Example 25-3.

Figure 25.11 shows a timing diagram associated with Example 25-3. Here are the cool things to note about the timing diagram in Figure 25.11.

- For this example timing diagram, the starting state of Q was provided in the timing diagram. In other words, there was no way you could figure out what it was from the problem statement; the problem had to provide you with this information.

- The flip-flop ignores the S pulse between the first and second rising clock edge because the S input in this example is synchronous (meaning that its actions are synchronized to the clock edge). The same is true of the S pulse between the third and fourth clock edges.

- The flip-flop output sets on the fifth clock edge because the S input was in its active state at the arrival of the active clock edge.

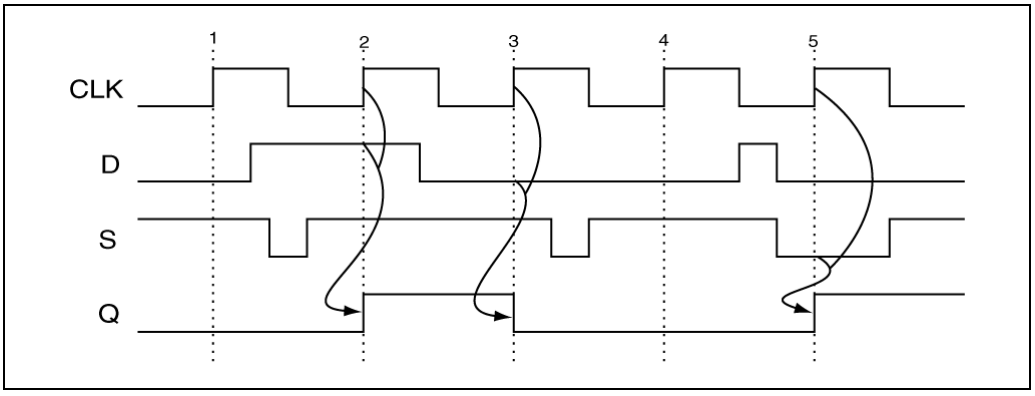
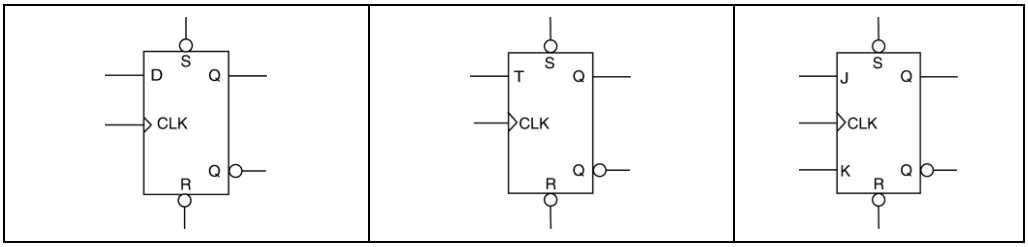


Figure 25.11: Timing diagram associated with Example 25-3.

The previous two examples are important for several reasons. First, the examples provided insight in to modeling asynchronous and synchronous inputs using VHDL. The timing diagrams associated with these examples clearly show these characteristics. Secondly, a significant port of sequential circuit design deals with timing issues associated with where to place control signals relative to the active clock edge. As with these examples, you need to keep in mind the sequential nature of VHDL behavioral modeling as you're generating your VHDL models in order to ensure proper circuit operation. These two examples are once again massively important. If you totally have nothing better to do, go back and closely compare these two VHDL models.

25.3.3 Flip-flops with Multiple Control Inputs

Flip-flops models often contain both preset and reset inputs. If this is the case, you need to be careful to specify the flip-flop operation when both of these inputs are asserted[273]. We're not going to do that now but we are going to do a final few examples regarding D, T, and JK flip-flops that include both preset and reset inputs. The final set of D, T, and JK devices with asynchronous presets and clears are shown in Figure 25.12. Once again, examining a few timing diagrams should drive home how both the asynchronous and synchronous inputs affect the output of the flip-flops. The following timing diagram analysis is primarily concerned with the affects the asynchronous inputs have on the outputs since we see these quite often in digital design land.



(a) (b) (c)

Figure 25.12: The fully loaded set of D, T, and JK flip-flops.

The next three examples are based on the flip-flop models shown in Figure 25.12. Each of the flip-flops of Figure 25.12 has S and R inputs, which are asynchronous inputs. Note that these inputs are active low (the negative logic thing). We consider the D, T, and JK inputs to be synchronous. For each of the following three timing diagrams, the initial state of the flip-flop is unknown until one of the synchronous signals put the flip-flop into a known state. As you'll find out, this is a common approach in these types of problems and sequential-based digital design in general.

The timing diagram in Figure 25.13 is based on the schematic diagram of Figure 25.12(a): a RET D flip-flop with active low asynchronous preset and clear. The output of the D flip-flop goes to an initial '1' state by the low pulse on the S input. The first and second clock edges transfer the D inputs of '0' and '1' to the output of the device. The first low pulse of the R signal represents a reset, which makes the state of the device a '0' independent of the active edge of the clock. Note that when the R signal returns to the '1' state, the output of the device remains in the '0' state; this represents normal flip-flop operation for asynchronous inputs. The output of the device once again follows the D input on the next two clock edges. The second low pulse on the R signal does not affect the state of the flip-flop since the

[273] In general, all VHDL models need to specify what circuit operation in every possible scenario. If you don't 100% specify circuit operation in the circuit model, the synthesizer will most likely specify it for you. Often times the synthesizer will generate a warning regarding this condition, but don't count on it.

flip-flop is current in a ‘0’ state. The final low pulse on the S signal sets the output of the flip-flop. Once again, the output of the flip-flop remains set after the low pulse of S returns to the high state.

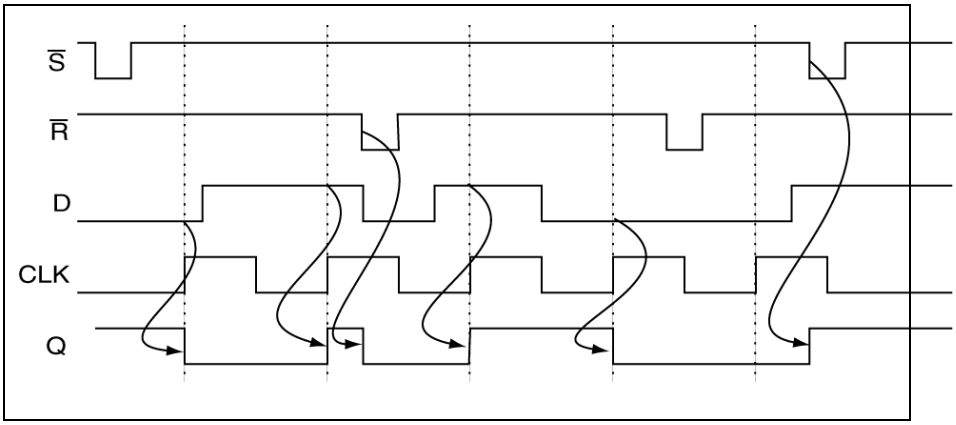


Figure 25.13: Example timing diagram for a RET D flip-flop with active low asynchronous preset and clear (see Figure 25.12(a)).

The timing diagram in Figure 25.14 is based on the schematic diagram of Figure 25.12(b): a RET T flip-flop with active low asynchronous preset and clear. The initial low pulse on the S input forces the output of the flip-flop into the ‘1’ state. The first active clock edge causes the output of the flip-flop to toggle. Since the output of the device is currently in the low state, the first low pulse on the R signal does not change the flip-flop’s output. The second pulse on the S signal, however, does cause the device to change state from a reset state to a set state. The clock edge following the S pulse and the high state of the T input cause the device to reset once again. The final pulse on the R input causes the flip-flop to return to the ‘0’ state.

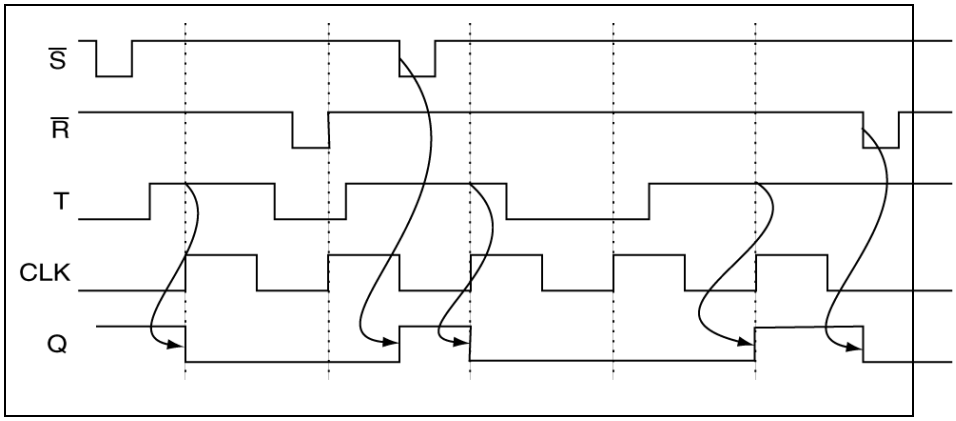


Figure 25.14: Example timing diagram for a RET T flip-flop with active low asynchronous preset and clear (see Figure 25.12(b)).

The timing diagram in Figure 25.14 is based on the schematic diagram of Figure 25.12(b): a RET JK flip-flop with active low asynchronous preset and clear. The initial low pulse on the R inputs forces the output of the flip-flop into the ‘0’ state; the flip-flop remains in this state after the low R pulse returns to its high state. The JK=”10” causes the flip-flop to set on the next active clock edge. Because the flip-flop is already set when the first low pulse arrives on the S input, the device output does not change.

The second low pulse on the R input causes a state change in the flip-flop from high to low. The final low pulse on the flip-flop causes a similar change near the end of the timing diagram.

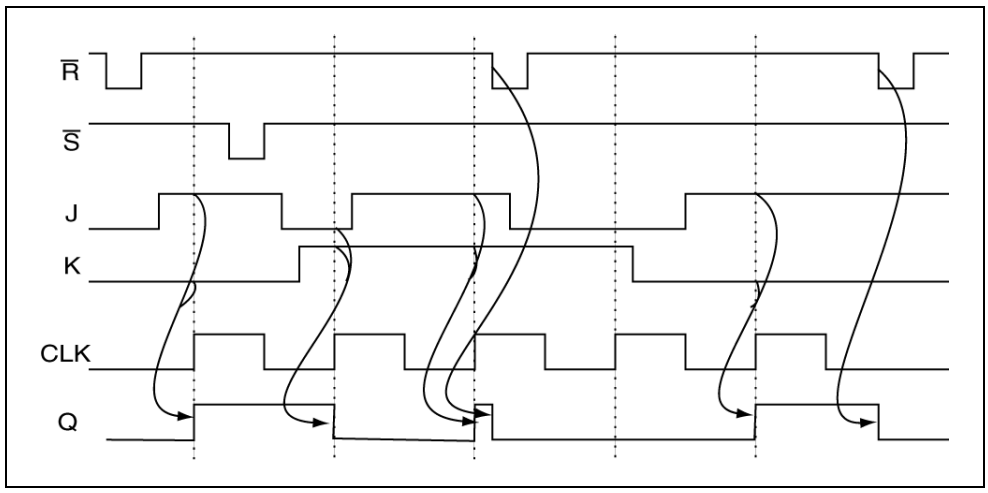
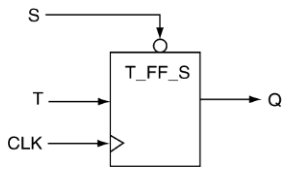


Figure 25.15: Example timing diagram for a RET JK flip-flop with active low asynchronous preset and clear (see Figure 25.12(c)).

The following example has some interesting properties. The good news is that we'll be able to take a look at these interesting properties and maybe learn something useful from it. The bad news is that modeling T flip-flops with VHDL is more of an academic exercise rather that something that is useful or done often in digital design land. The truth is, and you'll discover it when you take a look at this problem, is that D flip-flops are so much easier to model in VHDL that you're rarely see T or JK flip-flops models[274].

Example 25-4

Write the VHDL code that describes a T flip-flop shown on the right. Use a behavioral model in your description as well as the characteristic equation for a T-FF. Consider the S input to be an active-low, asynchronous input that sets the T flip-flop outputs when asserted.		

Solution: Figure 25.16 shows the solution to Example 25-4. This example has some massively important techniques associated with it that are well worth mentioning below.

- This implementation of a T flip-flop demonstrates a unique quality of the D flip-flop. The output of a D flip-flop is only dependent upon the D input and is not a function of the present output of the flip-flop. The output of a T flip-flop is dependent upon both the T

[274] Except in the problem set associated with this chapter. Sad but true: academic exercises make us more academically fit.

input and the current output of the flip-flop. This adds a certain amount of extra complexity to the T flip-flop model as compared to the D flip-flop as shown in Figure 25.16. The T flip-flop model in Figure 25.16 uses a temporary signal in order to use the current state of the flip-flop as in input. In other words, since Q appears as a port to the entity it must be assigned a mode specifier, and in this case, it has been assigned a mode specifier of "out". Signals declared as outputs can therefore not appear on the right side of a signal assignment operator. The standard approach to bypassing this apparent limitation in VHDL is to use intermediate signals which, as opposed to port signals, do not have mode specifications and can thus be used as either inputs or outputs (can appear on both sides of the signal assignment operator) in the body of the architecture. The approach is to not only manipulate the intermediate signal in the body of the architecture but to also use a concurrent signal assignment statement to assign the intermediate signal to the appropriate output. Note that in the key statement in the solution shown in Figure 25.16 that the intermediate signal appears on both sides of the signal assignment operator. We've seen this coding style before; get used to it; become one with it.

- This code uses the characteristics equation of a T flip-flop in its implementation. We technically used a characteristic equation when we implemented the D flip-flop but since the characteristic equation of a D flip-flop is relatively trivial ($\mathbf{Q}^+ = D$), you may not have been aware of it.

- Where there are certain advantages to using T flip-flops in some conditions, D flip-flops are generally the storage element of choice when using VHDL. If you don't have a specific reason for using some type of flip-flop other than a D flip-flop, you probably shouldn't unless you're friends are easily impressed[275].

```
-------------------------------------------------------------------
-- RET T Flip-flop model with active-low asynchronous set input.
-------------------------------------------------------------------
entity t_ff_s is
    port ( T,S,CLK : in  std_logic;
                   Q : out std_logic);
end t_ff_s;


architecture my_t_ff_s of t_ff_s is
   signal s_tmp : std_logic; -- intermediate signal declaration
begin
   tff:  process (T,S,CLK)
   begin
      if (S = '0') then
         Q <= '1';
      elsif (rising_edge(CLK)) then
         s_tmp <= T XOR s_tmp; -- temp output assignment
      end if;
   end process tff;

   Q <= s_tmp; -- final output assignment

end my_t_ff_s;
```

Figure 25.16: Solution to Example 25-4.

[275] If you actually have any friends.

25.4 Inducing Memory: Dataflow vs. Behavior Modeling

A major portion of digital design deals with sequential circuits. In addition, most sequential circuit design is synchronized to a clock edge. In other words, output changes in sequential circuits generally only occur on an active clock edge. The introduction to memory elements in VHDL presented in this section may lead the reader to think that memory in VHDL is only associated with behavioral modeling, but this is not the case. The same concept of inducing memory holds for dataflow modeling as well: not explicitly specifying an output for every possible input condition generates memory. On this note, checking for unintended memory element generation is one of the duties of the digital designer. As you would imagine, memory elements add an element of needless complexity to the synthesized circuit.

One common approach to learning the syntax and mechanics of new computer languages is to implement the same task in as many different ways as possible. This approach utilizes the flexibility of the language and is arguably a valid approach to learning a new language. This is also the case in VHDL. However, probably more so in VHDL than other languages, there are specific ways of doing things and you the digital designer should always do these things in these specific ways. Although it would be possible to generate flip-flops using dataflow models, most knowledgeable people examining your VHDL code would not initially be clear as to what exactly you're doing. As far as generating synchronous memory elements go, the methods outlined in this section are simply the optimal method of choice. This is one area not be clever with.

Chapter Overview

- While a latch is considered a level-sensitive device since the outputs can change any time the inputs change. When special control inputs are added to latches, name a clock input, and changes in the state of the circuit can only happen on a clock edge, the circuit is considered to be a flip-flop. There are three main types of flip-flops: the D, T, and JK flip-flops.

- Flip-flops are generally considered synchronous circuits in that the state of the flip-flop is synchronized to the active clock edge. Flip-flops can also contain inputs whose effects are not synchronized to the clock edge; these inputs are referred to as asynchronous inputs.

- Memory in VHDL is model as an incompletely specified input condition. If an output is not specified by every possible input condition, the device must "remember" the previous output. Specifying catch-all conditions in VHDL models prevent the VHDL synthesizer from inducing memory.

- Memory in VHDL can be induced with both dataflow and behavioral models. When modeling circuits that are sensitive to clock edges, behavioral models are generally used.

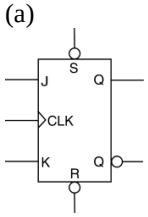
Chapter Exercises

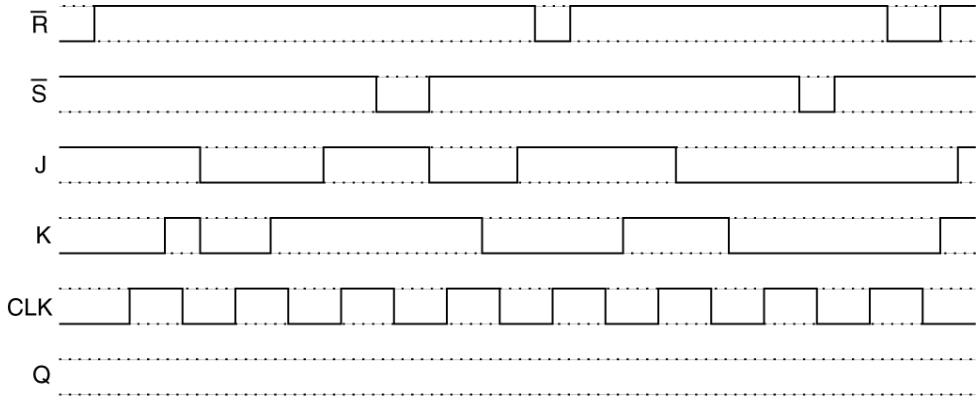
1) Does the following VHDL model describe a sequential or combinatorial circuit? Briefly justify your answer.

```
entity my_ckt1 is
    Port ( ABC : in std_logic_vector(2 downto 0);
           F_OUT : out std_logic);
end my_ckt1;

architecture ckt11 of my_ckt1 is
begin

   my_proc: process (ABC)
   begin
      if (ABC = "000") then
         F_OUT <= '1';
      elsif (ABC = "111") then
         F_OUT <= '0';
      end if;
   end process;

end ckt11;
```

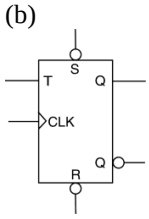
2) Provide the Q output (sometimes labeled as OUTPUT) signal using the associated flip-flops listed below. Consider all S and R inputs to be asynchronous. The asynchronous inputs take precedence over the synchronous inputs. Assume that propagation delays are negligent.

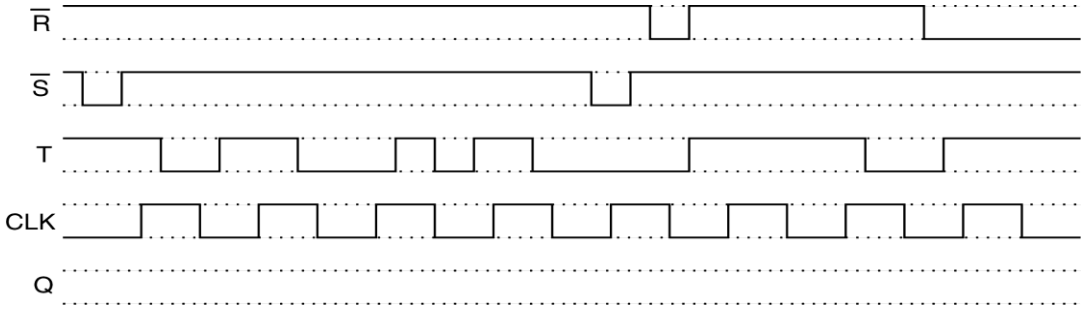
(a)



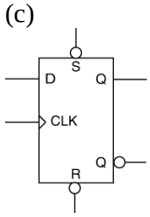


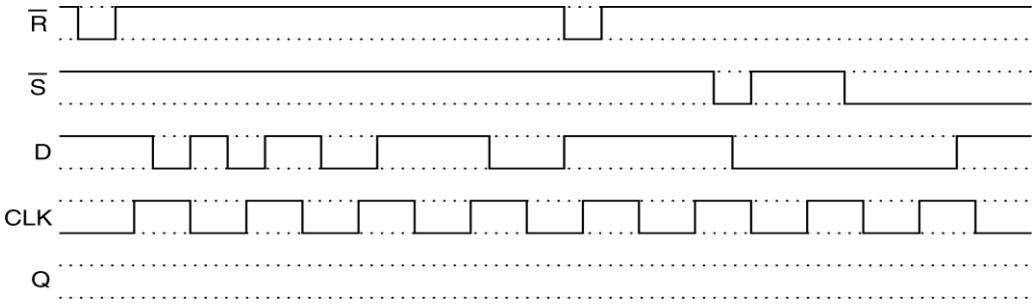
(b)


S
T
Q
CLK
Q
R

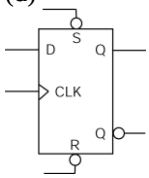

R
S
T
CLK
Q

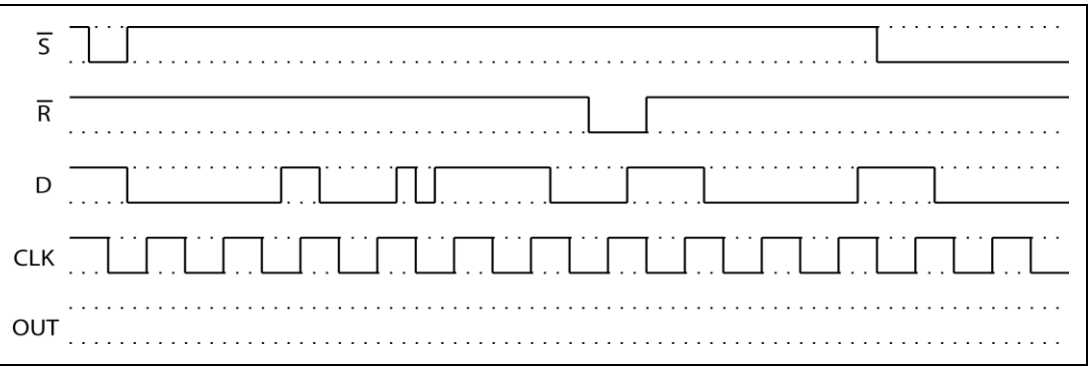
(c)


S
D
Q
CLK
Q
R

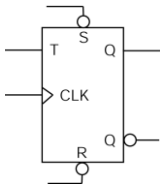

R
S
D
CLK
Q

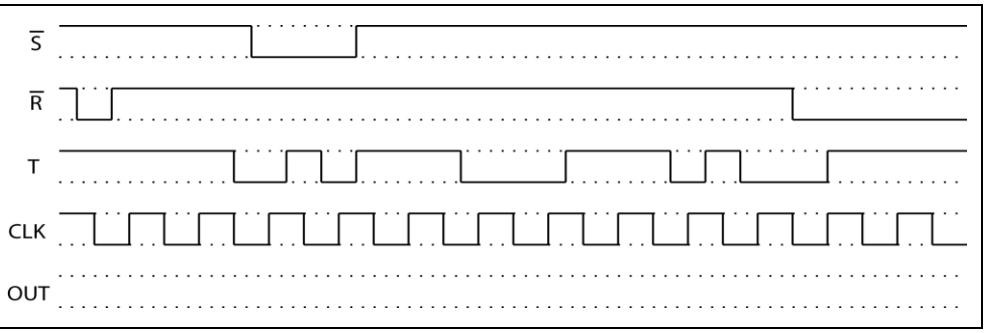
(d)


S
D
Q
CLK
Q
R

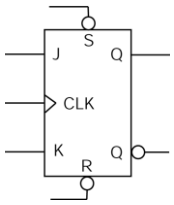

S
R
D
CLK
OUT

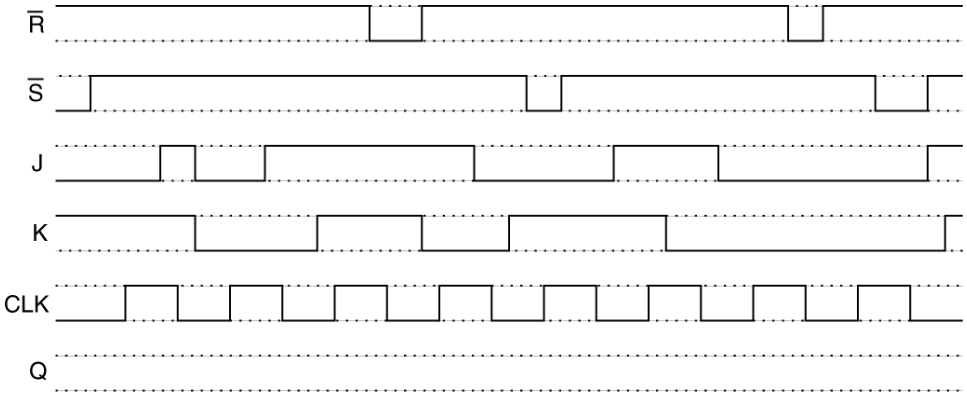
(e)


S
T
Q
CLK
Q
R

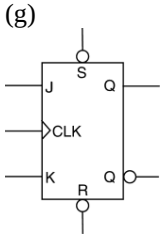

S
R
T
CLK
OUT

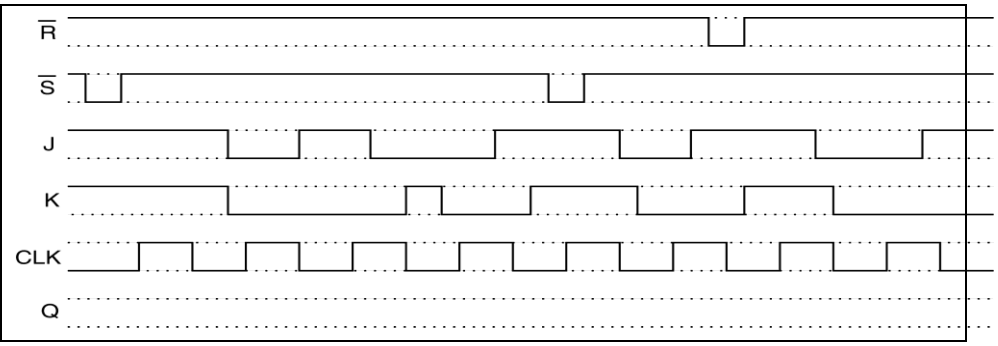
(f)



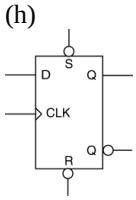


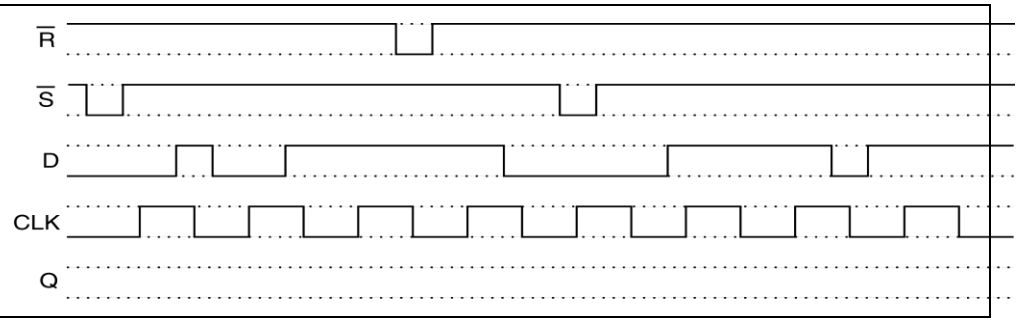
(g)



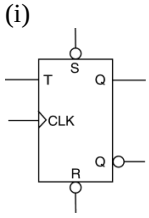


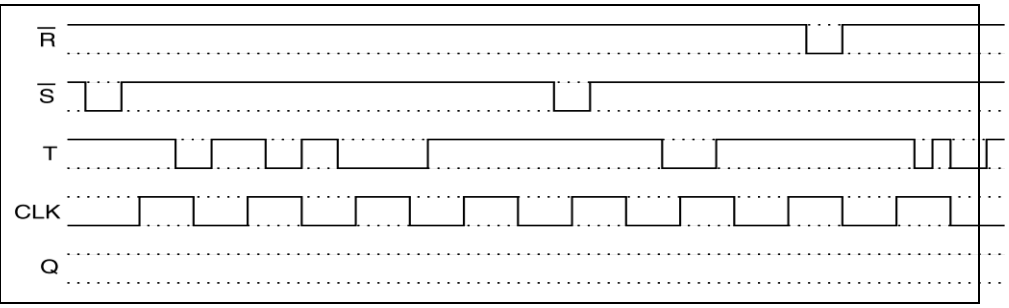
(h)



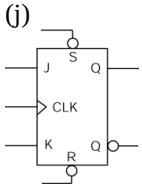


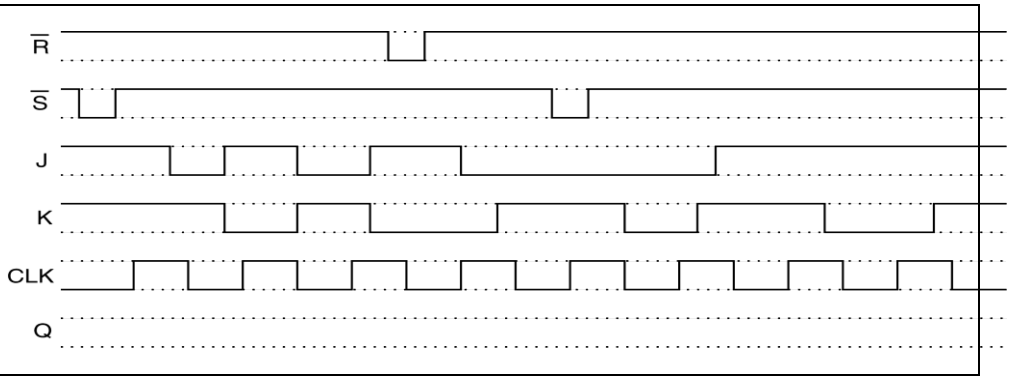
(i)


S
T
Q
CLK
Q
R

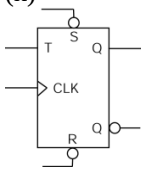

R
S
T
CLK
Q

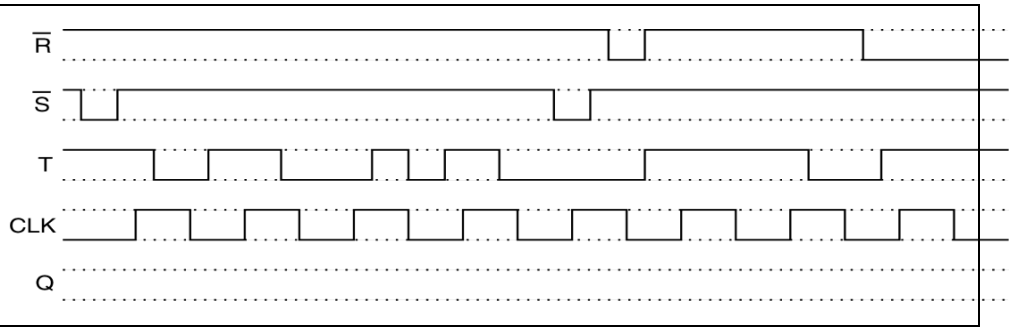
(j)


S
J
Q
CLK
K
Q
R

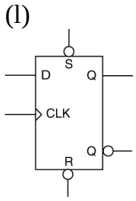

R
S
J
K
CLK
Q

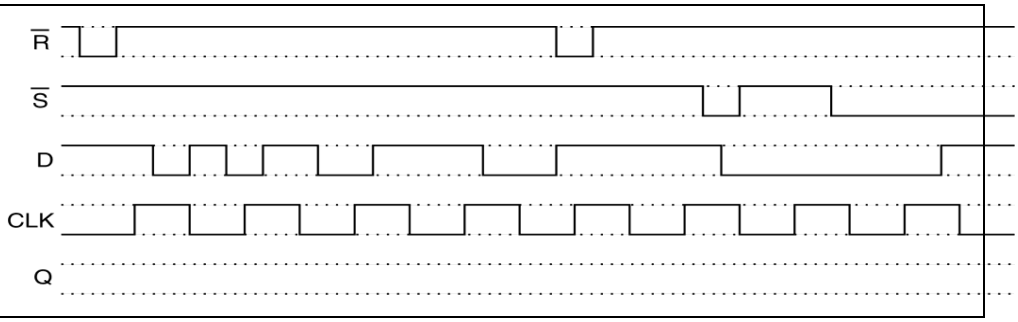
(k)


S
T
Q
CLK
Q
R

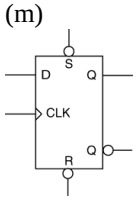

R
S
T
CLK
Q

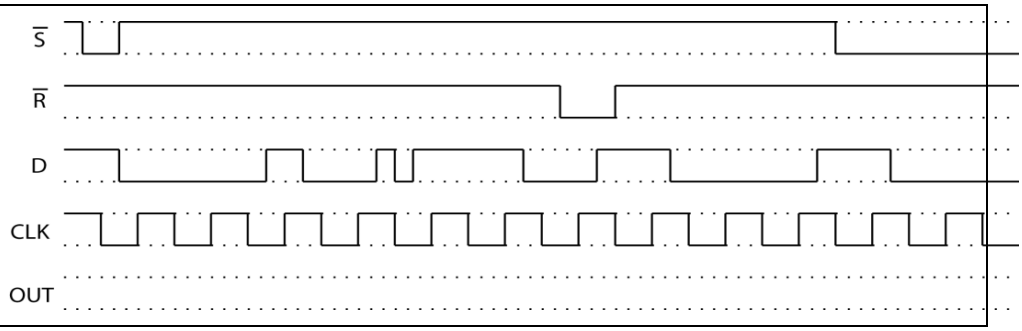
(l)


S
D
Q
CLK
Q
R

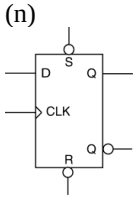

R
S
D
CLK
Q

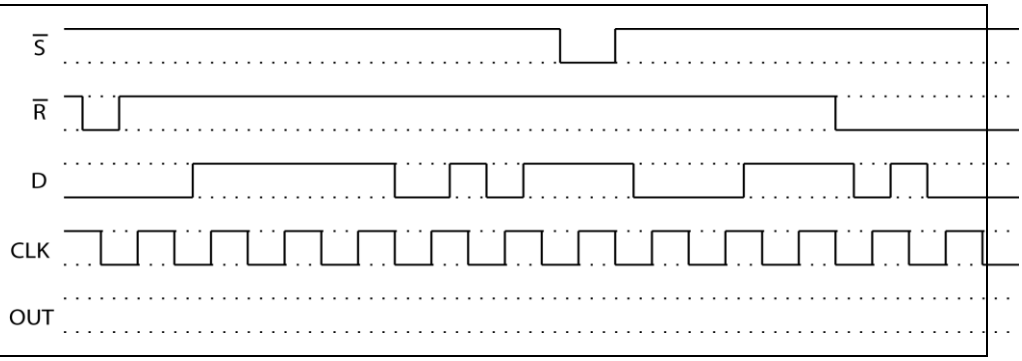
(m)



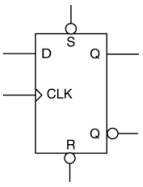


(n)

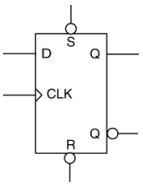




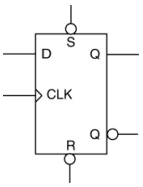
3) Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



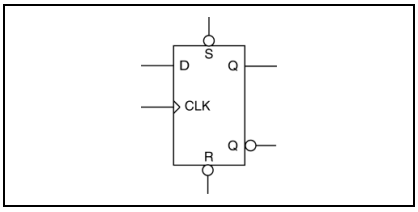
4) Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume the S input takes precedence over the R input in the case where both are asserted simultaneously.



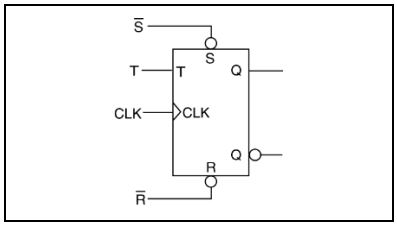
5) Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are synchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



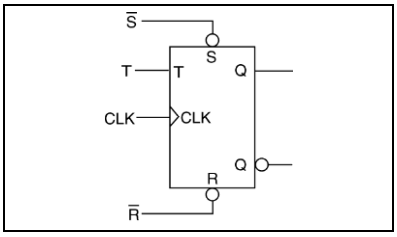
6) Provide a VHDL behavioral model of the D flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. If both the S and R inputs are asserted simultaneously, the output of the flip-flop will toggle.



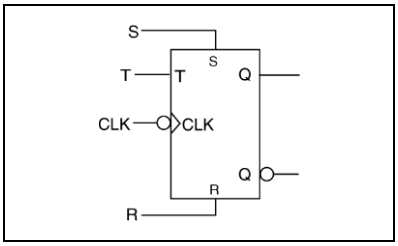
7) Provide a VHDL behavioral model of the T flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously. Implement this flip-flop first using an equation description of the outputs and then using a behavioral description of the outputs.



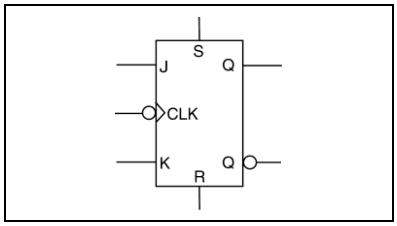
8) Provide a VHDL behavioral model of the T flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



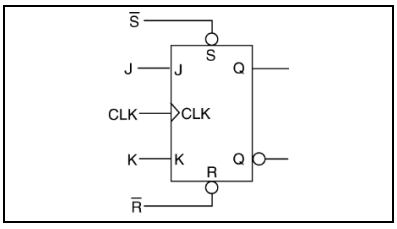
9) Provide a VHDL behavioral model of the T flip-flop shown at the right. The S and R inputs are an active high asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



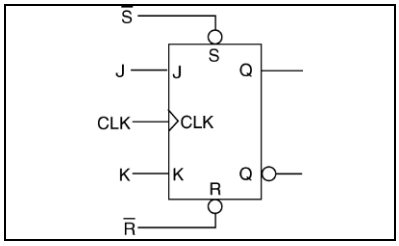
10) Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are an asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously. Implement this flip-flop first using an equation description of the outputs and then using a behavioral description of the outputs.



11) Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are an active low asynchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



12) Provide a VHDL behavioral model of the JK flip-flop shown on the right. The S and R inputs are active low synchronous preset and clear. Assume both the S and R inputs will never be asserted simultaneously.



13) Circle the option that best describes the following problem (RET, FET: rising and falling edge trigger).

RET D flip-flop with complimentary outputs	T flip-flop with single-ended outputs
RET D flip-flop with asynchronous active low S input	FET D flip-flop with asynchronous active high S input
RET D flip-flop with synchronous active low S input	FET D flip-flop with synchronous active high S input
RET D flip-flop with asynchronous active low S input	FET D flip-flop with asynchronous active high S input
RET D flip-flop with synchronous active low S input	FET D flip-flop with synchronous active high S input
RET Smokin Joe Bob Briggs flip-flop	This is not a flip-flop
RET D flip-flop with asynchronous active low S input	FET D flip-flop with asynchronous active high S input
RET D flip-flop with synchronous active low S input	FET D flip-flop with synchronous active high S input
RET D flip-flop with asynchronous active low S input	FET D flip-flop with asynchronous active high S input
RET D flip-flop with synchronous active low S input	FET D flip-flop with synchronous active high S input

```
entity dff3 is
   port ( D,CLK,S : in std_logic;
                Q : out std_logic);
end dff3;

architecture dff3 of dff3 is
begin
   dffx:  process (D, CLK)
   begin
      if (S = '1') then
         Q <= '1';
      elsif (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process dffx;
end dff3;
```

Design Problems

1. Configure a T flip-flop such that it will divide the frequency of a clock signal by a factor of two.

2. Provide the logic to turn a D flip-flop into a T flip-flop.

3. Provide the logic to turn a T flip-flop into a JK flip-flop.

4. Provide the logic to turn a JK flip-flop into a D flip-flop.

26 Chapter Twenty-Six

(Bryan Mealy 2012 ©)

26.1 Chapter Overview

The primary focus of the previous chapters was the introduction of sequential circuits. Although we went though a couple of major derivations, we did not present much information as to the true purpose and the subsequent power of sequential circuits. This chapter represents a more towards doing something actually useful with sequential circuits with its description of various techniques associated with both designing and analyzing sequential circuits.

The primary focus of this chapter is the Finite State Machine (FSM) analysis and design. One of the interesting features of FSMs relative to their circuit implementations is that they include major elements of both combinatorial and sequential design. This is a high-level introduction to FSMs; we'll be filling in the details and providing useful problems once you grasp the details of FSMs from a high-level point of view.

Main Chapter Topics

- **INTRODUCTION TO FINITE STATE MACHINES (FSM):** This chapter provides the basic theory behind FSMs. This introduction includes a description of the basic FSM forms and various FSM representations.
- **FSM ANALYSIS AND FSM DESIGN:** This chapter explains the basic techniques of FSM analysis and FSM design. The examples provided in this chapter provide full explanation of the low-level details regarding FSM analysis and design.
- **FSM ILLEGAL STATE RECOVERY:** This chapter describes the notion of hang states and provides techniques on how to avoid this unwanted behavior in FSM.

Why This Chapter is Important

This chapter is important because it describes the basic procedures and theories regarding the design and analysis of finite state machines.

26.2 Finite State Machines (FSMs)

The term "Finite State Machine" has many official meanings and definitions in digital-land. As you have seen previously, any circuit that has the ability to remember something (namely bits), can be regarded as having a "state". The official definition of state (relative to actual logic circuits) is the *unique configuration of information within a machine*. The term "machine" makes the term "FSM" as

generic as possible[1]. The term "finite" references the fact that the machine we're dealing with can be successfully modeled in some tractable form. Conversely, if the machine had an infinite number of states, we could not produce a model of its behavior. In the end, a semi-official, circuit-oriented definition of a FSM is this: *a circuit whose behavior can be modeled using the concept of "state" and the transition between the various states in a that circuit*.

FSMs are used in one form or another in many different technical disciplines and each discipline seems to have its own particular flavor of representing FSMs. Although FSMs in different disciplines are often implemented in many different ways, they are generally described using, for lack of a better term, the universal language of state diagrams. The state diagram is a model of a FSM that visually describes the behavior of the FSM. We developed a few state diagrams to describe the operation of a simple latch and various flip-flops in a previous chapter. Actually, a state diagram can be implemented in many different ways; we'll be implementing them using digital circuitry.

Our workings with FSMs is divided into three distinct steps: 1) a high-level overview of the concepts and associated terminology, 2) analysis and design of FSMs in circuit form and their relation to the state diagram, and 3) developing the state diagram. The first two steps are straightforward and almost mechanical in nature. What we'll hopefully be learning from these steps is a basic understanding of state diagrams and their relation to digital circuitry. The third step is where the engineering is involved. Creating a state diagram requires learning a new language of sorts: the language of state diagrams. You'll be using this language to solve various engineering problems in upcoming chapters. FSMs are amazing devices; the state diagram represents the most useful tool to understanding FSM operation[2].

Although any digital circuit that contains a memory element is officially a FSM (hence, any sequential circuit), we'll not be using this broad definition in the following discussion. At this level of digital design, we'll use FSMs primarily as *a circuit that controls other circuits*. The key word here is *control*; keep this word in mind in this chapter. The problem is that this chapter presents definitions and basic techniques of dealing with FSMs; the fact that the FSM act as a controllers can be easily forgotten in the details that follow. Try not to lose grasp of the ultimate function of FSMs; we'll start working with more interesting problems in upcoming chapters.

26.3 High-Level Modeling of Finite State Machines

Figure 26.1 and Figure 26.2 show that there are two basic types of FSMs: *Moore* and *Mealy* machines. As you can see from a brief perusing of these figures, these two types of FSMs are more similar than they are different so we'll discuss the similarities first. The terminology used to describe the three basic components of FSM differs widely from source to source but the general function of the three components is equivalent. If by chance you are lucky enough to delve deeper into the field of FSM design, you'll find that there are some variations in the functioning of these blocks. However, since this is simply an introduction, we'll stick to the basics.

Figure 26.1 shows a basic model of a Moore-type FSM. Although a FSM has lots of internal digital circuitry, we can easily abstract the functionality into three separate blocks: 1) Next State Decoder, 2) the State Registers, and 3) the Output decoder. Table 26.1 provides a detailed description of these individual blocks of Figure 26.1 and Figure 26.2; understanding the basic functioning of the blocks is the key to understanding how these blocks interact with each other (it's the system-level thing all over again) and form the FSM.

[1] Keep in mind that FSMs and the concept of FSMs are used in many other disciplines. In this course we'll deal with them primarily in the realm of digital circuitry.

[2] Once again, a state diagram is a visual tool designed to facilitate human understanding of a FSM's operation. Other FSM representations may be more appropriate for other applications such as implementing a FSM via software.

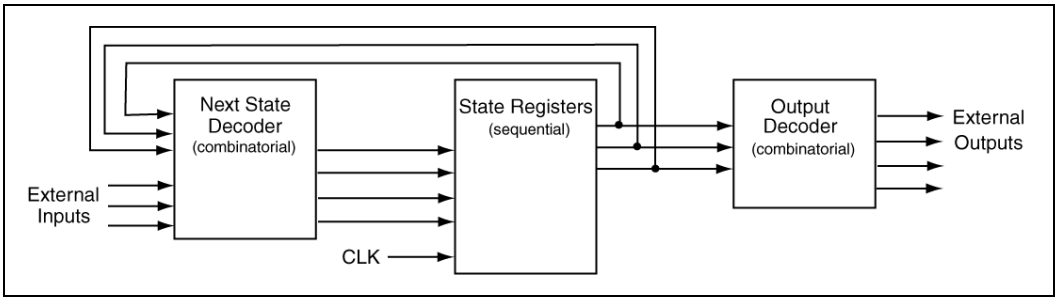


Figure 26.1: Model for a Moore-type FSM.

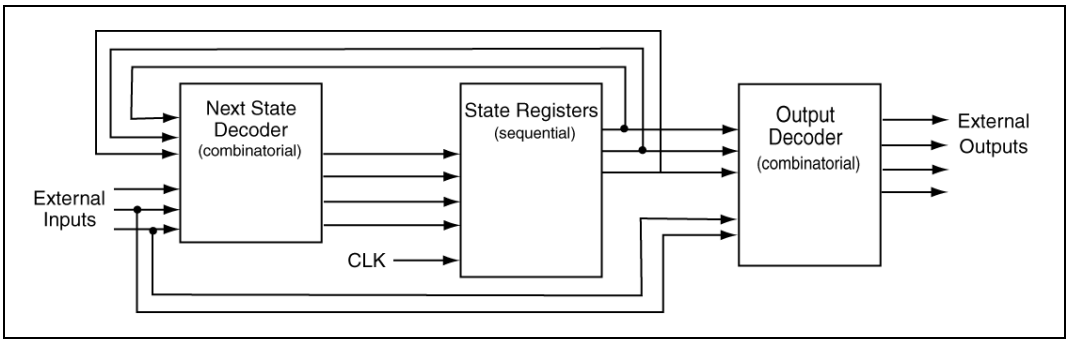


Figure 26.2: Model for a Mealy-type FSM.

Module	Description and Comments
State Registers	The state registers represent the memory elements in the FSM. The term *register* in digital-land is term that is implies that some type of synchronous storage elements are involved. In the case of the flip-flops, we'll be using flip-flops as storage elements and they can store a single bit of information. The state register is the only sequential part of the FSM; the other two blocks use combinatorial logic in their implementations. As you can see from Figure 26.1, the only synchronous portion of the FSM is the state registers. In other words, the clock signal present in Figure 26.1 only affects the state registers. The state registers store the *state variables* of the FSM. As you'll see in upcoming examples, the bits that are stored in the flip-flops in the state registers determine the state of the FSM. The purpose and function of the state registers is identical for both Mealy and Moore-type FSMs.
Next State Decoder	The next state decoder is a piece of combinatorial logic that provides excitation input logic to the flip-flops in the state registers. The next state logic generally has two types of inputs: 1) the current value of the state variables, and, 2) the current value of the inputs from the external world. These two sets of values form what is referred to as *excitation inputs* to the state register flip-flops. Recall that the inputs to the flip-flops determine the *next state* of the flip-flops (following the next active clock edge). The important thing to notice is that the next state of the flip-flops, or the next state of the FSM, is a function of both the external inputs and the current present state of the state registers. Once the active clock edge arrives, the flip-flops act on the excitation inputs and the next state becomes the current state. This type of cycling occurs every clock edge. The next state decoder is sometimes referred to as the *next state logic*, or the *next state forming logic*. Keep in mind that the internal inputs are a key feature of the FSM function: the external inputs to the next state decoder essentially function as status signals from the world outside of the FSM. If you think about this in an intuitive sense, when the FSM is controlling something, it needs to be aware of the status of what it is controlling. The FSM does this via the external inputs to the next state decoder. The purpose and function of the state registers is identical for both Mealy and Moore-type FSMs.
Output Decoder	The output decoder is a set of combinatorial logic that generates the external outputs of the FSM. The difference between a Mealy and Moore-type FSM is based solely upon the inputs to the output decoder; these differences are shown by comparing and contrasting the output decoder blocks in Figure 26.1 and Figure 26.2. While the outputs of the next state decoder module are the same for both a Mealy and Moore-type machine, the inputs have one difference. In a Mealy-type FSM, the external outputs are a function of both the state variables and the internal inputs. In a Moore-type FSM, the external outputs are strictly a function of the state variables. This difference is massively important and one that we'll be dealing with often in the discussion that follows. Having a fundamental understanding of the differences between a Mealy and Moore-type FSM is integral to understanding and designing FSM-based controllers. The external outputs from the output decoder generally serve as control signals to the device(s) controlled by the FSM.

Table 26.1: A detailed description of the three main FSM functional blocks.

You should now have somewhat of a feel for the operation of a FSM based on the diagrams of Figure 26.1 and Figure 26.2. The heart of the FSM is the state registers; the heartbeat of the FSM is the clocking signal that controls the state-to-state transitions of the FSM. On each active edge of the clock, the state of the FSM (the values stored by the flip-flops) can change. The excitation inputs to those flip-flops determine the state transitions of each flip-flop in the state registers. The excitation inputs to the flip-flops are the outputs of the next state decoder. The next state decoder outputs are formed by the logic internal to the next state decoder and are a function of the present state of the FSM and the external inputs. The external inputs are generally status signals from the outside world. The FSM sends the control signals to the outside world via the output decoder. The external outputs from the FSM are a function of the state variables (Moore-type FSM) or a function of both the state variables and the current external inputs (Mealy-type FSM). Read through this description a few times; we'll fill in the details real soon.

26.4 FSM Analysis

The best way to understand the functioning of an FSM is to examine one. Since we'll be taking a real close look at the basic operation of one FSM, we can officially refer to our viewing as *FSM analysis*. Our initial goal of this analysis is to determine how FSMs work; the secondary goal of this analysis is to be able to model the behavior of this particular example. Once we do a few analysis examples, we'll switch over to design-type examples. Our final goal of analyzing a given FSM is to generate a state diagram. Let's do it.

Example 26-1

Analyze the FSM shown in Figure 26.4. Provide a PS/NS table (including the outputs) and a state diagram that describes the circuit.

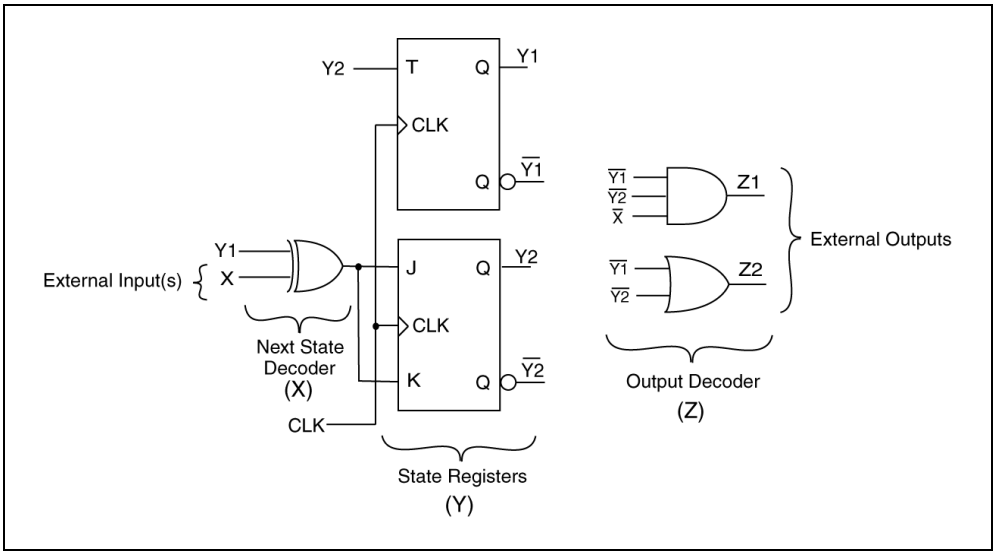


Figure 26.3: A typical Mealy-type FSM.

Solution: Figure 26.3 shows a circuit that represents a typical FSM implementation. The first step in any analysis is to stare at the diagram to get a feel for the approach you'll need to take to successfully analyze the circuit. Figure 26.3 shows a sequential circuit (the flip-flops provide it with memory) which officially makes it a FSM. More importantly, it's in a form that could be considered typical for FSM. After staring at the FSM for a while, you'll note the following stuff regarding the FSM of Figure 26.3. After we stare at it for a minute, we present a procedure for analyzing the FSM. We'll then switch over to Figure 26.4, which is a cleaner version of this example.

- The circuit is drawn with a shorthand notation. For example, the outputs of the T flip-flop are the complimented and uncomplimented values of Y1 (complementary outputs). These two outputs serve as inputs to other portions of the circuit but are not explicitly connected. For example, the Y1 output of the T flip-flop acts as an input the XOR gates on the left; the complemented Y1 output also acts as an input the AND and OR gates on the right of the diagram. The same story is true of the Y2 output. The X signal acts as an input for the XOR gate and also, once complimented, acts as an input for the AND gate on the right. The inverter that complements the X input is not shown. Once again, this shorthand notation de-clutters the circuit diagram.

- The outputs of the flip-flops, Y1 and Y2, are the *state variables* for the FSM. Since there are two state variables and since the variables are binary in nature, this FSM has four different unique states. There are two flip-flops and each flip-flop can store one bit of information. The means there are four final states for the machine: Y1Y2 = "00", "01", "10", and "11". The number of possible states relates to the number of bit-storage elements by a power of two[3]. For example, there are two flip-flops, two flip-flops raised to the power of two is four. If the FSM contained three flip-flops, there could be up to eight unique states.

- The external inputs, the state variables, and the external outputs are represented by X, Y, and Z variables, respectively. This is generally done to keep things simple, which is nice when you're first dealing with FSMs. The truth is that once you know more about how state machines generally operate, you'll change from the X and Z variables to names that are symbolic, self-commenting and thus more meaningful in nature.

- There is one external input (X) and two external outputs (Z1 and Z2) for this FSM. The Z2 output is a Moore-type output while the Z1 output is a Mealy-type output.

- The three standard functional blocks of an FSM are not readily apparent from the diagram so we explicitly list them. The XOR gate to the left of the JK flip-flop forms the Next State Decoder logic. The two storage elements (the T and JK flip-flops) form the State Registers. The AND and the OR gates form the Output Decoder logic.

- This FSM is a Mealy-type FSM. The way you know this is by examining the Output Decoder block. The presence of the X variable on the input to the AND gates essentially make the external outputs a function of the external input (remembering that the X represented external inputs). Even though only one of the external output is a function of the external input X, it is still considered a Mealy-type FSM. If the Output Decoder were not dependent upon the external input, we would then classify this FSM as a Moore-type FSM.

[3] This is true for now but will change in later chapters.

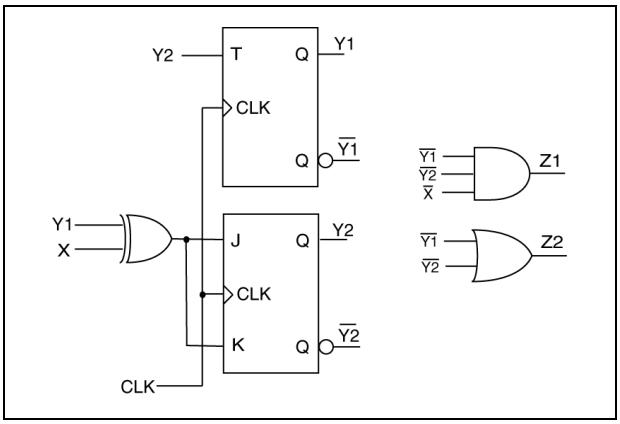


Figure 26.4: A cleaner looking version of Example 26-1.

The following list shows the basic steps in the solution to this problem in painful detail below. The idea here is to drag you through the process one time in excruciating detail and then allow you to decide upon your own personal level of detail when analyzing FSMs. The following analysis has been broken up into steps that seem to make sense to me; you need to make them make sense to you. Keep in mind that engineering is not a matter of following steps[4]; following these steps are simply an aid to your understanding of FSM analysis. You'll soon be forced to fend on your own.

Step 1) Stare at the diagram and note the important structure and features
Step 2) Write down the equations for the excitation logic
Step 3) Write down the equations for the output logic
Step 4) Generate the empty of a PS/NS table
Step 5) Provide columns in the PS/NS table for the excitation variables
Step 6) Use the excitation equations to fill in the columns representing the
Step 7) Provide columns for the next state variables
Step 8) Fill in the columns associated with the output logic
Step 9) Draw as state diagram
Step 10) Allow the celebration to begin (not really a step; it just sounds good)

Step 1) We already went through step one in deep detail. The main point of this step in general is to discern the following:

- The number of external and inputs and outputs
- Whether the FSM is a Mealy-type or Moore-type machine
- The maximum number of FSM states (based on the number of storage elements)

Step 2) Write down the equations for the excitation logic. For this circuit, the excitation logic is the logic attached to the flip-flop's synchronous inputs (the stuff connected to the T, J, and K inputs of the individual flip-flops). From examining the diagram of Figure 26.4, you can generate the equations shown in Equation 26-1.

[4] Although selectively following and/or enforcing rules is the hallmark of an academic administrator.

$$T = Y2$$
$$J = Y1 \oplus X$$
$$K = Y1 \oplus X$$

Equation 26-1

Step 3) Write down the equations for the output decoder logic. In this case, you see that there are two external outputs from the FSM: Z1 and Z2. From inspection of the circuit, you can generate the following equations. We know that Z1 is a Mealy-type output because Z1 is a function of X while Z2 is not.

$$Z1 = \overline{Y1} \cdot \overline{Y2} \cdot \overline{X}$$
$$Z2 = \overline{Y1} + \overline{Y2}$$

Equation 26-2

Step 4) Generate the initial PS/NS table which is essentially a truth table. The PS/NS table provides a listing of the present state, the variables that effect the state transition from the present state to the next state, and the next state (we'll also list the output variables at later step). In the beginnings of our PS/NS table shown Table 26.2, we've listed the present state and the external input as the independent variables in the table. At this point, the only information that matters to us is the present state and the value of the external input. Only the Y1 and Y2 variables form the present state: *the X input is not part of the present state of the FSM*. Recall from Figure 26.1 and Figure 26.2 that the only things affecting the transition from one state to another are the state variables and the external inputs. Another way of looking at the PS/NS table is that the number of inputs to the Next State Decoder determines the number of rows in the truth table. One other thing to note in Table 26.2 is that we have many extra columns; we'll fill these in during subsequent steps.

Next State Decoder Inputs									
Present State		**Ext Input**							
Y1	**Y2**	**X**							
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
1	0	1							
1	1	0							
1	1	1							

Table 26.2: After Step 4).

Step 5) Since we need to find out how the next state decoder outputs affect the flip-flops, we need to examine the excitation inputs of the flip-flops. The logic in the Next State Decoder forms the excitation logic for the flip-flops that represent the state variable. In this step, we need to provide columns in the developing PS/NS table to list the logic generated by excitation equations. There are three inputs to the two flip-flops; Table 26.3 shows that each of these inputs receives a column in the PS/NS table.

Next State Decoder Inputs			**State Register Excitation Logic**						
Present State		**Ext Input**							
Y1	**Y2**	**X**	**T**	**J**	**K**				
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
1	0	1							
1	1	0							
1	1	1							

Table 26.3: After Step 5).

Step 6) In this step, the excitation logic generated from the excitation equations of Step 2) is entered into the T, J, and K columns of the developing PS/NS table. We enter the 1's and 0's in these columns directly based on the inputs to the Next State Decoder. For example, since the excitation input equation for the T input is T = Y2, we copy the Y2 column into the T column. A better example is the logic for the J and K columns: the data in these columns represents an exclusive ORing of the Y1 and X columns from the Next State Decoder Inputs section of the PS/NS table. Table 26.4 shows the results of this step.

Next State Decoder Inputs			**State Register Excitation Logic**						
Present State		**Ext Input**							
Y1	**Y2**	**X**	**T**	**J**	**K**				
0	0	0	0	0	0				
0	0	1	0	1	1				
0	1	0	1	0	0				
0	1	1	1	1	1				
1	0	0	0	1	1				
1	0	1	0	0	0				
1	1	0	1	1	1				
1	1	1	1	0	0				

Table 26.4: After Step 6).

Step 7) Now that we've listed the Next State Decoder information and the excitation equation logic, we can generated the Next State (NS) values. Keep in mind that the next state values are the values that the

flip-flops will have after the next active clock edge. Since we know what the excitation logic is (as generated from the present state and external input values), we can generate the next state values based on that logic. We list a few examples below; make sure you understand how these examples relate to the developing PS/NS table. Table 26.5 shows the complete table for this step.

- In the first row of the truth table, the present state of Y1 is a '0'; the current input to the T flip-flop is a '0'. A '0' input on a T flip-flop will cause no change in the state of the flip-flop so the next state value will thus be a '0'.

- In the second row of the truth table, the present state of the Y2 variable is a '0'; the current inputs to the JK flip-flop are JK = "11". This is the toggle condition for the JK flip-flop and thus causes the present state input of '0' to toggle which results in a next state output of '1' for the JK flip-flop.

Next State Decoder Inputs			**State Register Excitation Logic**						
Present State (PS)		**Ext Input**				**Next State (NS)**			
Y1	**Y2**	**X**	**T**	**J**	**K**	**$Y1^+$**	**$Y2^+$**		
0	0	0	0	0	0	0	0		
0	0	1	0	1	1	0	1		
0	1	0	1	0	0	1	1		
0	1	1	1	1	1	1	0		
1	0	0	0	1	1	1	1		
1	0	1	0	0	0	1	0		
1	1	0	1	1	1	0	0		
1	1	1	1	0	0	0	1		

Table 26.5: After Step 7).

Step 8) Now that PS/NS table is complete in terms of the state transition information, we can tack on the external output logic. There are two external outputs for this FSM: Z1 and Z2. Although this FSM is officially a Mealy-type FSM, we can consider the outputs as both a Mealy output (Z1) and Moore output (Z2)[5]. The Z1 output is a Mealy output because it is a function of both the present state (PS) variables as well as the external input variables X. The Z2 output is a Moore output since it is only a function of the present state variables. We enter the logic in the Z1 and Z2 columns in Table 26.6 by examining the output equations from Step 3. For Z1, the only time the output is a '1' is when each of the three inputs (Y1, Y2, and X) are '0'. Conversely, the only time the Z2 output is a zero is when each of the state variables is '1'; this condition occurs in the final two rows of the PS/NS table. There are two massively important points that are typically sticking points when first working with FSMs.

- The outputs, both Mealy and Moore-types, are always a function of the present state variables (Y1 and Y2). When entering the logic into the output columns, make sure you don't base the output on the next state variables ($Y1^+$ & $Y2^+$). This is a common mistake; don't try it here.

- Note that the Moore output (Z2) is always the same per set of state variables (Y1 & Y2). In contrast, the Mealy output (Z1) can change for a given set of state variables.

[5] To be considered a Moore-type FSM, all of the outputs would need to be Moore-type outputs.

Note that in the first two rows of the PS/NS table, Z1 has two different outputs. This is because the Z1 output is a Mealy-type output and is a function of both the X input and the state variables. With the Moore output (Z2), since it is strictly a function of the state variables, it will not change so long as Y1 and Y2 are the same. Since Y1 and Y2 are based on the first two columns in the PS/NS table, the Moore output is effectively the same for the pairs of rows.

Table 26.6 shows that the PS/NS table is now officially complete. There is another important point about the PS/NS table: there is no clock signal listed in the table. The PS/NS table never includes the clock signal because it all state transitions occur on the active clock edge. In other words, the only time the values on the state registers can change is on the active clock edge. The clock signal could be included in the PS/NS table but it would clutter an already cluttered table and provide no information.

And speaking of a cluttered PS/NS table… there is a better way to represent the operation of a given FSM. While the PS/NS table does in fact provide all the necessary information to describe the operation of the FSM, it provides the information in an un-friendly manner. A better approach is to use the information in the PS/NS table to generate a state diagram. This is the final step in the analysis process.

Next State Decoder Inputs			**State Register Excitation Logic**					**Output Decoder Outputs**	
Present State (PS)		**Ext Input**				**Next State (NS)**		**Mealy**	**Moore**
Y1	**Y2**	**X**	**T**	**J**	**K**	**$Y1^+$**	**$Y2^+$**	**Z1**	**Z2**
0	0	0	0	0	0	0	0	1	1
0	0	1	0	1	1	0	1	0	1
0	1	0	1	0	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	1	1	0	1
1	0	1	0	0	0	1	0	0	1
1	1	0	1	1	1	0	0	0	0
1	1	1	1	0	0	0	1	0	0

Table 26.6: After Step 8).

Step 9) The final step is to draw the state diagram associated with the PS/NS table. The state diagram is simply a visual representation of the information provided by the PS/NS table. There are many approaches to drawing a state diagram; the most important item is to provide a legend to allow the human viewer to know what is going on. We decompose the drawing of the state diagram into a bunch of humongously boring steps in a fashion similar to the development of the PS/NS table. Once you do a few of these problems, you'll not need to follow the steps; it becomes second nature because you'll be loving it as well as understanding it. One last thing to note is that the order of the steps in drawing the state diagram are somewhat arbitrary; it can actually be done in many different ways.

- **Step 9(a):** Draw a bubble representing each possible state in the FSM. Since this FSM has two state variables (Y1 & Y2) based on two flip-flops, there will be four states in the state diagram. The four states are come from the notion that each of the flip-flops can store one bit; thus there are four unique combinations of the two state variables. The PS/NS table reflects this condition. Another important task to perform

in this step is creating a legend to describe some of the pertinent information in your state diagram. Note that the legend is located on the left.

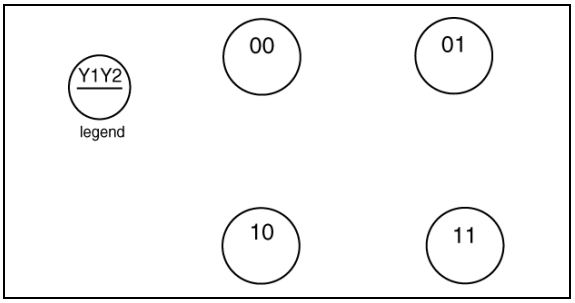


Figure 26.5: The results of Step 9(a).

- **Step 9(b):** Enter the Moore-type output information into the state diagram. Recall that a Moore-type output is only a function of the present state of the FSM. In other words, the Moore output variable only changes when the state changes which allows the Moore-type output to be included as part of the state bubbles. We do this by dividing the state bubble into state information and output information as indicated on the right. The Z2 information is in the Z2 column of the PS/NS table. Note that because Z2 is a Moore-type output, changes in the external input variable X do not affect the Z2 output. This forces the Z2 output to follow pairs of rows in the PS/NS table. For example, the first two rows in the PS/NS table have the exact same state variables. Note here that the outputs of the FSM are always based on the present state of the FSM, and never the next state. Also note that the legend has been updated to reflect the fact that the Z2 output is indicated in the state bubbles. Figure 26.7 shows the results of this step.

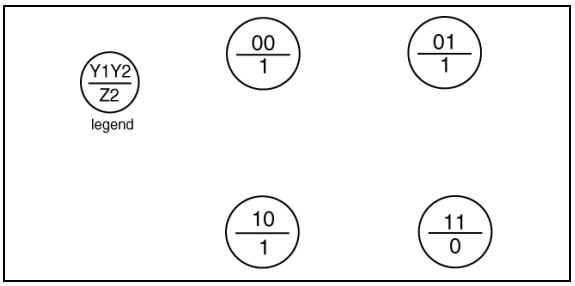


Figure 26.6: The results of Step 9(b).

- **Step 9(c):** Enter the state transition information. There are several important tasks associated with this step. First note from the PS/NS table that all state transitions are a function of both the present state variables and the external X input. Since there is only one external input, there can be at most two different transition possibilities in the associated state diagram: these two possibilities are associated with the two possible state of X: '0' and '1'. We represent the state transitions by drawing singly directed arrows; in the context of the PS/NS table, the arrows originate in the current state and end in the next state. The arrows reflect the conditions in the individual rows in the PS/NS table. For example, in the first row of the PS/NS table, the present state variables are "00"; if the X input is a '0' when the active edge of the clock arrives, the next state of the FSM will be "00". Also associated with this transition is the value

of the Z1 output. Since Z1 is a Mealy-type output, it is a function of the external X input, and thus can change based on changes in the X input. For this FSM (as indicated by the first row of the PS/NS table), when the X input is '0' in state "00", the Z1 output is '1'. Similarly, when the X input is '1' in state "00", the Z1 output is '0'. Since the output decoder is combinatorial, the Z1 output has an immediate response to the X input and is not associated with the clock edge. The value of the X input on the active edge of the clock determines the state transitions. These two conditions are somewhat confusing based on the way the state diagram indicates these conditions. The problem is that there is no easy way to reflect both the Mealy-type output and the state transitions in the state diagram. Figure 26.7 shows the standard approach. Note that in this approach, the X input and Z1 output are associated with the state transition arrow, which is somewhat misleading. The important thing to remember here is that the output drawn with the state transition arrows is associated with the state that the arrow is emanating from and not the state that the arrow is entering. We'll deal with this in more detail in an upcoming chapter.

- Be sure to note that the first row in the truth table forms what we call a self-loop in the state diagram. This condition indicates that the FSM does not change state on the next active clock edge. Finally, be sure to note that the state diagram clearly indicates which values are X and Z variables; you must do this in order for someone to understand what the state diagram is really doing.

- You interpret the first two arrows drawn in the diagram. For the first left-most arrow of Figure 26.7, the FSM will not change state on the active clock edge if the X input is '0'. Also, then the X input is '0' in the Y1Y2 = "00" state, the Z1 output is a '1'. Similarly, for the other arrow in Figure 26.7, if the X input is a '1' in the Y1Y2 = "00" state the Z1 output is a '0'; if an active clock edge occurs while X = '1', the FSM will transition to the Y1Y2 = "01" state.

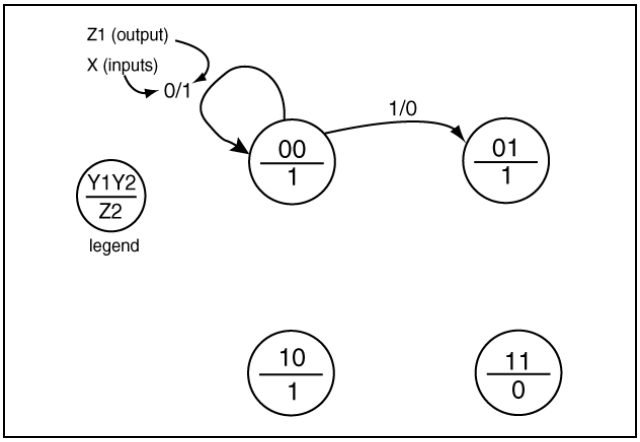


Figure 26.7: The results of Step 9(c).

- **Step 9(d):** This step is not really a step; what you need to do from this point is continue to transfer information from the PS/NS table into the state diagram. Figure 26.8 shows the results of entering the third and fourth row from the PS/NS table into the state diagram. Figure 26.9 shows the complete state diagram for this example.

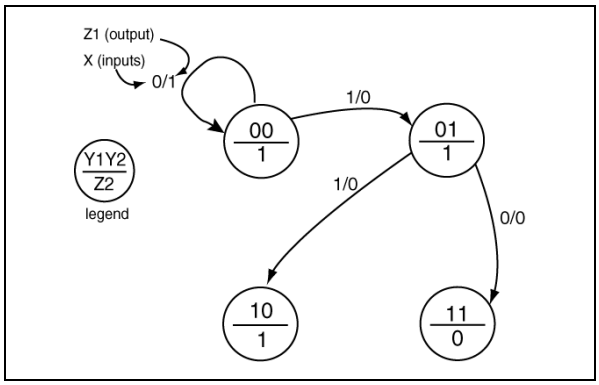


Figure 26.8: The results of Step 9(d).

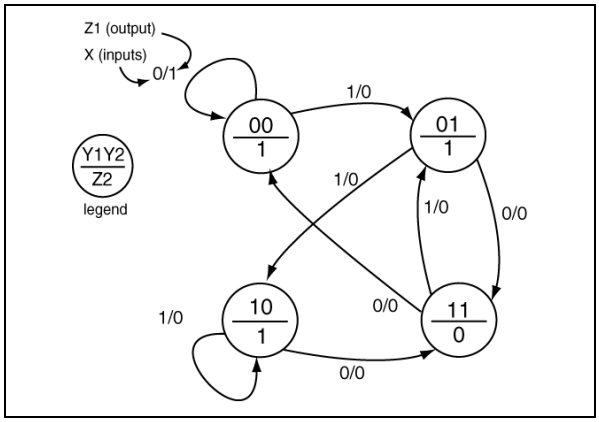


Figure 26.9: The completed state diagram.

Here's a quick summary of this example.

- A legend and legend-type information is *always* included with the state diagram.
- The state diagram lists each possible state in the state diagram by using a bubble; different combinations of state variables represent the states. Since there are two state variables and each state variable can be either a '0' or a '1', there are four possible states.
- State transitions are represented using arrows; transitions can occur from one state to that same state or to any other state in the state diagram.
- The external input information that causes the state transitions are listed next to the state transition arrows.
- The Moore-type outputs are listed inside of the state bubbles since they are a function of the present state only.

- The Mealy-type outputs are listed next to the external inputs controlling the state-to-state transitions arrow. Note that the Mealy-type outputs are associated with the state that the arrows are emanating from as opposed to the state to where they are going.

One thing that is glaringly missing from the state diagram and the PS/NS table is the system clock. Since the FSM uses flip-flops as the storage elements, state transitions can only occur on the active clock edge. Since this information is inherent to the operation of the FSM, it is never included in either the PS/NS table of the state diagram. To put this in another way, both the PS/NS table and the state diagram list the state-to-state transitions of the FSM. Since these transitions only occur on the active clock edge that controls the flip-flops, explicitly showing this information in the PS/NS table or state diagram would only serve to clutter them. There probably is a way to include the clock in one of these representations but I'm not sure what it is and I never seen it listed anywhere.

The style that is used drawing the state diagram is not unique; there are actually many different ways to draw them. This example uses an approach that is probably the clearest. As you become more fluent with drawing state diagrams, you'll probably alter your own approach to drawing them; be sure to indicate you own particular style somewhere in the state diagram annotation and/or legend. The key to using a different style to drawing state diagrams is making sure you explicitly state your approach with the legend provided with the state diagram. A state diagram without legend is useless. If you are clear with your state diagram style, there will be much joy and the world will be happy.

In the big scheme of things, recall that you have just analyzed a sequential circuit. You characterized the operation of this circuit by generating a state diagram. You know that a sequential circuit has memory and that the outputs of a sequential circuit are a function of the "sequence" of inputs to the circuit. The state diagram explicitly shows this so-called sequence.

One of the most important factors in developing a working knowledge of FSMs is understanding how the state diagram relates to some of the timing aspects of the circuit. We'll be spending more time on this subject later but we'll take an introductory look at it for this example. Figure 26.10 shows a sample timing diagram for this example. Note that this is only a sample; we provide it in complete form, which somewhat obscures the starting point of the state diagram.

The starting point of this state diagram is the state in sometime before the first clock edge. The first state in this timing diagram was provided for us (it had to be otherwise we would not know what it was). This timing diagram also provides the entire X input. From the initial state and the X input information, we can complete the timing diagram as shown in Figure 26.10. Note there are three forms of information that we need to include in this timing diagram: 1) state information (how the states change on the active clock edges), 2) Z1 output information, and 3) Z2 output information. Here are a few things to note about this timing diagram.

- The Z2 output is Moore-type output so it is only a function of the state variables. This means it can only change when the state changes. This condition manifests itself in the Z2 output by having changes in the Z2 output *always* synchronized with the clock edge.

- The Z1 output is Mealy-type output so it can change in between active clock edges. As you can see from Figure 26.10, the Z1 output occasionally changes between clock edges as listed in the state diagram.

- The X input and the current values of the state variables control the state-to-state transitions. The only time we consider the X input is for the state transitions is on the active clock edge.

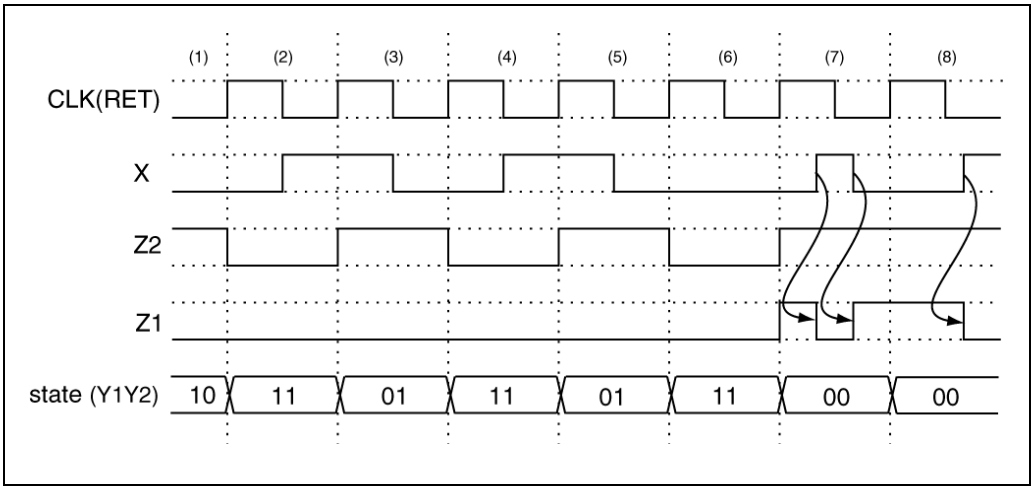


Figure 26.10: Example timing diagram associated with the state diagram of Figure 26.9.

The timing diagram was completed by examining the value of the X input and state variables in at each clock edge in the timing diagram and using this information in conjunction with the state diagram to glean where the next state value after the active clock edge. Here is a description of a few of the time slots in Figure 26.10.

- Time Slot (1): In this time slot, the state variables are "10"; the X input value at the next active clock edge is '0'. Cross referencing this information into the state diagram you see that from state "10", if X is a '0', then the FSM will transition to the "11" state. Also in state "10", the Z1 output is a '1', which is indicated by the number under the line in the "10" bubble. The Z2 output is a '0' independent of the value of X during Time Slot (1); this condition is noted by the fact that both arrows leaving the "10" state bubble have Z2 outputs of '0'.

- Time Slot (7): In this time slot, the Z1 output, which is the Mealy-type output, is changing between the active clock edges. This is once again because the Z1 output is a function of the X input. In particular, as you can see from the state diagram, the Z1 output in the "00" state is the opposite value of the X input. These conditions area listed next to the state transition arrows. Keep in mind that the Z1 output is the same value for the other three states in the state diagram, which is why you don't see the Z1 output changing in-between the active clock edges as see in this time slow. This is a massively important attribute of the Mealy input; we'll be spending much more time with this later.

26.5 FSM Design

Designing counters using FSMs is one of the more basic FSM design exercises. For this reason, counter-design provides a good introduction to the subject of FSM design. All of the important steps are here but as you will see later, the specification step is greatly simplified. In this context, the specification primarily is the count you'll need to implement. A spec such as a simple counter is more straightforward than designing a FSM that acts as a controller. Once you become fluent at designing simple FSMs such as counters, we'll move onto designing FSMs that are used to control things that need controlling.

The first step in FSM design is defining the state diagram. Once the state diagram is specified, the final generation of the FSM is pretty much cookbook no matter if you're implementing it with actual gates or with a VHDL model. As mentioned earlier, the design of FSMs implies that you're taking a specification and generating a circuit.

Example 26-2

Design a counter that counts in the following sequence: 0, 2, 3, 1, 0, 2… . Use one T and one JK flip-flop for each of the state variables. Show a PS/NS table and state diagram that describes the FSM. Provide flip-flop excitation equations in reduced form and draw the final circuit.

Solution: This is a counter design problem; two characteristics make this design straightforward. First, we implement the desired count as Moore-type outputs of the FSM. This simplifies the problem in that the FSM does not need to have an output decoder. In other words, the desired count is the direct output of the flip-flops. In yet other words, the state variables directly represent the count; the external outputs from this FSM are simply the state variables. Secondly, there are no external input variables. In this way, all the state transitions are unconditional which simplifies the entire process. As you'll see in the upcoming design, there are simply less steps in the process. Here are the basic design steps:

Step 1) Stare at the problem
Step 2) Generate the states in the state diagram
Step 3) Generate the state transitions for the state diagram
Step 4) Generate the initial PS/NS table
Step 5) Enter the next state information into the PS/NS table
Step 6) Generate the excitation logic for the flip-flops
Step 7) Generate the excitation equations for the flip-flops
Step 8) Draw the final circuit
Step 9) Commence celebration: (not a required step)

Step 0) This isn't really a step but it sure is a great idea. Before we go on with this problem, let's draw a timing diagram that would represent the output of this problem. Figure 26.11 shows such a timing diagram. Notice in Figure 26.11 that the FSM arbitrarily choose the outputs to be Y1 and Y2. For this case, Y1 is the MSB as judged by the order of appearance. The initial values of the Y1Y2 outputs have no significance. There are two major items of significance in this drawing. First, the Y1Y2 row does indeed show the desired sequence listed in binary. In addition, the changes in the Y1Y2 sequence are synchronized to the rising clock edge. Once again, the problem did not mention anything regarding active clock edges so we are therefore free to choose either rising or falling clock edge.

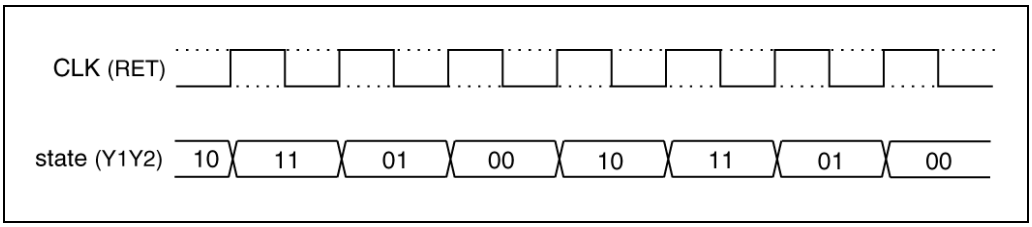


Figure 26.11: Example timing diagram for Example 26-2.

Step 1) Stare at the problem. From this wanton gaze you'll be able to gather the following high level information.

- Looking at required sequence, you can see that there are only four unique numbers in the sequence. The sequence then repeats itself after encountering each of the four numbers. Since there are four states, you'll be able to implement this design using two flip-flops (as is implied by the problem statement). If there were five numbers in the required sequence, you would have needed three flip-flops to implement the circuit.

- The problem provides the numbers in the required sequence in decimal form. The implication here is that the actual implementation is in binary since the outputs of the flip-flops (where the state variables are stored) hold the count values. Sometimes this will not be the case but since this is a simple counter, it is the case for this problem.

- After you stare at this problem for a while, you realize that drawing block box diagram for this FSM is a good idea. Figure 26.12 shows such a block box diagram. For this diagram, the choice of Y2 and Y1 are arbitrary but these choices match the timing diagram musing of Figure 26.11.

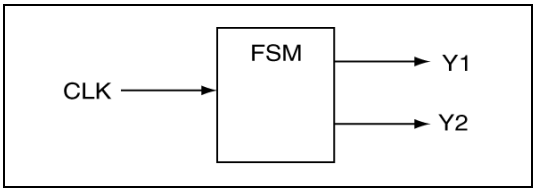


Figure 26.12: Black box diagram for the FSM of Example 26-2.

Step 2) Generate the states in the state diagram. Figure 26.13 shows the initial state diagram along with a legend. Note that the solution chose variable names of Y1 and Y2 for the state variables (which are arbitrary). The diagram represents all four states in the count sequence. The diagram also provides the decimal equivalents of the required binary count below the state variable declarations for each state.

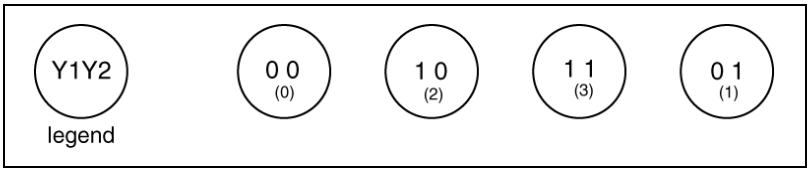


Figure 26.13: Initial state diagram for Example 26-2.

Step 3) Generate the state transitions for the state diagram. For this counter, the state transitions occur unconditionally on each active clock edge of the flip-flops. For this step, we only need to include the state transition arrows in the state diagram; there are no conditions to associate with these arrows. Figure 26.14 shows the resulting state diagram.

Once you complete this step, the state diagram is officially complete. From this state diagram, you can choose to implement the actual FSM using many different approaches. As you will see in the remainder of this example, none of these approaches is overly complicated. As you get used to the FSM implementation procedures, you see that they become cookbook. In reality, the only complicated step is in the design of the state diagram: this is where the engineering and deep thought is required. This is a simple example and we're purposely not attempting to gather an all encompassing understand of FSM (we'll be doing that soon though).

State diagram design requires a mindset all its own; once you grasp the intricacies of state diagrams (and there's not that many of them), you'll be able to design many powerful circuits. As you'll no doubt agree after finishing this example, once the state diagram is generated, it becomes no big deal to implement the FSM using a circuit model.

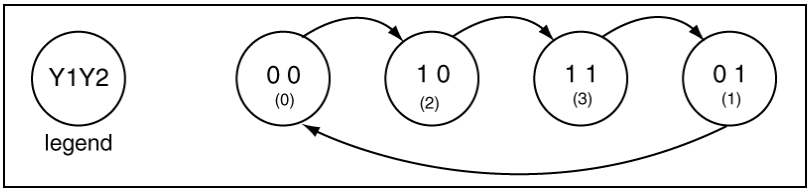


Figure 26.14: State diagram for the example problem.

Step 4) Generate the initial PS/NS table. The initial PS/NS table will contain only the present and next state flavors of the state variables. Since there are two state variables (two flip-flops), the PS/NS table only contains four rows. Figure 26.15 shows the resulting PS/NS table. Note that this table represents all possible combinations of the state variables Y1 and Y2 in the columns under the "PS" label.

(PS)		(NS)				
Y1	**Y2**	**$Y1^+$**	**$Y2^+$**			
0	0					
0	1					
1	0					
1	1					

Figure 26.15: The initial PS/NS table for the example.

Step 5) Enter the next state information into the PS/NS table. The state diagram of Figure 26.14 provides all the next state information. For example, the first of the PS/NS table is associated with the Y1Y2 = "00" state of the FSM (present state). From looking at the state diagram of Figure 26.14, the

state variables associated with the next state are "10". This "10" represents the state that the FSM will transition to on the next active clock edge. This state transition represents the 0 → 2 count in the desired sequence. We complete the remainder to of the PS/NS table by transferring the information from the state diagram to the PS/NS table. Figure 26.16 shows the resulting PS/NS table.

(PS)		(NS)				
Y1	**Y2**	**$Y1^+$**	**$Y2^+$**			
0	0	1	0			
0	1	0	0			
1	0	1	1			
1	1	0	1			

Figure 26.16: PS/NS table for the example.

Step 6) Generate the excitation logic for the flip-flops used in the FSM implementation. Although the PS/NS table is complete as shown in Figure 26.16, it is typical to include some extra columns with the PS/NS table in order to aid in the implementation of the actual circuit. Since we'll be implementing this FSM with a T and a JK flip-flop, we need to generate the excitation logic for these devices. The excitation logic describes the inputs to the flip-flops; in terms of the FSM, the excitation logic forms the Next State Decoder. This excitation logic forces the flip-flops to output the desired counts in the specified order.

Since the PS/NS table already lists the desired state transitions for each of the two state variables, we need to be able to configure the T and JK flip-flops such that these transitions actually occur. While the state transition information is somewhat generic, we need to make it specific in the context of the T and JK flip-flops. Our main tool for this operation is the excitation tables associated with each flip-flop. Figure 26.17 shows the excitation tables for the D, T, and JK flip-flops, which we derived previously. These tables are vastly important; you should know this information by heart (even if it requires that you memorize it).

D Flip-flop

Q	**Q^+**	**D**
0	0	0
0	1	1
1	0	0
1	1	1

(a)

T Flip-flop

Q	**Q^+**	**T**
0	0	0
0	1	1
1	0	1
1	1	0

(b)

JK Flip-flop

Q	**Q^+**	**J**	**K**
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

(c)

Figure 26.17: Excitation tables for the D, T, and JK flip-flops.

The excitation tables show what input values (D, T, and JK) that will cause the listed change ($Q \rightarrow Q^+$) in state for a given flip-flop. For our example, the PS/NS table in Figure 26.16 lists the changes in state variables (Y1 and Y2) that are required for this problem. We need to use the excitation tables to make

these changes happen in our flip-flops. More specifically, the PS/NS table lists the present-state and next-state for each of the state variables ($Y1 \rightarrow Y1^+$ and $Y2 \rightarrow Y2^+$). Even more specifically, the column labeled "T" actuates the state change in the Y1 state variable for each associated row; columns labeled J and K actuate the state change in the Y2 variable for each associated row. We've somewhat arbitrarily decided to use a T FF and a JK FF for the state variables Y1 and Y2, respectively. The T and JK flip-flops are slightly more challenging than using a D flip-flop but we'll redo this problem using D flip-flops after we complete this example.

For example, in the first row of the PS/NS table, the $Y1 \rightarrow Y1^+$ transition is a "$0 \rightarrow 1$" transition. To obtain this transition on a T flip-flop, you examine the excitation table for T flip-flop shown in Figure 26.17(b) and search for the $(Q \rightarrow Q^+) = (0 \rightarrow 1)$ transition. You'll find this transition in the second row of Figure 26.17(b); the value of T that causes this transition is located in the T column of Figure 26.17(b). This value is a '1'; we then enter it into the T column in the PS/NS table for this problem.

We use a similar procedure for entering the JK values in the first row of the PS/NS table. Since the $Y2 \rightarrow Y2^+$ transition is $0 \rightarrow 0$, the J and K values that force this transition are read from the JK excitation table of Figure 26.17(c). From this excitation table, a $0 \rightarrow 0$ transition occurs when the JK inputs are values "0-"; this information is then entered into the JK columns of the first row of the PS/NS table for this example. The remainder of the table in filled in accordingly. Figure 26.18 shows the PS/NS table containing all the T and JK logic.

(PS)		(NS)				
Y1	**Y2**	**$Y1^+$**	**$Y2^+$**	**T**	**J**	**K**
0	0	1	0	1	0	-
0	1	0	0	0	-	1
1	0	1	1	0	1	-
1	1	0	1	1	-	0

Figure 26.18: The PS/NS table with logic for the T and JK flip-flops.

Step 7) Generate the excitation equations for the flip-flops. For this step, you'll need to generate equations for the T, J, K flip flop inputs. The T, J, and K columns of Figure 26.18 lists the logic that is required for these equations. From this point, you see that the T, J, and K columns of the table represent functions; this is good because you're currently an expert at implementing functions using useful tools such as K-maps. From here on out, this problem is a matter of using some of your previous skills to finish the problem. You'll of course want to reduce the equations since you'll be provided a circuit diagram that implements this example. In this case, we use the K-map to generate a Boolean expression for the data in each of the T, J, and K columns. Figure 26.19 lists the final excitation equations for this example. The K-maps are not shown so as not to bore you to death any more than you're already facing death by boring pointless problems. You are encouraged to generate these equations for yourself or trying to get some other sucker to do it for you[6]. The good news here is that we can generate all of these equations by inspection of the T, J, and K logic shown in Figure 26.18. This is generally the case for two variable K-maps.

[6] If you successfully get someone else to do your work for you, you'll have the one and only skill necessary to be an academic administrator.

$T = \overline{Y1 \oplus Y2}$	$J = Y1$	$K = \overline{Y1}$

Figure 26.19: The excitation equations for this example.

Step 8) Draw the final circuit: This is generally a waste of time because if you've gotten this far, then you obviously know what you're doing. But… the final circuit is shown in Figure 26.20 just in case you're interested. You should take a minute to get a feel for this circuit and how it operates. A quick analysis of the circuit shown in Figure 26.20 cries out the following:

- There are no external inputs to the circuit.
- The external outputs in this case are the state variables. This also means there is no need for an output decoder (see Figure 26.21). This FSM is a Moore-type FSM because the outputs are only a function of the state variables (in this case, they are the state variables).
- The next-state decoder is a single exclusive-NOR gate.
- The problem never stated which edge of the clock was active; RET flip-flops were arbitrarily used.
- The active clock edge arrives; at this point, the T, J, and K inputs to the flip-flops become meaningful and cause state transitions (from the present-state to the next-state). The outputs of the flip-flops change and the process continues ad-nausuem.

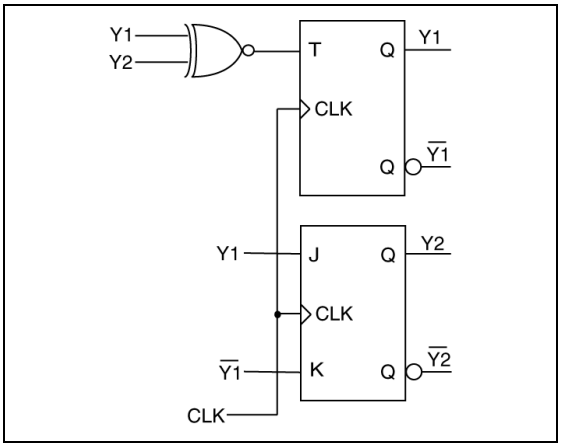


Figure 26.20: The final circuit for the example problem.

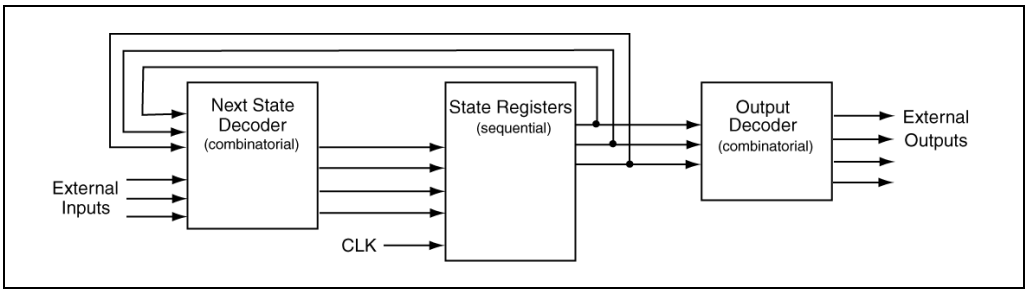


Figure 26.21: The relisted model for a Moore-type FSM.

Example 26-3

Redo the previous example but use two D flip-flops in your design in place of the T and JK flip-flops. For the previous problem, you were asked to design a counter that counts in the following sequence: 0, 2, 3, 1, 0, 2…

Solution: The nice thing about his example is that you can reuse most everything from the previous example. The state diagram is the same for both of these examples because the count sequence and the number of flip-flops required to implement this design are independent of the type of flip-flops used in the design. This being the case, Figure 26.22 shows the state diagram for this example.

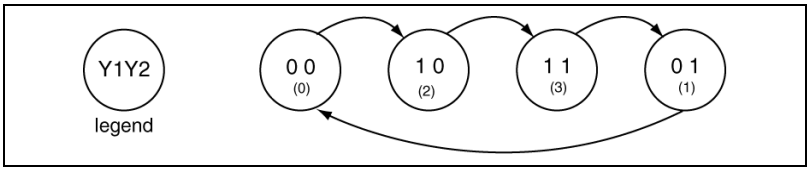


Figure 26.22: State diagram reused for this example problem.

Since the state diagram is the same for both examples the PS/NS table necessarily is the same also. Figure 26.23(a) shows the basic PS/NS table. We've modified the PS/NS table to include the columns for the D flip-flop logic as shown in Figure 26.23(b). The D1 and D2 flip-flops implement the Y1 and Y2 state variables. Since we are implementing this FSM using D flip-flops, we use the excitation table shown in Figure 26.17(a) for the D1 and D2 flip-flops. The nice thing about using D flip-flops is the fact that the NS columns represent the excitation logic for the D flip-flops. You can note this by comparing the D1 and $Y1^+$ columns in the PS/NS table shown in Figure 26.23(b). You can generate the D1 and D2 excitation logic by inspection of Figure 26.23(b); Figure 26.24 shows this logic.

(PS) (NS)

Y1	Y2	$Y1^+$	$Y2^+$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	1

(a)

(PS) (NS)

Y1	Y2	$Y1^+$	$Y2^+$	D1	D2
0	0	1	0	1	0
0	1	0	0	0	0
1	0	1	1	1	1
1	1	0	1	0	1

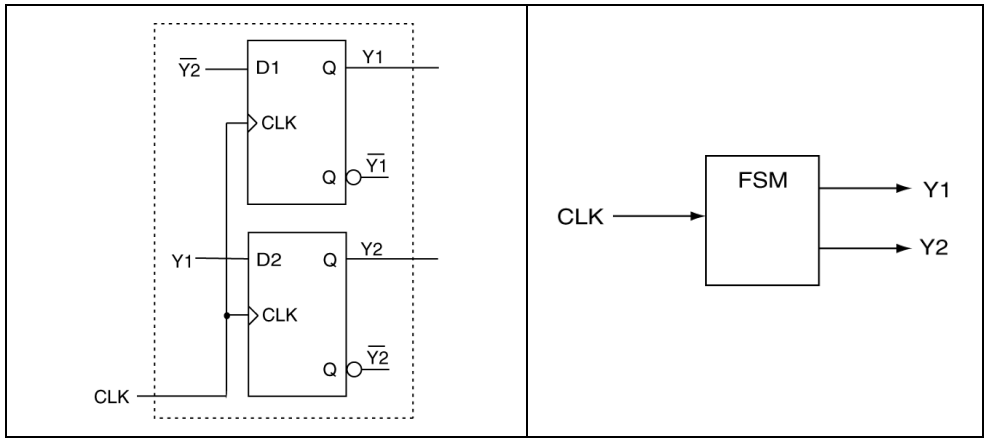
(b)

Figure 26.23: The PS/NS table for Example 26-3.

$D1 = \overline{Y2}$	$D2 = Y1$

Figure 26.24: The excitation equations for Example 26-3.

In addition, as the final act of kindness, Figure 26.25(a) shows the circuit implementation for this example. The dotted lines are included to impress upon you the fact that all the outside world needs to know of this circuit is the CLK input and Y1Y2 outputs. Figure 26.25(b) shows the associated block diagram for this FSM.



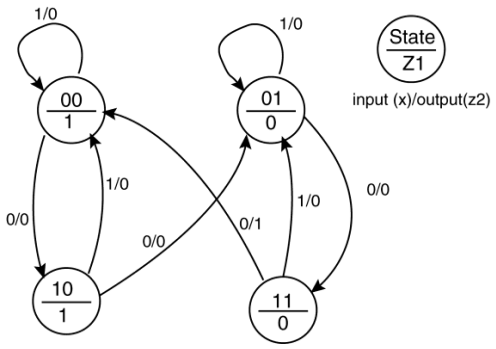
(a) (b)

Figure 26.25: The PS/NS table for this example.

Often time in make-believe digital-land, you'll be given a state diagram and be asked to implement the state diagram using a specific set of flip-flops. Not like this happens too much, but in case this situation did come up, it would be helpful to you if you knew how to proceed when the state diagram is the starting point of your problem as it is in the next example.

Example 26-4

Draw a circuit that implements the following state diagram. Use one T flip-flop and one D flip-flop in your design. Minimize the amount of required combinatorial logic used in your design.



Solution: This type of problem is essentially an undoing of the previous flavor of design problem. If you can recall from a few pages ago, we used the PS/NS table in order to generate the excitation logic for the required flip-flops. This leads us to the solution in this particular problem: we must generate the PS/NS table. This is not overly complicated in that you have previously used the PS/NS table to generate the state diagram; this problem takes the opposite approach. This approach underscores that fact that the PS/NS table and the state diagram contain the same information: they are modeling the same thing. Here is an outline of steps required to solve this problem:

Step 1) Stare at the problem for while
Step 2) Generate the initial PS/NS table
Step 3) Generate the next state logic
Step 4) Generate the excitation logic
Step 5) Generate the output logic
Step 6) Draw the final circuit

Step 1) Stare at the problem for while. This should not be a glazed over stare; this should be a visual analysis of the information provided to you by the state diagram. Here is a list of some of the items you should be looking for in this step:

- The state diagram contains four states as indicated by those funny little circle jobbers. What this should indicate to you is that the circuit that implements this FSM will require two flip-flops. Note the state diagram lists every possible combination of the two flip-flops states.

- The state diagram includes a legend which is the key to understanding all the funky terminology listed in the state diagram.

- The FSM has one external input: X. The FSM also has two external outputs: Z1 & Z2. Since the state bubbles include the Z1 output, the Z1 output must be Moore-type output. The Z2 output is listed with the X input with the state transition arrows. The fact that the Z2 output has different values as associated with the arrows exiting the "11" state indicates that the Z2 output is a Mealy-type output. In other words, the Z2 output is different in state "11" which indicates that Z2 is a function of the X input; by definition, the Z2 output is a Mealy-type.

Step 2) Generate the initial PS/NS table. The important part of this step is to figure out how many rows are required in the associated PS/NS table. For this example, the independent variables are the inputs that directly affect the circuit. For this FSM, that would include both the present state of the state variables and the external input variable X. Since there are two flip-flops required, there are going to be two present state variables. This gives a grand total of three inputs and thus eight rows in the truth table. Figure 26.26 shows the initial PS/NS table. We'll deal with the extra columns included in this table in later steps.

Y1	Y2	X				
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Figure 26.26: The initial PS/NS table for Example 26-4.

Step 3) Generate the next state logic. For this example, since there are two state variables, we'll need to represent the excitation logic with two excitation equations. This subsequently requires two columns in the PS/NS table. We take the data in these columns directly from the state diagram. For example, as read directly from the state diagram, if the FSM is in the "00" state and the X in put is a '0' when the next active clock edge arrives, the FSM transitions to the "10" state. The "10" is then entered in to the table under the $Y1^+$ and $Y2^+$ headings. Similarly, if the FSM is in state "00" and the X input is a '1', the next state will be "00". We complete the entire table by using this approach to transfer information from the state diagram to the PS/NS table. Figure 26.27 shows the results of this step.

(PS)			(NS)			
Y1	Y2	X	$Y1^+$	$Y2^+$		
0	0	0	1	0		
0	0	1	0	0		
0	1	0	1	1		
0	1	1	0	1		
1	0	0	0	1		
1	0	1	0	0		
1	1	0	0	0		
1	1	1	0	1		

Figure 26.27: The continued PS/NS table for Example 26-4.

Step 4) Generate the excitation logic. This step is dependent upon the type of flip-flops specified in the design. For this step, apply the excitation tables for the flip-flops specified in the design to translate the

$Yx \rightarrow Yx^+$ state transition information to excitation logic for the flip-flop inputs. For this design, we'll use a D flip-flop for Y1 and a T flip-flop for Y2. As stated previously, the excitation logic for the D flip-flop is identical to the next state logic. For the T flip-flop, state changes in the $Y2 \rightarrow Y2^+$ are represented by entering '1' in the corresponding rows where the Y2 variable changed state. Figure 26.28 shows the resulting PS/NS table.

(PS)			(NS)		Excitation logic	
Y1	**Y2**	**X**	**$Y1^+$**	**$Y2^+$**	**D1**	**T2**
0	0	0	1	0	1	0
0	0	1	0	0	0	0
0	1	0	1	1	1	0
0	1	1	0	1	0	0
1	0	0	0	1	0	0
1	0	1	0	0	0	1
1	1	0	0	0	0	0
1	1	1	0	1	0	1

Figure 26.28: The PS/NS table with including the excitation logic for Example 26-4.

Step 5) Generate the output logic. This FSM has two outputs: Z1 (Moore-type) and Z2 (Mealy-type). Although the logic for these outputs is typically not part of the PS/NS table, it sure is handy to include it in the table. From the state diagram, you can see that the Z1 input is a '1' when in states "00" and "10". Figure 26.29 shows that we enter this information directly into the table in the Z1 column. Keep in mind that the output is always based on the present state variables as listed in the PS/NS table. The next state columns represent the state the machine enter on the next active clock edge. It's a common mistake to base the output logic on the next state logic; but please don't let this happen to you.

Note that since Z1 is a Moore-type output, the output values always appear in pairs in the PS/NS table. The fact that they come in pairs is a result of our choice of placing X as the least significant of the independent variables. The Z2 output is a Mealy-type output, which means its value can change while in a given state (between the active clock edges). Note that in the original state diagram, the only time the Z2 output is a '1' is in the "11" state under the condition that X equals '0'. This results in the only '1' being in the Z2 column of the output logic is located in the Y1Y2X = "110" row. Figure 26.29 shows the complete PS/NS table.

(PS)			(NS)		Excitation Logic		Output Logic	
Y1	**Y2**	**X**	**$Y1^+$**	**$Y2^+$**	**D1**	**T2**	**Z1**	**Z2**
0	0	0	1	0	1	0	1	0
0	0	1	0	0	0	0	1	0
0	1	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1	0
1	1	0	0	0	0	0	0	1
1	1	1	0	1	0	1	0	0

Figure 26.29: The PS/NS table with including the excitation logic for Example 26-4.

Step 6) Draw the final circuit. Figure 26.30 shows the final circuit. We obtain the excitation logic and output logic by dropping the D1 and T2 columns into K-maps; we didn't include these details so as to spare you a slow death from this weapons grade boredom. One thing worthy to note in the circuit diagram of Figure 26.30 is that the Z1 output uses a buffer circuit element. The buffer is a circuit element that does not alter the logic levels of the signal it processes. For this signal, the complemented Y2 signal is equal to the Z1 signal. Note that our Z2 output logic does in fact contain an X on the input, which is what we would expect since Z2 is a Mealy-type output. On the other hand, the Z1 output is not a function of the X input, which is what we'd expect from a Moore-type output.

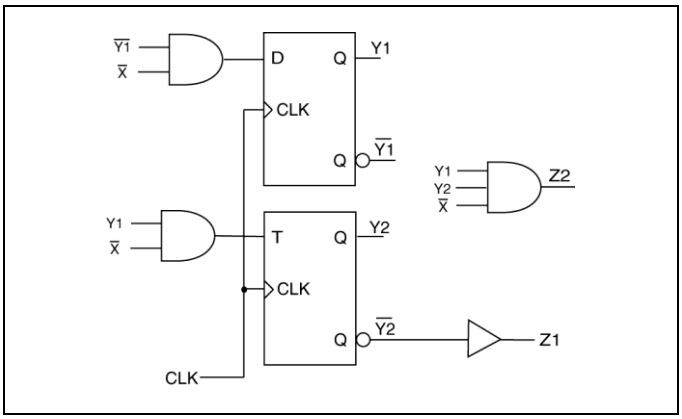


Figure 26.30: The PS/NS table with including the excitation logic for Example 26-4.

26.6 FSM Illegal State Recovery

The state machines we've examined at this point have had a certain quality that is not always present in all FSM design. Note that all FSM designs up to this point have magically used every code available in the count sequence or state diagram. For example, both of the examples we've explored contained four states which was the maximum number of states that we could represent using two flip-flops.

Now consider the case where we have a count sequence of five numbers that we want to implement using a FSM. For this case, we will need three flip-flops. The potential problem here is that with three flip-flops, we can represent up to eight states. The question that arises is what exactly happens to the other states? The answer is that in a super important solid design, you should know what those states are doing. The problem is you want you FSM to fix itself if it finds itself in a state that it was not intended to be in. In general, you need to design a fix into the hardware; this approach is referred to as illegal state recovery. The next example sheds light on the problem; you'll hopefully see that this is not a major concept.

Example 26-5

Design a counter that counts in the following sequence: 0, 5, 7, 3, 6, 0, 5… For this example, provide a PS/NS table only. Use only D flip-flops in your design. Make sure all unused states are directed to state "000".

Solution: This problem is similar to the other counter problems except there are more numbers in the count sequence. The first step in problems such as these is to draw the state diagram. The tendency here is to draw the desired sequence first as shown in Figure 26.31. This problem is special in that we need to represent the states not listed in the counting sequence.

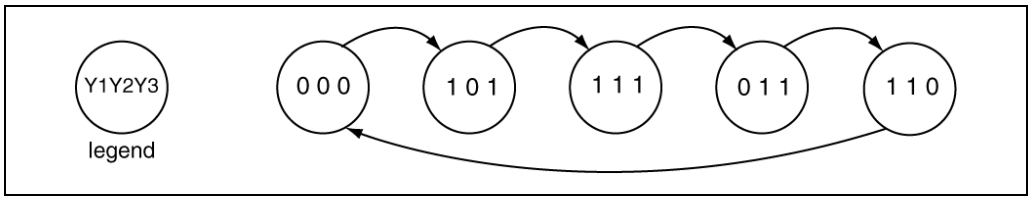


Figure 26.31: The initial state diagram for Example 26-5.

The first attempt to represent the unused states would maybe appear something like the diagram of Figure 26.32. This state diagram includes the unused states of "001", "010", and "100", but it is initially unclear what to do with them. If we had not listed these states, meaning we did not care about them, we could use "don't cares" in the associated rows of the PS/NS table. In this way, we direct the unused states to some other state, but we don't know which state until equations for the excitation logic is completed[7]. Keep in mind that the excitation logic for these examples is completed using K-maps; the unused counts in the sequence appear as "don't cares" in the excitation input columns in the PS/NS table. Recall that it is the K-map groupings that decided if the don't cares are assigned a '1' or a '0'.

[7] This means that it's the final K-map groupings that decided whether the "don't care" cells in the K-map will be either a '1' or a '0'.

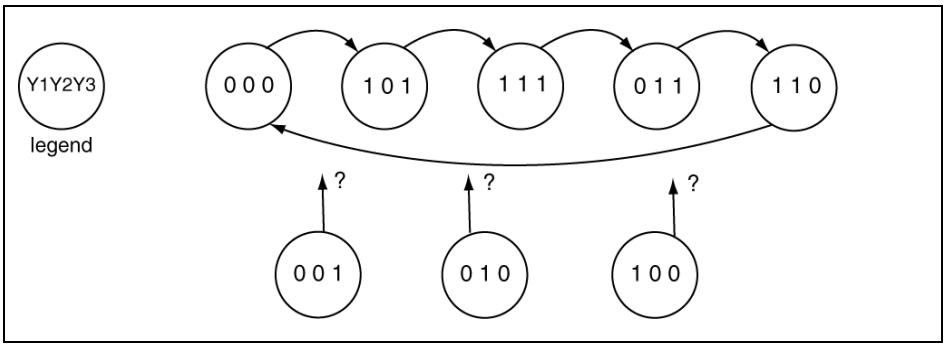


Figure 26.32: The PS/NS table for Example 26-5.

What we're trying to avoid in this problem is the generation of as *hang* states. In the state diagram, if we do not explicitly direct all the unused states back to the desired counting sequence, we may end up with a state diagram that has hang states. Figure 26.33 shows an example of a state diagram with hang states. Note that in Figure 26.33 we do indeed have the desired sequence; but we also have the unused states included in a pattern that we could implement because of our particular choice for the excitation logic. The thought here is that there is some magic entity that assures your FSM will always start in a certain state in the desired sequence. After that, if life is good, FSM never strays from that sequence.

In reality, FSM are implemented with actual circuits (it's the electronic thing). That means they are susceptible to various types of noise[8]. It just may happen that the noise places you in a state that is not part of the desire sequence. If this happens, you'll never make it back to the desire sequence. The FSM is thusly hung because it is stuck in a hang state. Figure 26.33 shows two flavors of hang states. The "001"-"010" pair is a small cycle; the "100" state is a self-looping hang state. In either case, there is no path back to the original counting sequence, which may or may not be important to the problem at hand[9]. Bummer!

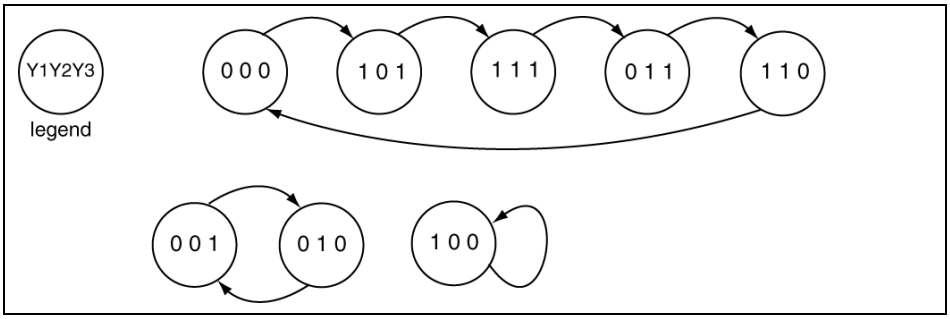


Figure 26.33: A state diagram containing hang states and other terrible things.

However, all is not lost. The approach that saves the day is to direct the unused states back to a state in the desired count sequence. In this way, if for some reason your FSM finds itself[10] in a hang state,

[8] This refers to unwanted electronic effects. A loud stereo will most likely have not effect on your digital circuit designs.

[9] Imagine if your FSM was controlling a heart pacemaker; it would not be good if your FSM got hung in a state that no longer directed the heart to beat. Of course, this would not matter for academic administrators as they have all had their hearts surgically removed as the basic requirement of their employment in academia.

[10] Yes Virginia, FSMs are self-aware (or about as self-aware as the average academic administrator).

you'll quickly (in one clock cycle) return to a count in the sequence. The problem description states that you should direct all of your unused states back to state "000". Figure 26.35 shows the resulting state diagram for this approach. From this point, it is not a big deal to generate the PS/NS table using techniques we've used in previous examples. Figure 26.34 shows the final PS/NS table for this example.

(PS)			(NS)			Excitation Inputs		
Y1	**Y2**	**Y3**	**Y1+**	**Y2+**	**Y3+**	**D1**	**D2**	**D3**
0	0	0	1	0	1	1	0	1
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0
1	1	1	0	1	1	0	1	1

Figure 26.34: The final PS/NS table for Example 26-5.

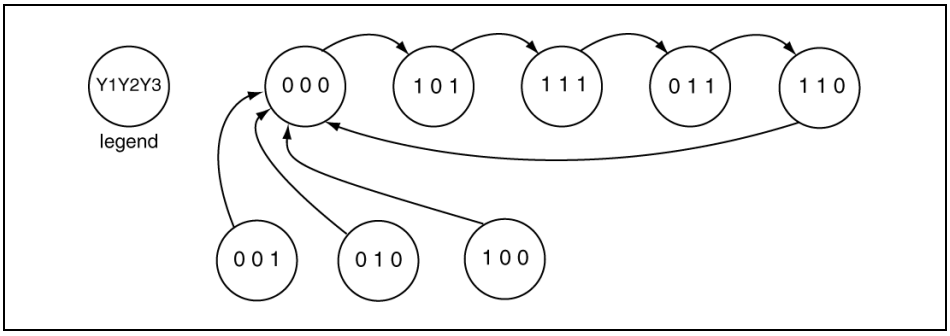


Figure 26.35: The state diagram with hang-state recovery.

(PS)			(NS)			Excitation Inputs		
Y1	**Y2**	**Y3**	**$Y1^+$**	**$Y2^+$**	**$Y3^+$**	**D1**	**D2**	**D3**
0	0	0	1	0	1	1	0	1
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0
1	1	1	0	1	1	0	1	1

Figure 26.36: PS/NS table for the Example 26-5.

Now that we included illegal state recovery in our FSM design, the FSM is said to be *self-correcting*. This means that if the FSM were to find itself in some undesired state, the FSM would eventually find its way back to the desired portion of the state diagram. Making your FSM designs self-correcting is important because statistically speaking, you're going to have unused states in your FSM as a result of the binary nature of the elements that are used to store the state variables.

Chapter Summary

- The basic model of a FSM includes three major parts: 1) the Next State Decoder, 2) the State Variables, and 3) the Output Decoder. Flip-flops are used to implement the state variables and represent the only sequential part of a FSM.

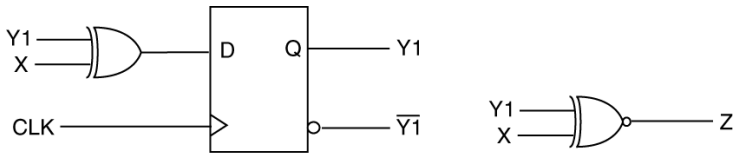
- The two major types of FSMs include the Mealy machine and the Moore machine. These two FSM types differ in only the output decoder: the outputs on a Moore machine are a function of the state variables only while the outputs of a Mealy machine are a function of both the state variables and the external inputs.

- FSM analysis starts with a circuit diagram and generates a state diagram and/or a PS/NS table. FSM design starts with either a state diagram or PS/NS table and generates a circuit diagram.

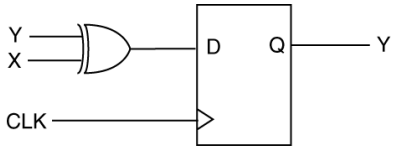
- FSMs can be designed such that they contain no hang states by designing them to have illegal state recovery attributes. FSM designs that do not contain hang states are *self-correcting*.

Chapter Exercises

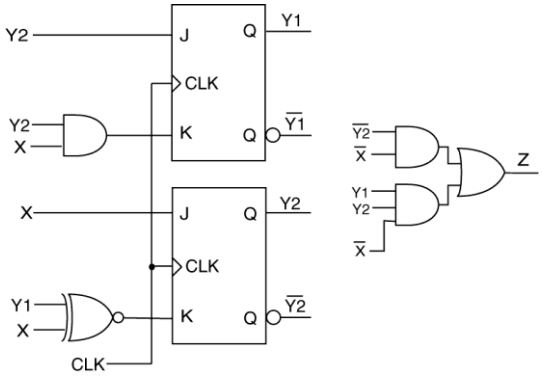
1) Analyze the following circuit and provide a PS/NS table and a state diagram associated with the circuit. Include the Z output in both the PS/NS table and state diagram.



2) Analyze the following circuit and provide a PS/NS table and a state diagram associated with the circuit. Include the Z output in both the PS/NS table and state diagram.



3) For the following circuit, provide a PS/NS table and a state diagram that describes the circuit. Include the output variable Z in both the PS/NS table and the state diagram. Be sure to provide a legend for you FSM.



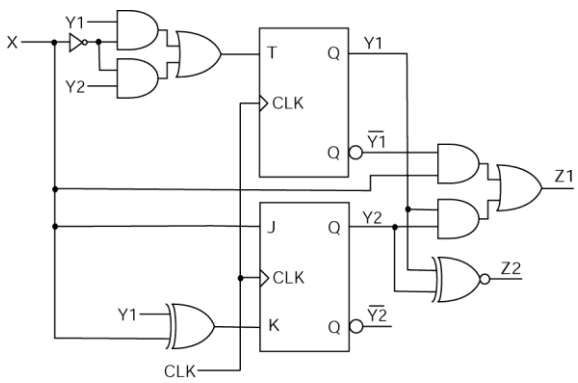
1) For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. You should *only* provide illegal-state recovery from states 3 and 4 (direct these states to state 6). Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). A Moore output Z is high only when the count is less than 3 in the intended counting sequence. Show a PS/NS table and state diagram that accounts for used states. Output sequence = 2,6,1,5,0. Make sure all excitation equations are in reduced form. *Don't draw the circuit – just provide the required excitation equations.*

2) For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. You should *only* provide illegal-state recovery from state 7 (direct this state to state 5). Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). A Moore output Z is high only when the count is less than 3 in the intended counting sequence. Show a PS/NS table and state diagram that accounts for used states. Output sequence = 2,3,4,0,5. Make sure all excitation equations are in reduced form. *Don't draw the circuit – just provide the required excitation equations.*

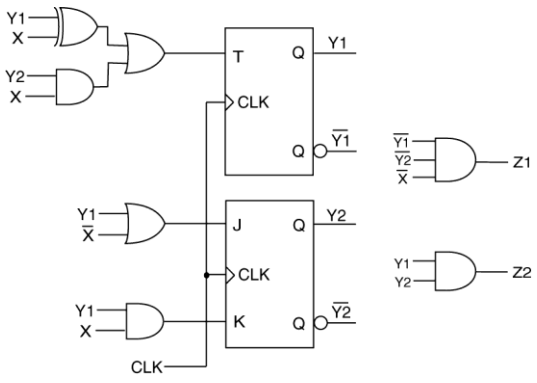
3) For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. You should *only* provide illegal-state recovery from states 3 and 4 (direct these states to state 2). Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). A Moore output Z is high only when the count is less than 3 in the intended counting sequence. Show a PS/NS table and state diagram that accounts for used states. Output sequence = 2,1,6,0,5. Make sure all excitation equations are in reduced form. *Don't draw the circuit – just provide the required excitation equations.*

4) 5. For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. You should ***only*** provide illegal-state recovery from state 2 (direct this state to state 7). Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). A Moore output Z is high only when the count is an odd number in the intended counting sequence. Show a PS/NS table and state diagram that accounts for used states. Output sequence = 0,3,4,6,7. Make sure all equations are in reduced form. *Don't draw the circuit – just provide the required excitation equations.*

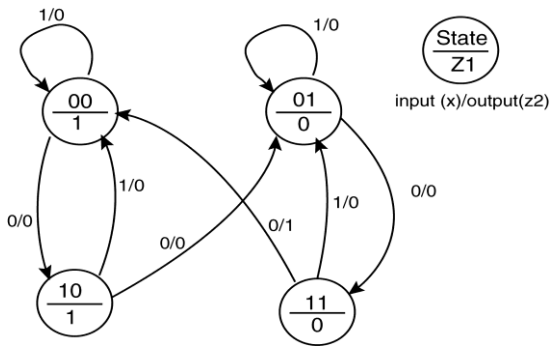
5) For the following circuit, provide a PS/NS table and a state diagram that describes the circuit. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram.



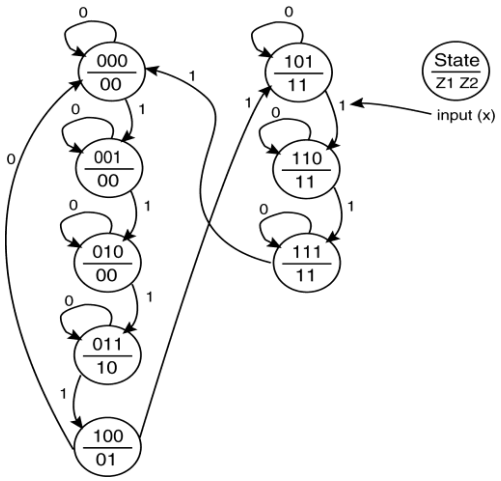
6) For the following circuit, provide a PS/NS table and a state diagram that describes the circuit. Include the Z1 and Z2 output variables in both the PS/NS table and the state diagram. Be sure to provide a legend for you FSM.



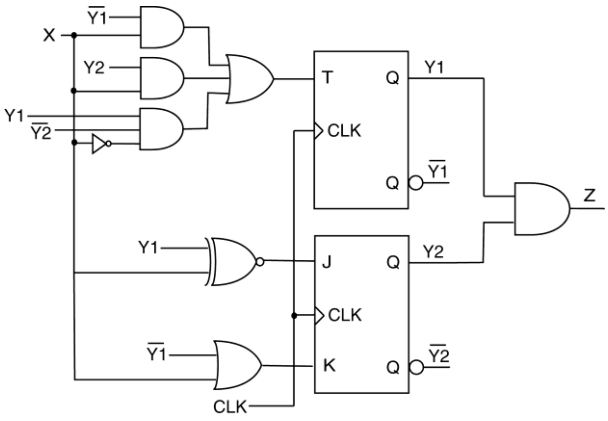
7) Draw a circuit that implements the following state diagram. Use one T flip-flop and one JK flip-flop in your design. Minimize the amount of required combinatorial logic.



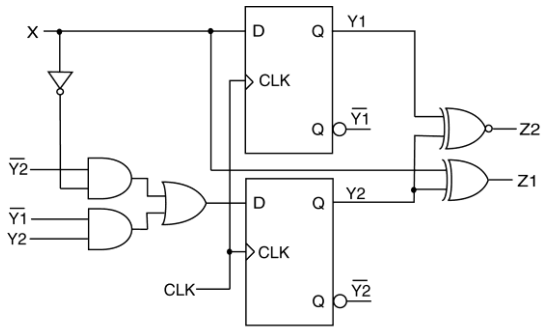
8) Draw a circuit that implements the following state diagram. Use only D flip-flops in your design. Minimize the amount of required combinatorial logic.



9) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



10) For the following circuit, provide a PS/NS table and a state diagram. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram.

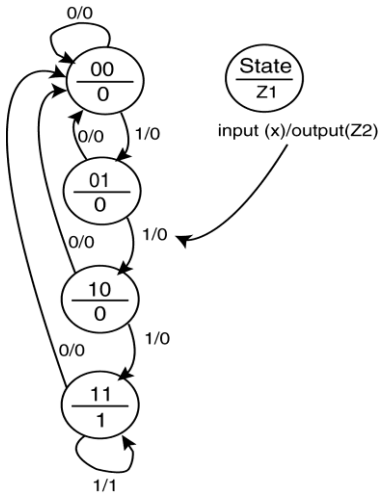


11) For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. Provide illegal-state recovery by directing all unused states to state 6. Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). Show a PS/NS table and state diagram that accounts for all states. Sequence = 2,3,4,5,6. Make sure all excitation circuits are in reduced form. *Don't draw the circuit – just provide all excitation equations.*

12) For the following output sequence, design a synchronous counter that repeats the given sequence indefinitely. Provide illegal-state recovery by directing all unused states to state 1. Use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). Show a PS/NS table and state diagram that accounts for all states. Sequence = 7,6,5,2,1. Make sure all excitation circuitry are in reduced form. *Don't draw the circuit – just provide all excitation equations.*

13) For the following output sequence, design a synchronous counter that repeats indefinitely. Provide illegal-state recovery by directing all unused states to state 7. Use positive edge triggered flip-flops and use a D flip-flop for the MSB (most significant bit), a T flip-flop for the middle bit, and a JK flop-flop for the LSB (least significant bit). Show a PS/NS table and state diagram that accounts for all states. Sequence = 1,2,4,5,7. Make sure all excitation circuits are in reduced form. *Don't draw the circuit – just provide all excitation equations.*

14) The following state diagram is to be implemented using D flip-flops and any type of discrete logic gates. Write the reduced excitation equations for the D flip-flops and the equations for the Z1 and Z2 outputs. *Don't draw the final circuit.*

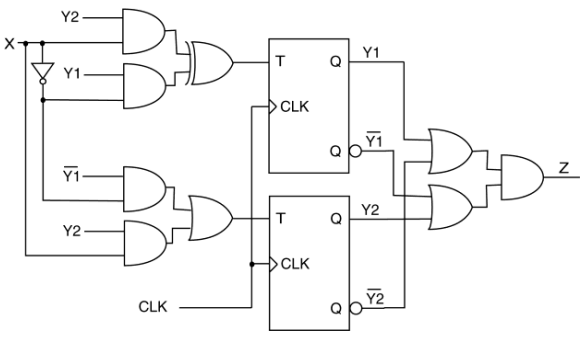


15) Design a counter that counts in the following sequence: 0,3,2,0,3,2,0… Provide illegal-state recovery into state 3. Use T or JK flip-flops for each of the state variables and use binary encoding. Show a PS/NS table and state diagram that describes the FSM. Provide flip-flop excitation equations in reduced form. *Don't draw the circuit.*

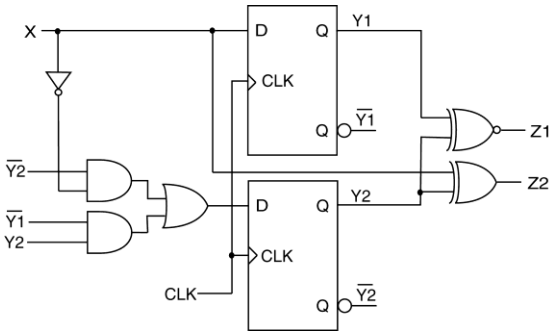
16) The following equations describe a counter that steps through a 5 number sequence. The counter only uses D flip-flops. Using the equations below, redesign the counter and make it self-correcting by sending unused states to *any* state in the 5-number sequence. Provide new D excitation equations in compact minterm form. Show the old state diagram and the new state diagram and account for all possible states. *Don't draw the final circuit.*

$$D1(Y1,Y2,Y3) = \sum(2,3,4,6,7)$$
$$D2(Y1,Y2,Y3) = \sum(1,3,6,7)$$
$$D3(Y1,Y2,Y3) = \sum(0,4,6)$$

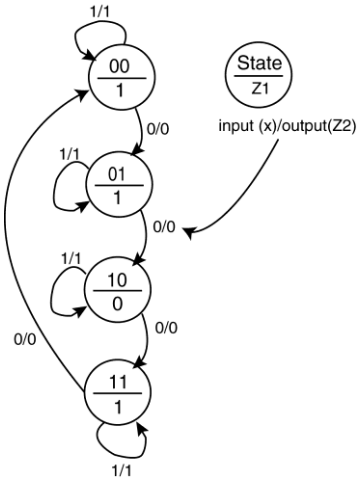
17) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



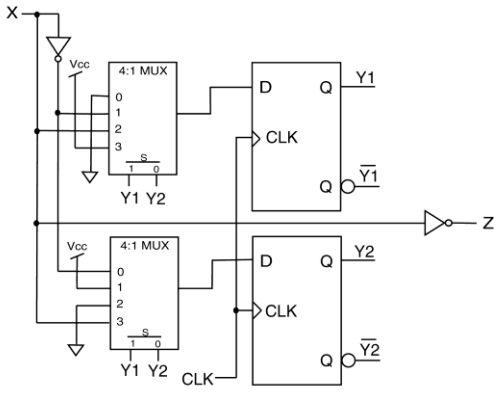
18) For the following circuit, provide a PS/NS table and a state diagram. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram.



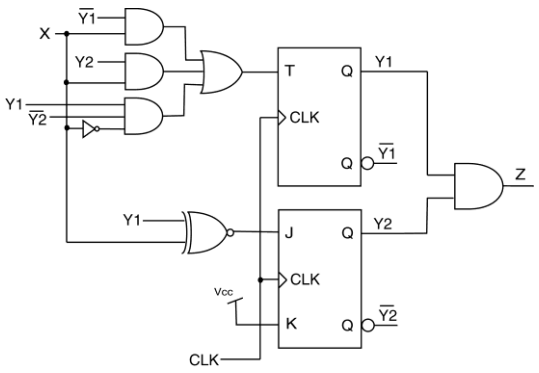
19) The following state diagram is to be implemented using D flip-flops and any type of discrete logic gates. Write the reduced excitation equations for the D flip-flops and the equations for the Z1 and Z2 outputs. *Don't draw the final circuit.*



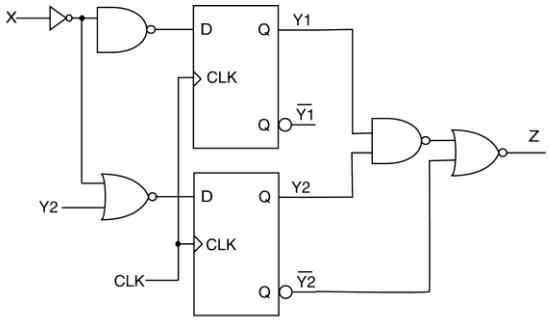
20) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



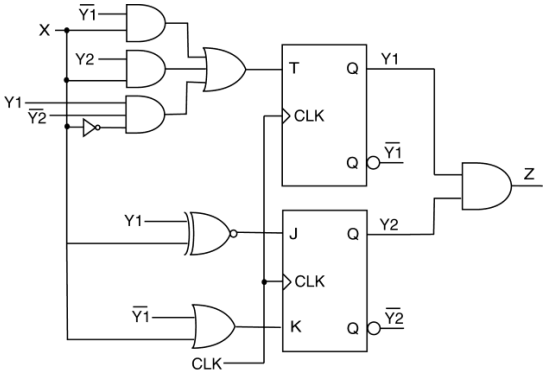
21) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



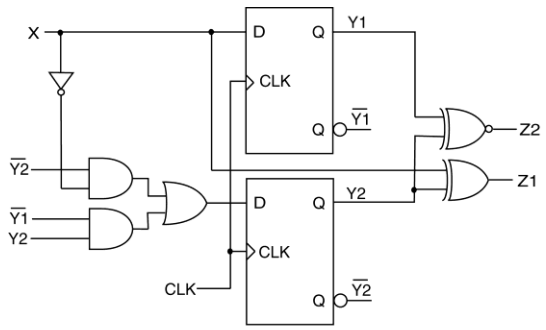
22) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



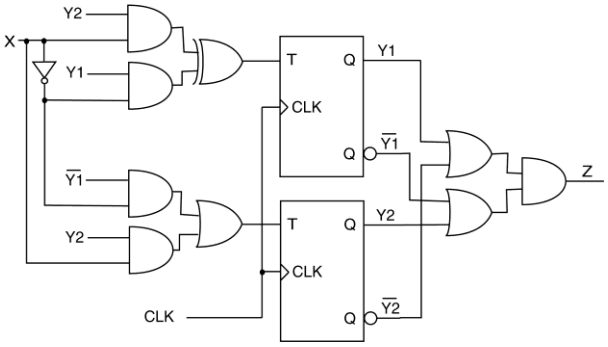
23) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



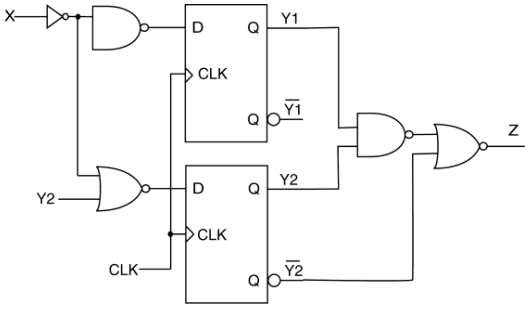
24) For the following circuit, provide a PS/NS table and a state diagram. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram.



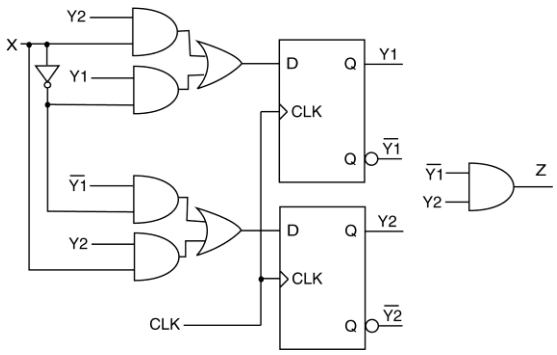
25) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



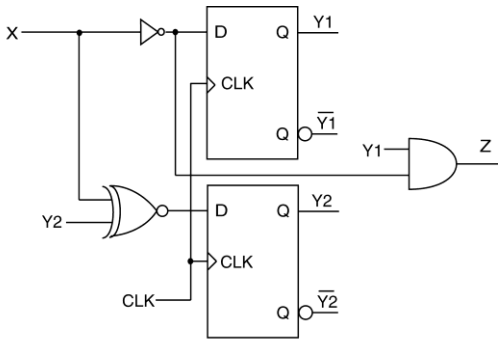
26) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



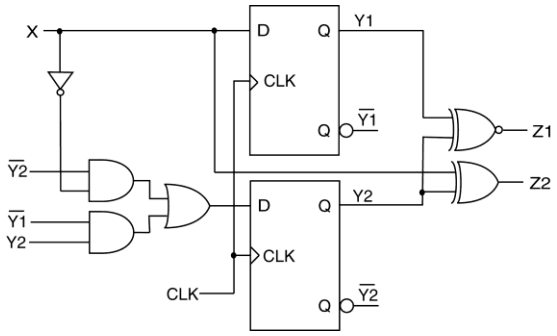
27) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



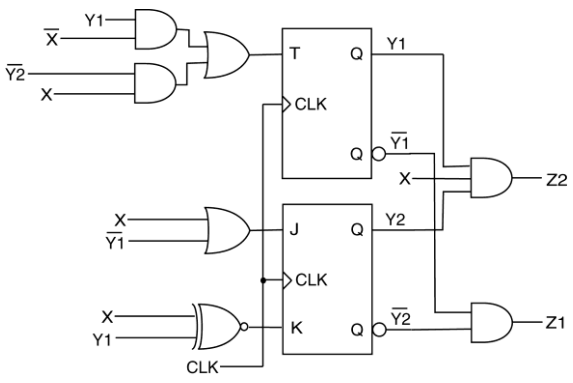
28) For the following circuit, provide a PS/NS table and a state diagram. Include the output variable Z in both the PS/NS table and the state diagram.



29) For the following circuit, provide a PS/NS table and a state diagram. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram.



30) For the following circuit, provide a PS/NS table and a state diagram. Include the output variables Z1 and Z2 in both the PS/NS table and the state diagram. Be sure to include a legend for your state diagram.



27 Chapter Twenty-Seven

(Bryan Mealy 2012 ©)

27.1 Chapter Overview

The primary focus of the previous few chapters was the introduction of the various aspects involved in finite state machines. The idea that we've been claiming is that FSMs are typically used as counters and controllers. We've done a few example problems that highlighted FSM use as a counter and upcoming chapters will show how FSMs act as controller circuits.

A major aspect of working with FSMs is to understand their underlying timing issues. This is particularly true with FSM circuits that act as controllers, as the FSM's output signals are what controls other circuits. We've skipped over most timing issues up to now, but we'll take a look at a few in this chapter. In the real world, we need to deal with many timing issues. Some of these issues are somewhat advanced, such as issues dealing with propagation delays through the various sub-blocks of the FSM. For this chapter, we'll primarily focus on the differences in timing diagrams associated with the Mealy and Moore-type FSMs. These issues are not overly complicated but you'll for sure find it helpful to work through a few meaningful examples.

Main Chapter Topics

- **FSM AND TIMING DIAGRAMS:** FSMs have interesting timing aspects, particularly in the context of Mealy vs. Moore FSMs. The timing associated with state diagrams represents the key to using FSMs for many applications as these timing aspects are essential to the proper operation of FSMs when used as controllers.

Why This Chapter is Important

This chapter is important because it introduces some of the major timing aspects associated with FSMs, particularly the differences between Mealy and Moore output timing.

27.2 Finite State Machines (FSMs): The Quick Review

Our workings with FSMs is divided into three distinct steps: 1) a high-level overview of the concepts and associated terminology, 2) analysis and design of FSMs in circuit form and their relation to the state diagram, and 3) developing the state diagram. The first two steps are straight-forward and almost mechanical in nature. What we've hopefully learned from these steps is a basic understanding of state diagrams and their relation to digital circuitry. The third step is the engineering step as it involves creating state diagrams.

Although any sequential circuit is officially a FSM, we'll consider FSMs to be *circuits that control other circuits*. Figure 26.1 and Figure 26.2 show the two types of FSMs: *Moore* and *Mealy* machines. There are many similarities between these two FSM models but they do have one big difference: The external outputs of a Mealy machine are a function of both the current state and the external inputs while the external outputs of a Moore machine are exclusively a function of the current state of the FSM. We described the functions associated with the basic FSM blocks in a previous chapter and we'll not waste the ink here.

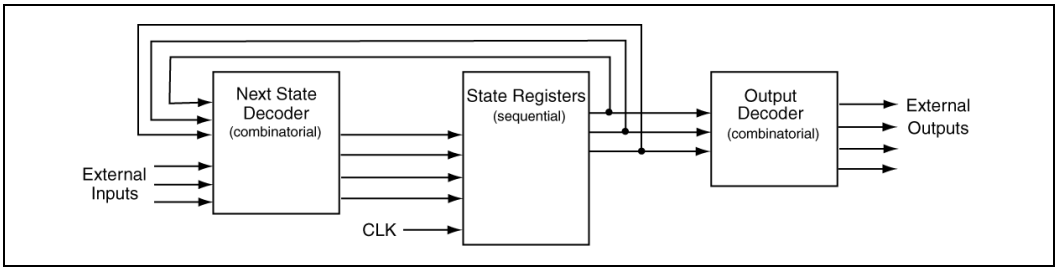


Figure 27.1: Model for a Moore-type FSM.

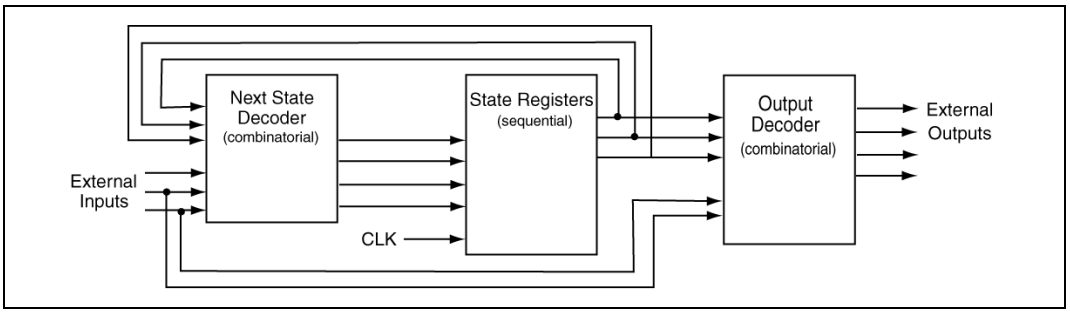


Figure 27.2: Model for a Mealy-type FSM.

27.3 Timing Diagrams: Mealy vs. Moore FSM

Our previous FSM discussions dealt primarily with FSM design and analysis. We knew that the storage elements (the flip-flops) were edge-triggered which allowed transitions between states to occur only on the active clock edges. We also had several flip-flops to choose from (D, T, and JK) when implementing our FSMs[1]. The FSMs were defined with either a PS/NS table or a state diagram (both presented the same information but in a different format). The FSMs had external inputs and external outputs which were generally named with X and Z variables, respectively. The external outputs could be either a Moore-type or Mealy-type; both outputs were a function of the present state of the FSM but Mealy-type outputs were also a function of external inputs. We also know that ultimately, the FSM would be used to control some other circuit. The external inputs (X) would provide information to the FSM, which would subsequently control the Mealy-type outputs and the state-to-state transitions. Despite what academic-types think, there is more to FSMs than designing counters of various types.

[1] Although we could choose between the three different types, PLD-based implementations primarily use D flip-flops (more on this later). The T and JK flip-flops have their advantages, but these advantages are mitigated in FPGA-land (because FPGAs have a lot of D flip-flops on-board).

The study of FSMs can be broken into four parts. You know the first part: the design and analysis of FSMs. The second part is to develop a basic understanding of state diagrams, which we dealt with in a previous chapter, and will be deal with further in an upcoming chapter. The third aspect is the timing diagrams associated with a given state diagram and the timing diagram's relation to the problem in need of solution. The final aspect, which is in an upcoming chapter, is to design FSMs from a given problem specifications.

Up until now, we really have not mentioned too much about FSM timing considerations. Possibly our only mention of the timing associated with the FSMs was that the state transitions only occurred on a clock edge. In reality, you're missing a big part of the story: the underlying timing diagram. Standard digital logic textbooks[2] typically dismiss the importance of timing diagrams. This chapter is primarily concerned with timing diagrams and their relation to FSMs. Because state diagram nice define FSMs, timing diagrams are the best place to start this discussion. From any state diagram, you can easily generate an associated timing diagram that adheres to the state diagram.

27.3.1 Timing Diagrams and State Diagrams

There are some classic problems associated with the relationship between the timing aspects associated with state diagrams. The thought here is that, under most circumstances, you should be able to generate a state diagram from a timing diagram and/or vice-versa. The good thing about exploring this relationship is that it clearly shows the relationship between the Mealy outputs listed in state diagrams and the effect they have on the associated timing diagrams. In my opinion, understanding this relationship is the most challenging part of developing a solid understanding of FSMs. However, once you have this understanding, you'll be able to take the final plunge in FSM-land: generating your own timing diagram from a set of specifications.

We center this section around a few example problems that will hopefully help you understand FSMs on an intuitive level. Understanding FSMs on this level is the key to being able to design real FSM circuits that actually do something other than count (did I already deliver that insult?).

Example 27-1

Draw a state diagram that could generate the timing diagram shown in Figure 27.3. Consider CLR to be an active low signal that resets the FSM. X is an external input and Z1 and Z2 are external outputs.

[2] The inadequate treatment of timing diagrams by most digital design textbooks is well known. The other problems are too numerous to mention here (and painfully well known by all).

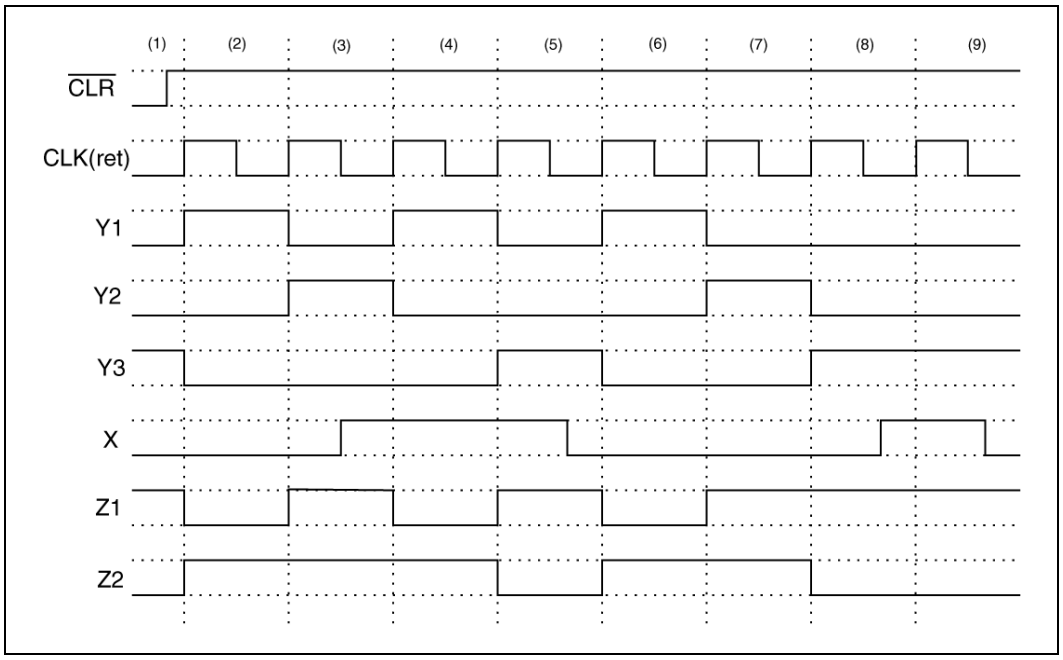


Figure 27.3: Timing diagram for Example 27-1.

Solution: The first thing to do with a problem like this is to stare at it and try to figure things out. If you were to do this for long enough, you'd notice items such as the things listed below.

- The FSM has one external input: X.
- The FSM has two external outputs: Z1 and Z2.
- The CLR signal places the FSM into a known state in the first time slot and then has no effect later in the timing sequence. In other words, the CLR signal asserts in the first time slot and then de-asserts for the remainder of the timing diagram.
- The clock is rising-edge triggered (RET). This fact is evident from examining the timing diagram: notice that all the state changes occur on the rising-edge of the clock signal.
- The outputs Z1 and Z2 are both Moore outputs. You know this because all of the changes in these outputs occur at the same time as changes in the state variables. In later examples, we'll look at some Mealy-type outputs.
- There are three state variables. Initially you may start thinking that there are eight possible states in this FSM, which is not a bad thought because there is a possibility that this FSM uses binary encoding. However, upon further examination of the timing diagram you'll notice that only one of the three state variables is in a high state at one time. This indicates that the FSM is one-hot encoded. The topic of one-hot encoding is not complicated: it simply means that there is one storage element (or flip-flop) for each state in the FSM. Furthermore, each state is represented by one storage element being in a '1' state while all other storage elements are in a '0' state[3]. The

[3] Much like a standard decoder's output.

timing diagram in this example thus indicates that this FSM has three states. We'll discuss one-hot encoding in more detail in a later chapter.

Without doing much work with this example, you can glean a lot of information. This should be your approach of choice for all of these problems because a solely mechanical approach can sometimes bypass true understanding of the problem and is certainly not an approach taken by real engineers[4]. The problem is now ready for solving.

Keep in mind that the hardest part of doing this problem may be to stay neat and organized. The task you should do is to explicitly list the states in terms of the state variables. This means you should group the variables into some order such as Y1Y2Y3. You should do the same for the output variables such as Z1 and Z2. Note that in this step, the ordering does not matter. What does matter is that you document what you're doing in a legend that is included with your state diagram. Figure 27.4 shows the results of this preliminary step in the upper left portion of the diagram.

Step 1) The first real step in the process of solving this problem would be to generate a simple state diagram that lists only the states and the state transitions. We accomplish this straightforward by reading the state variable information and the associated transitions from the timing diagram. Figure 27.4 shows the results of this step. Be sure to note the method used to represent the asynchronous CLR input. In order to stay organized, you may want to explicitly list the state variables in each of the time slots of Figure 27.3.

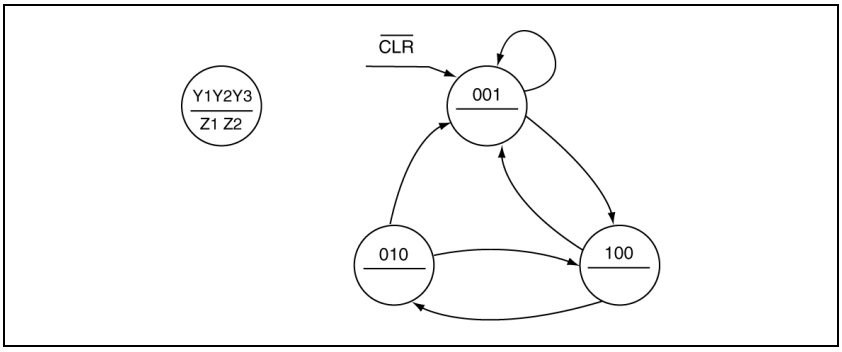


Figure 27.4: The results after the first step of Example 27-1 solution.

Step 2) Now that you have listed all the possible state transitions, list the external conditions that allow the state transitions to occur. As you would expect, since there are two arrows leaving each state bubble, the external input **X** is what determines the state transitions. We complete this step by examining the X input's value at the end of each state time-slot in the timing diagram. This is because the X input determines the next state based on the present state. The idea here is that when you considered any one state slot in the timing diagram, that slot is the "present state" which makes the next state slot the "next state". Figure 27.5 shows the results of this step once this entire analysis is complete.

[4] However, it is the approach taken by engineers who can't wait to become academic administrators: "lottabloat".

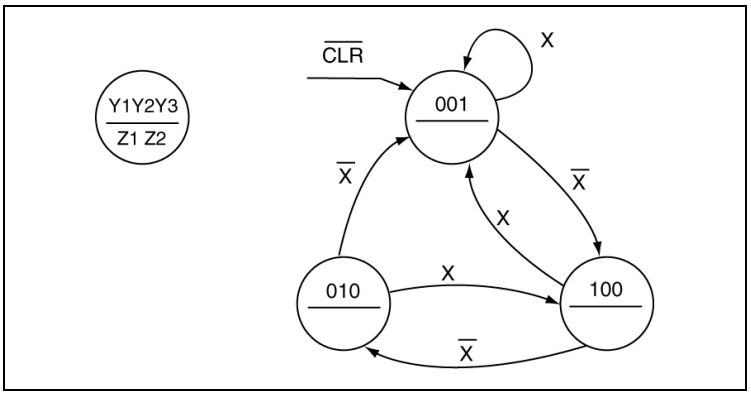


Figure 27.5: The results after Step 2 in the Example 27-1 solution.

Step 3) The final step is to include the outputs in the state diagram. Since these are Moore outputs, they can be (and should be) included within the state bubbles. The value of the outputs for each state is determined by examining the timing diagram for each output. When you perform this step, you can see that the output Z1 is a '1' except in state "100" and output Z2 is a '1' in all states except "001". If the person who drew the state diagram did a decent job, then there should be no ambiguities in the outputs in the context of the various states. It appears that the person who generated this problem did a fine job. Figure 27.6 shows the final solution for this example.

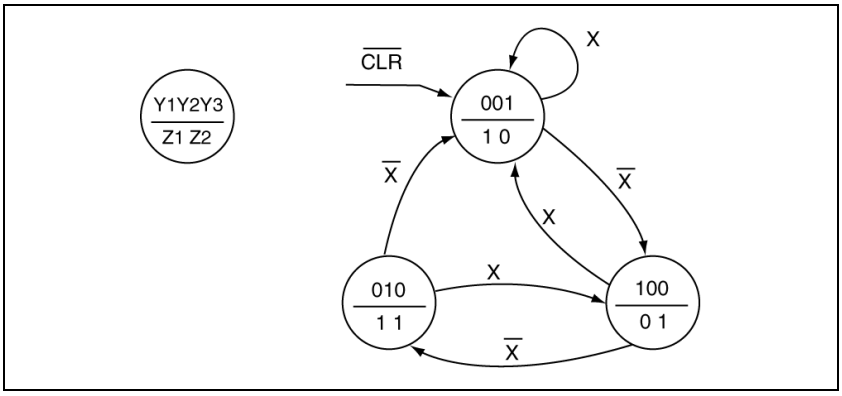


Figure 27.6: The final solution for Example 27-1.

Example 27-2

Draw a state diagram that could to generate the timing diagram shown in Figure 27.7. Consider CLR to be an active low signal that resets the FSM. X1 is an external input and Z is an external output.

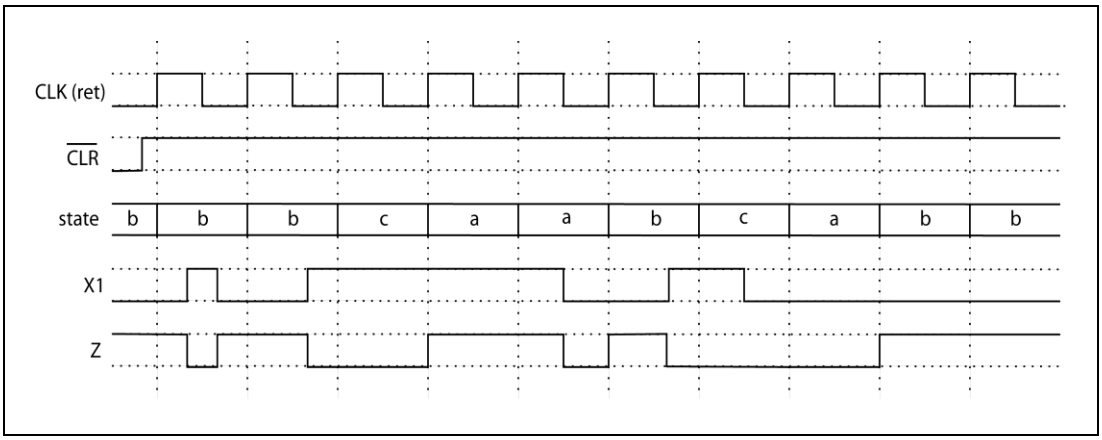


Figure 27.7: Timing diagram for Example 27-2.

Solution: The approach to solving Example 27-2 is similar to Example 27-1. The main difference between these problems is that Example 27-2 lists symbolic names for the states rather than listing the state variables as was the case in Example 27-1. In this way, you don't need to worry about what bits are representing the states. Besides that, taking a quick look at the problem (which always should be your first step in solving these problems) should yield the following information:

- The FSM has one external input: X1.
- The FSM has one external output: Z.
- The CLR signal puts the FSM into a known state in the first time slot and then has no effect later in the timing sequence.
- The clock is rising-edge triggered (RET). This fact is obvious from examining the timing diagram: all the state changes occur on the rising-edge of the clock signal.
- The output Z is a Mealy output. You can see this by the fact that changes in this output occur at places other then the active clock edges. The outputs in Example 27-1 were Moore-type and only changed with the states (on the clock edge). The Z output in this example is sometimes (but not always) synchronized to changes in the X1 input (and with the state variables which change on the active clock edge) which implies that Z is a function of X1.
- There are three states in this FSM, which you can derive by counting the different letters in the "state" row of the timing diagram.

Step 1) Draw a legend and a state diagram that shows only the state transitions. The information required for this step comes from the "state" row of the timing diagram shown in Figure 27.7. You can make the assumption in these types of problems that the timing diagram provides all the information you need to do the problem. In other words, you should ignore unlisted state transitions, should they actually exist.

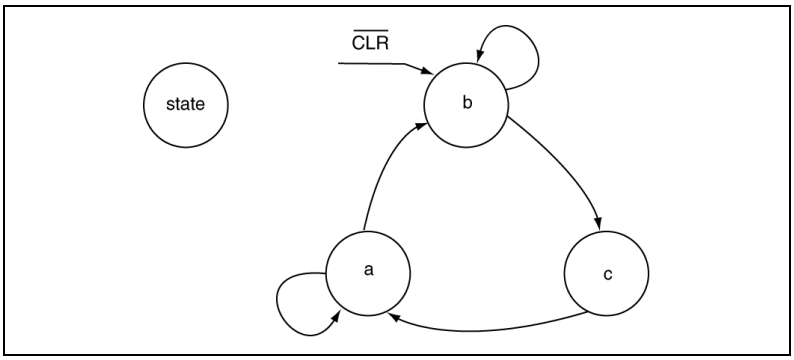


Figure 27.8: The results of Step 1 for Example 27-2.

Step 2) In the state diagram, list the conditions that cause the various state transitions listed in Figure 27.8. This information comes from examining the X row of the timing diagram. Note that there are two conditions associated with the transition from state **c** to state **a**. Although this may seem like a condition where you would place a *don't care* (a transition that happens on the clock edge regardless of the input conditions), you need to wait until you see what the output is doing. In other words, it's an unconditional transition from state **c** to **a**, but the output may be different depending on the X1 input. We'll check for this condition in the next step. Figure 27.9 shows the results of the current step.

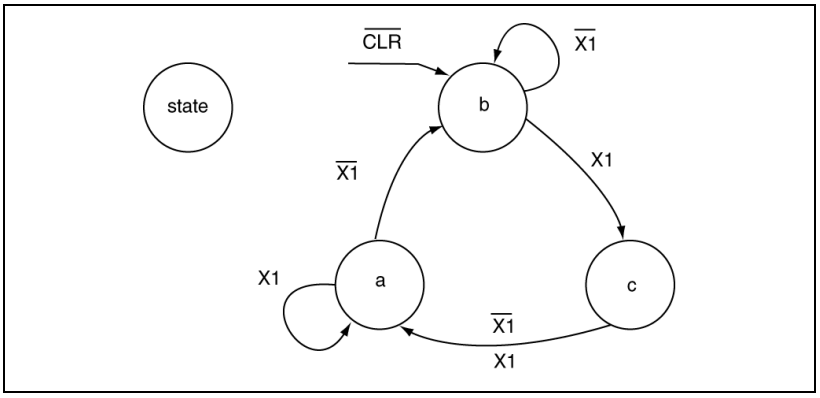
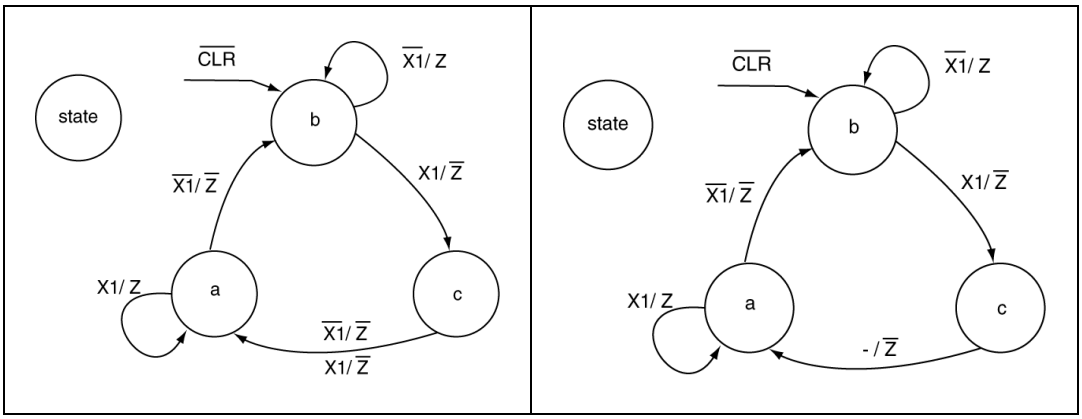


Figure 27.9: The results of Step 2 for Example 27-2.

Step 3) Add the values for the output variable. The output variable Z is a Mealy-type output its value is placed along side the input values in the state diagram. Figure 27.10(a) shows the results of this step. Notice that the value of the input variable does not matter for state (c) to state (a) state transition: the output Z is always a '0' which is listed in a more intelligent manner with the *don't care* symbol appearing in Figure 27.10(b). The example is complete; allow the celebration to begin.

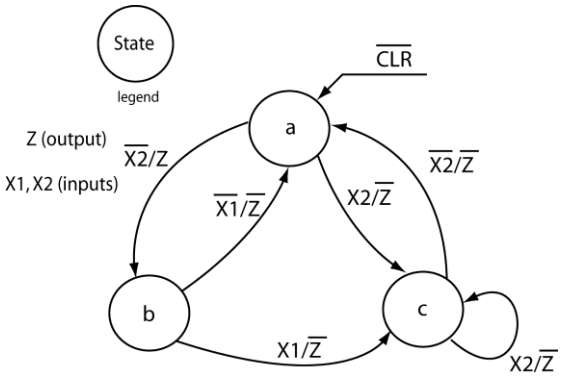


(a) (b)

Figure 27.10: The results of Step 3 for Example 27-2; (a) & (b) show two different solutions with (b) being the better approach.

Example 27-3

Use the state diagram below to complete the timing diagram shown in Figure 27.11. Consider CLR to be an active low signal that resets the FSM. X1 and X2 are external inputs and Z is an external output.



Solution: This example is slightly different from the previous two examples. It is the same basic FSM idea but the state diagram is given and you need to complete the timing diagram. The timing diagram is missing information for output Z and as well as the values of the states in each of the time slots. The approach to solving this problem is similar in that before you start, you should stare at the problem and see what is going on. If you are so bold as to take this step, you will be able to discern the following:

- The FSM has two external inputs: X1 and X2.
- The FSM has one external output: Z.

- The CLR signal in the first time slot places the FSM into a known state and then has no affect later in the timing sequence.

- The clock is rising-edge triggered (RET). This fact is obvious from examining the timing diagram: all the state changes occur on the rising-edge of the clock signal. We show this with the empty boxes in the state row of Figure 27.11.

- The output Z is a Mealy output. You can see this by examining the state diagram. Notice that the two transition arrows leaving state (**a)** have different values of the Z output; this characteristic is only true of state (**a)**. The key point to remember here is that the ***outputs are associated with the state that the arrow is leaving***. The fact that the Z output has two different values associated with state (**a)** indicates that the output is Mealy-type output. The output from the other two states essentially acts like a Moore-type output since they are for the given transitions from that state. You can thus expect the Z output to be changing with the X2 input because the output is a function of the X2 input, which the state diagram indicates. You should expect to see this characteristic in the timing diagram once it is completed.

- There are three states in this FSM, which you can discern by counting the different letters in the state diagram.

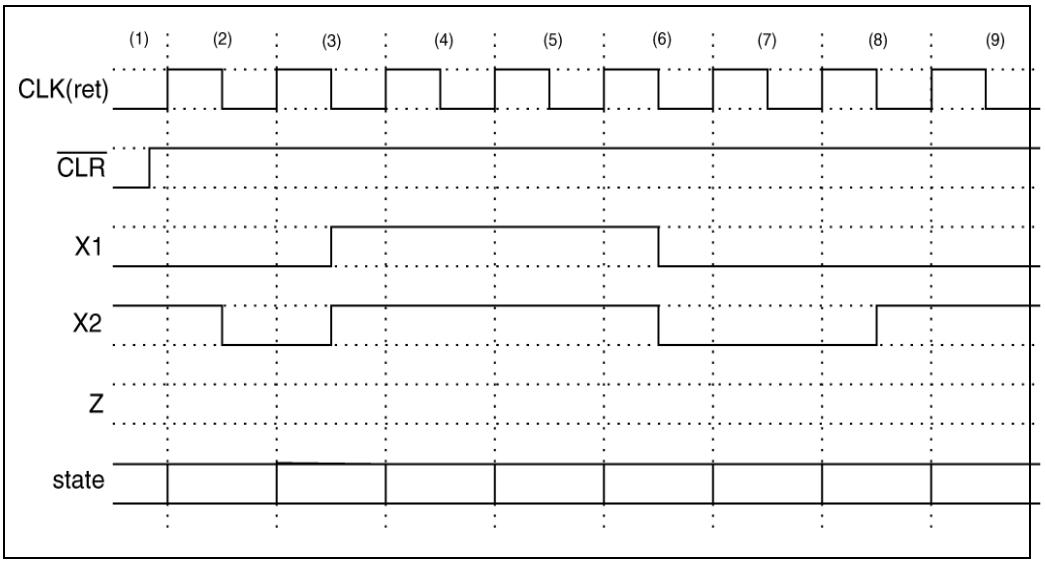


Figure 27.11: The uncompleted timing diagram of Example 27-3.

Step 1) The first step in this solution is to once again fill in the state transitions. To do this, you first need to establish a starting point. Lucky for us, the FSM is initially reset in the time slot before the first clock edge. By examining the state diagram, the asserted CLR signal places the output into state (a). Thus, we enter state (a) into the first box in the state row in the timing diagram. To fill in the other empty state boxes, you need to consider the present state and the input values that control the transitions from that state.

For state (a), the initial state of this example, the X2 inputs controls transitions from this state, which you can discern by examining the state diagram. If X2 is a '0', the FSM transitions from state (a) to state (b); if X2 is a '1', the FSM transitions from state (a) to state (c). To discern the state transition on

the first clock edge, you must first check the state of the X2 input in the associated state diagram. Examining the state diagram, you'll see that this input is a '1' so the FSM transitions to state (c). We continue this approach for every state in the state row. Notice that the state transitions associated with states (a) and (c) are dependent upon the X2 variables while state transitions associated with state (b) are associated with the X1 variable. Figure 27.12 shows the results of this step.

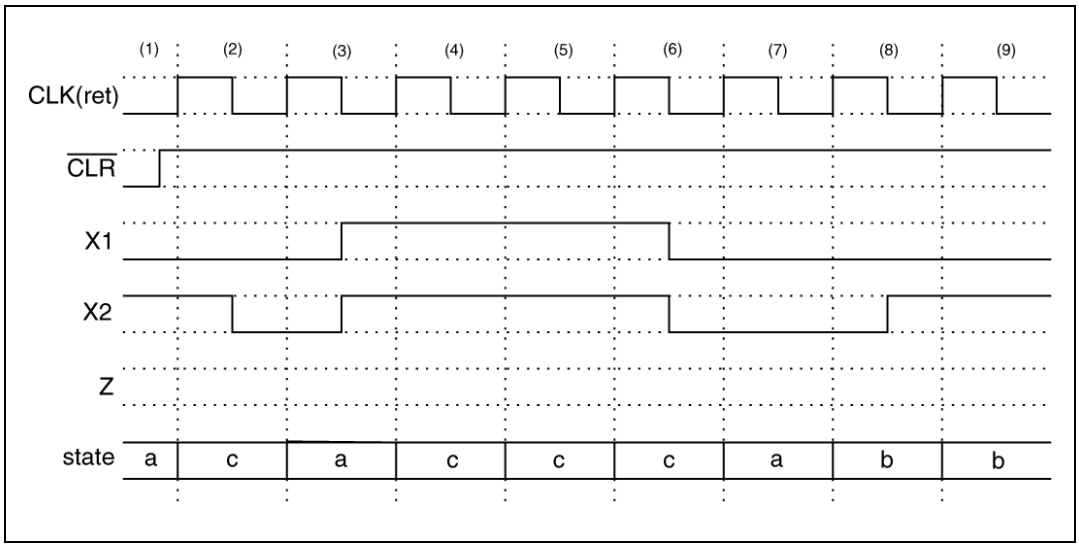


Figure 27.12: The results after the first step in the solution of Example 27-3.

Step 2) The second and final step associated with this example is to fill in the Z output based the present state and the X1 and Z2 inputs. To do this, you need to examine the state diagram for each state. For example, in the first time slot (where the resent takes place), the FSM is in state (a). From the state diagram, in state (a), the value of the Z output is based on the value of the X2 input. More specifically, if X2 input is a '1', the Z output is a '0'. Therefore the Z output would state in the '0' state. The second time slot is associated with state (c). In state (c), you'll notice that the output is always a '0' which is independent of any input variable. You'll also notice that state (b) has the same condition. In states (b) and (c), the output Z actually has Moore-type qualities. What makes it a true Mealy-type output is the output conditions associated with state (a). In other words, in state (a) the Z output is dependent upon the value of the X2 input. We take this approach for the entire Z row in the timing diagram. Figure 27.13 shows the results. A viable approach to these problems is to fill in the Moore outputs first since they require less neurons to successfully complete.

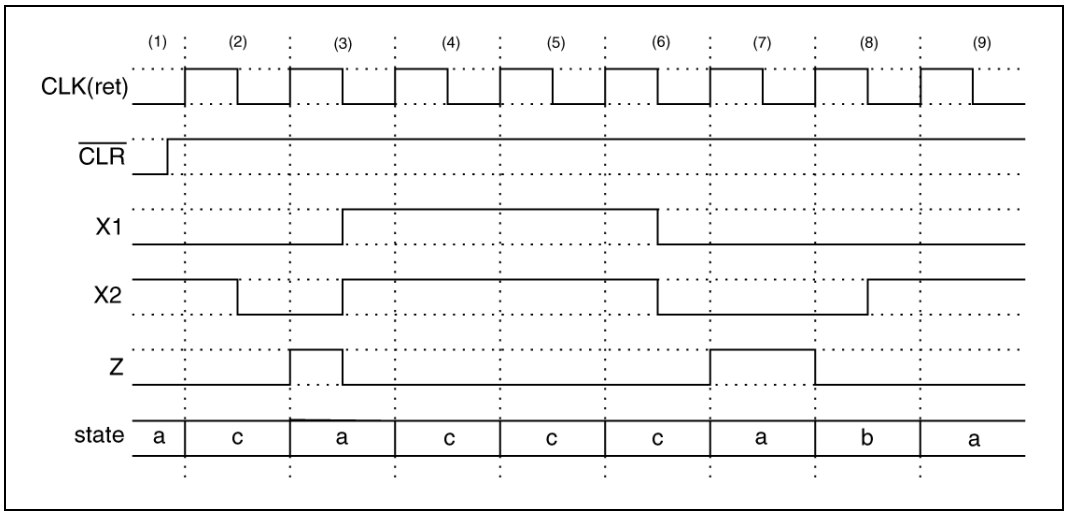


Figure 27.13: The final result for Example 3.

Chapter Summary

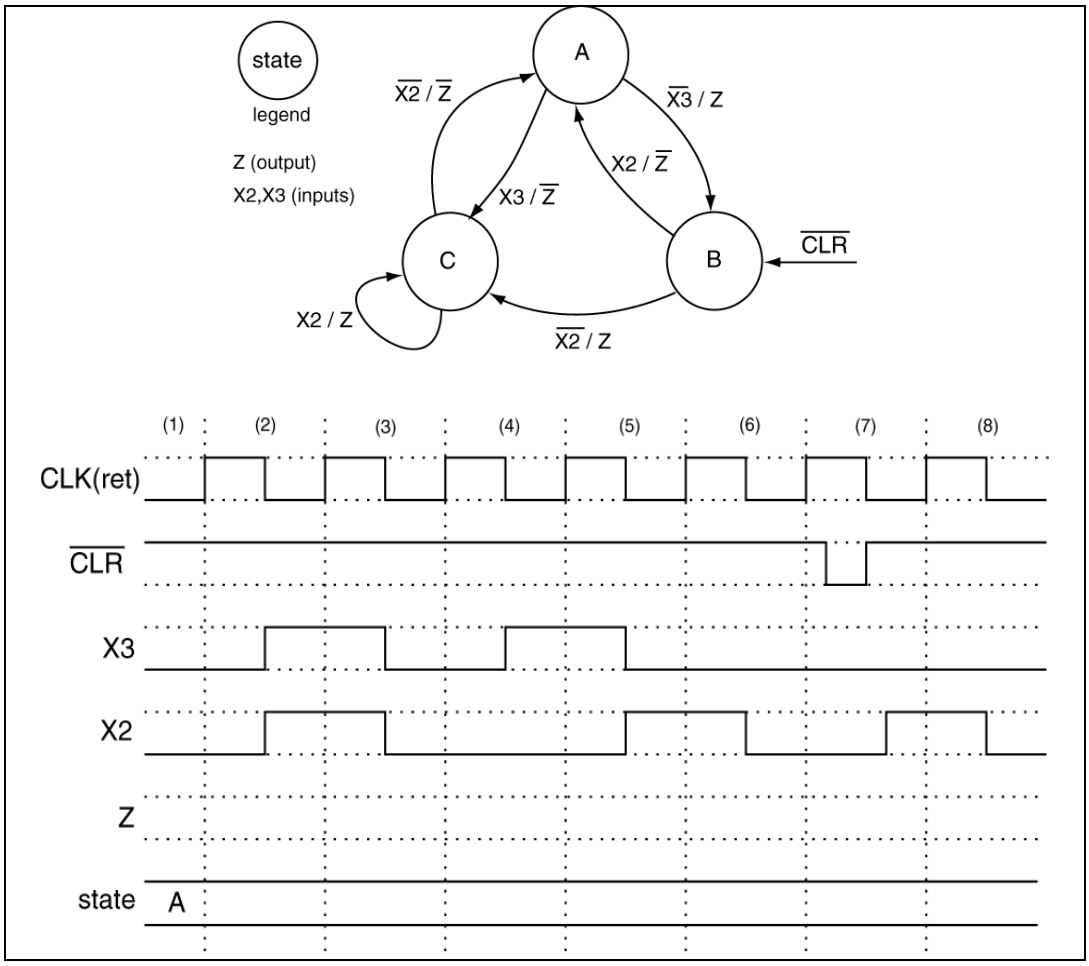
- The Moore-type outputs of a FSM can only change on the active clock edge associated with the FSM. If a particular output only changes on active clock edges, the output is a Moore-type output.

- The Mealy-type outputs of a FSM can only change either on active clock edges or with external inputs. If a particular external output changes anywhere else but an active clock edge, then the output must be a Mealy-type output because Moore-type outputs change only on active clock edges.

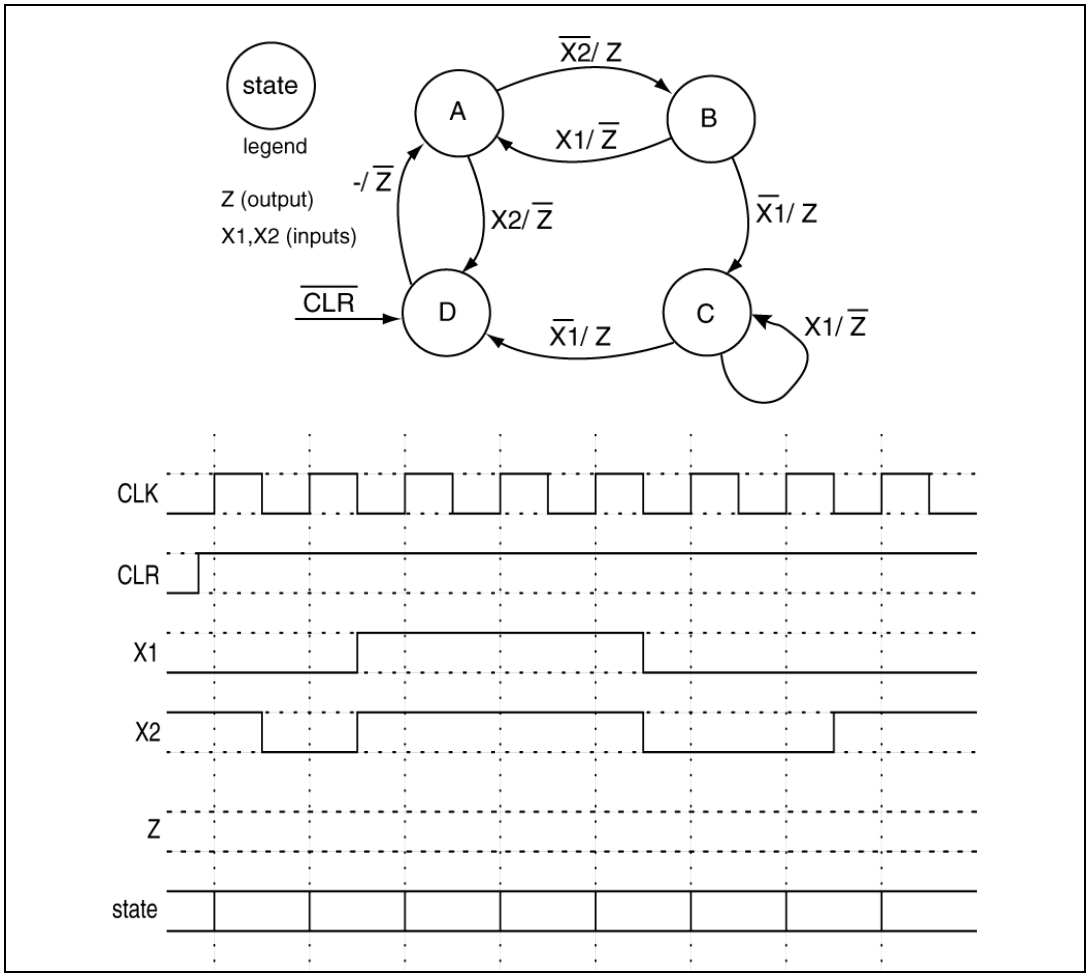
- Timing diagrams provide extremely valuable information describing the operation of a particular FSM. You can successfully derive a complete state diagram from a complete timing diagram (and vice versa).

Chapter Exercises

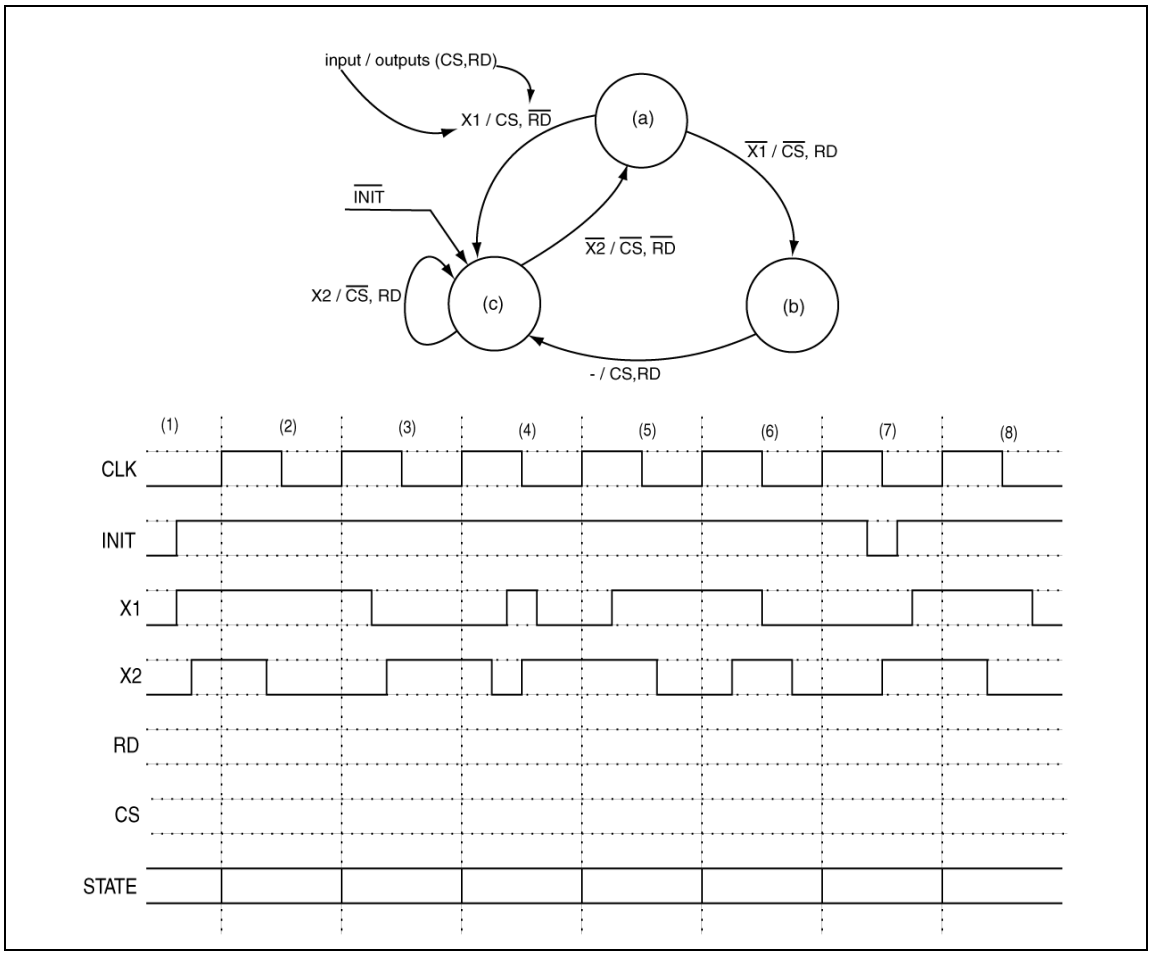
1) Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and output Z by filling in the "state" and "Z" lines in the timing diagram. Assume that propagation delay times are negligible. Assume state transitions occur on the rising edge of the clock signal. Assume CLR is an asynchronous, active low input.



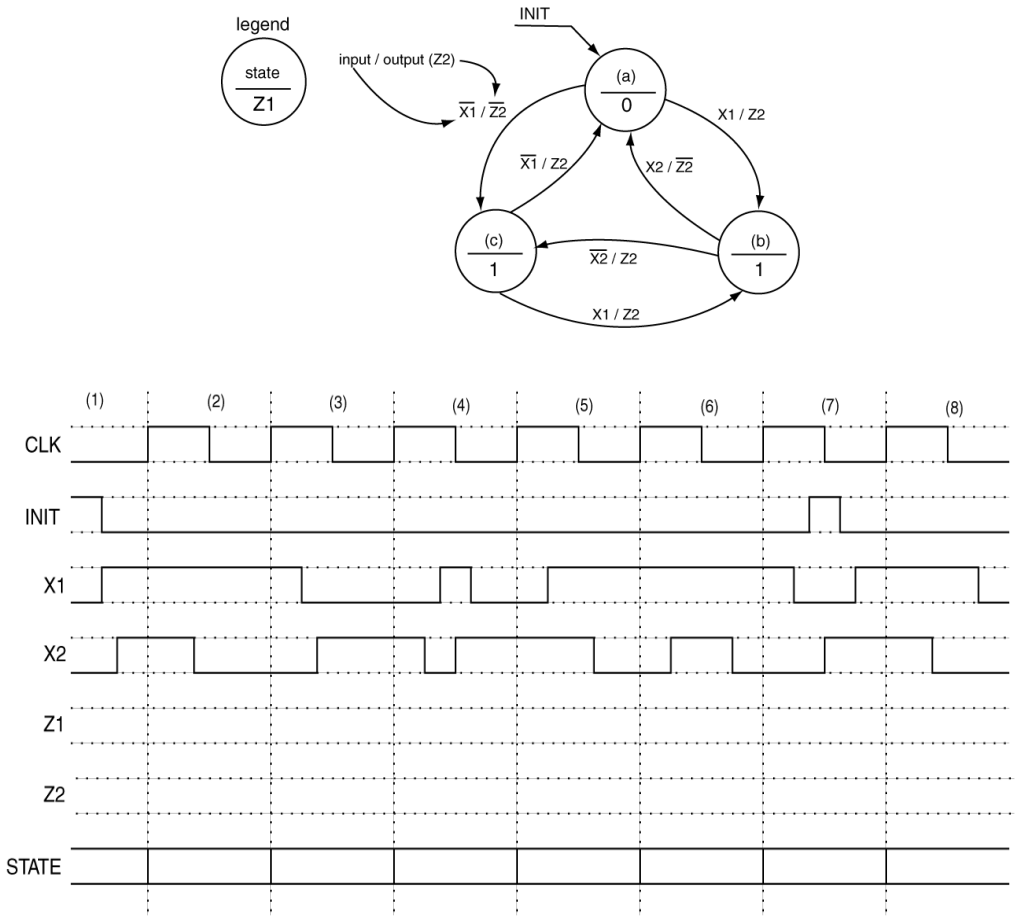
2) Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and output Z by filling in the "state" and "Z" lines in the timing diagram. Assume state transitions occur on the rising edge of the clock signal. Assume CLR is an asynchronous, active low input.



3) Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and outputs Z1 and Z2 by filling in the "STATE", "RD", and "CS" lines in the timing diagram. Assume that propagation delay times are negligible. Assume state transitions occur on the rising edge of the clock signal. Assume INIT is an asynchronous, active low input.



4) Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and outputs Z1 and Z2 by filling in the "STATE", "Z1", and "Z2" lines in the timing diagram. Assume that propagation delay times are negligible. Assume state transitions occur on the rising edge of the clock signal. Assume INIT is an asynchronous, active high input.



28 Chapter Twenty-Eight

(Bryan Mealy 2012 ©)

28.1 Chapter Overview

The previous chapters presented an over view of FSMs as well as some FSM design, analysis, and timing diagram techniques. Since this was a significant amount of information, it gave the appearance that the associated issues were complicated. But as you probably discovered after attempting a few problems on your own, you found these design and analysis techniques rather cookbook[290]. This chapter also covers more cookbook material. Modeling FSMs using VHDL behavioral modeling is similar to FSM design and analysis in that once you finish a few problems, you'll begin to view the topic as somewhat trivial.

As previously mentioned, FSMs are generally used as controllers in digital circuits. After working through the previous chapters, you've probably designed quite a few state machines on paper, but there was no real point for the design. You're now to the point where your designs still won't have much point but you'll be able to implement and test them using actual hardware if you so choose. The first step in this process is to learn how to model FSMs using VHDL. But have no fear, the theme of upcoming chapters are to use FSMs as they are intended; that's when the real fun begins[291].

Main Chapter Topics

- **MODELING FSMS USING VHDL:** The approach to representing FSMs using VHDL is straightforward. The power of VHDL and its behavioral modeling capabilities allows you to represent FSMs at a high-level of abstraction. The VHDL approach allows for direct modeling of the state diagram and thus avoiding dealing with implementation details such as next-state and output decoding logic.

- **ONE HOT ENCODING OF STATE VARIABLES:** The encoding of state variables is an important subject that up to now has been omitted. This chapter briefly covers how the encoding of state variables, and in particular, the use of *One-Hot Encoding* using VHDL models.

Why This Chapter is Important

This chapter is important because it describes a straightforward approach to modeling FSM using VHDL and describes some of the methods used to encode state variables.

[290] Despite this fact, you should have still developed a deep understanding of the general form and function of FSMs. If you did not, you'll most likely experience some trouble in the upcoming material.
[291] This chapter, however, has a high fun factor rating, but not as fun as bowling.

28.2 FSMs Using VHDL Behavioral Modeling

Figure 28.1 shows the block diagram of a standard Moore-type FSM as we worked with in a previous chapter. The *Next State Decoder* is a block of combinatorial logic that uses the current external inputs and the current state of the FSM to decide upon the next state of the FSM. The circuitry in Next State Decoder is generally the excitation equations for the storage elements (flip-flops) in the State Register block. The next state becomes the present state of the FSM when the clock input to the *state registers* block becomes active. The state registers block is storage elements that store the present state of the machine. The *Output Decoder* is yet another combinatorial circuit in the FSM. The inputs to the *Output Decoder* are used to generate the desired external outputs. The inputs to the output decoder are decoded via combinatorial logic to produce the external outputs. We classify this FSM as a Moore-type FSM because the external outputs are only dependent upon the current state of the machine.

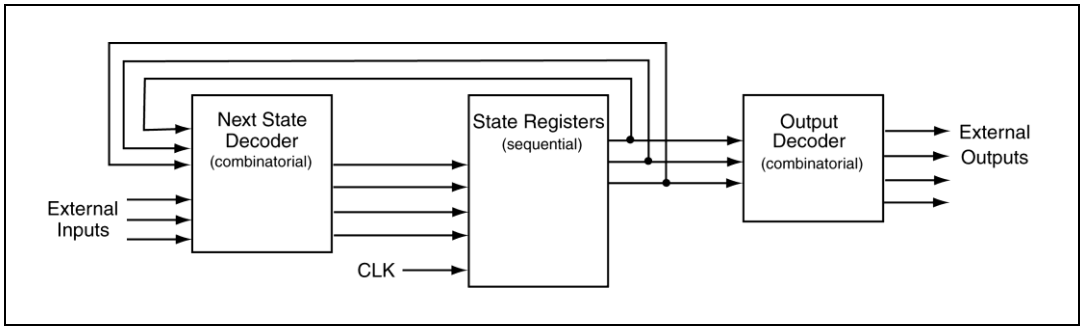


Figure 28.1: Block diagram for a Moore-type FSM.

Although the model in Figure 28.1 accurately describes the FSM in the context of the low-level design techniques we presented in a previous chapter, it does not adequately describe FSMs as we typically model them in VHDL. The true power of VHDL starts to emerge in its dealings with FSMs. As you'll see, the versatility of VHDL behavioral modeling removes the need for large paper designs of endless K-maps and endless combinatorial logic and other boring stuff you may have grown used to in previous chapters.

VHDL uses several different approaches to model FSMs. These many approaches are a result of the versatility of VHDL as a hardware description language. What we'll describe in this section is probably the clearest approach for FSM modeling using VHDL[292]. Figure 28.2 shows a block diagram of the approach we'll use for FSM behavioral modeling using VHDL.

Although it does not look that much clearer, you'll soon find the FSM model in Figure 28.2 to be a straightforward method to implement FSMs. The approach we use divides the FSM into two VHDL processes. One process, referred to as the *Synchronous Process*, handles all the matters regarding clocking and other controls associated with the storage elements. The other process, the *Combinatorial Process*, handles all the matters associated with the Next State Decoder and the Output Decoder of Figure 28.1. Recall that the Output Decoder and Next State Decoder blocks in Figure 28.1 use only combinatorial logic.

There is some new lingo used in the description of signals used in Figure 28.2; the outline below describes this new lingo:

[292] There are many sources available describing other approaches to FSM modeling in VHDL. Once you understand the basics presented in this chapter, understanding the other approaches is not a big deal and you are encouraged to seek these out.

- The inputs labeled *Parallel Inputs* signify inputs that act in parallel to each of the storage elements in the FSM. These inputs would include enables, presets, clears, etc. The thought here is that these input types control all of the storage elements as a group (parallel), and not individually. As you see in more complex FSM designs, there are many times where you'll need this level of control of the FSM's storage elements.

- The inputs labeled *State Transition Inputs* include external inputs that control state transitions. Recall that the external inputs to a FSM can have two functions: 1) they control the state transitions, 2) in the case of Mealy machines, they control the values of the external output signals. This FSM model lumps them together into one block[293].

- The Combinatorial Process box uses the *Present State* signals for both next state decoding and output decoding. The diagram of Figure 28.2 also shows that the Present State variables can also serve as outputs to the FSM. The present state variables can also be input to another combinatorial block (such as a generic decoder) for further massaging.

- The *Next State* signals are truly VHDL signals. Their sole purpose is to provide a means for the two processes to communicate. As opposed to the *Present State* signals, the Next State signals shown in Figure 28.2 are not outputs to the outside world.

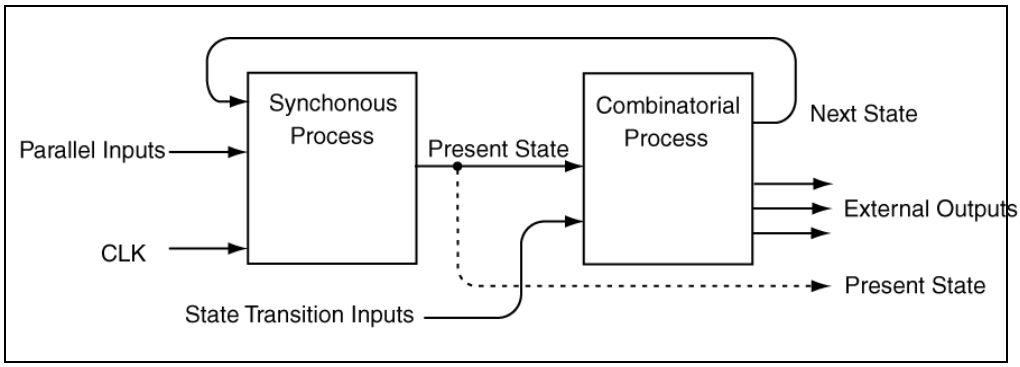


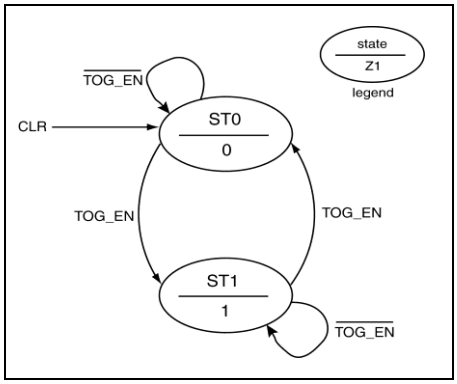
Figure 28.2: Model for VHDL implementations of FSMs.

One final comment before we begin… Although there are many different methods that you can use to model FSMs using VHDL, two of the more common approaches are the are the *dependent* and *independent PS/NS* styles. We've opted to only cover the dependent style in this chapter because it is clearer than the independent PS/NS style when you're first dealing with VHDL behavioral models of FSM. Figure 28.2 shows a model of the dependent PS/NS style of FSMs. One you understand the dependent PS/NS style of VHDL FSM modeling, understanding the independent PS/NS style or any other style is relatively painless. You can find more information on the other FSM coding styles in various VHDL texts or on the web. Keep in mind that if you're modeling FSMs with VHDL, you're handing over a significant amount of control to the VHDL synthesizer, which is generally a happy thing to do.

[293] Recall that we combinatorial process in the current FSM model now has the functionality of both the Next State Decoder and Output Decoder blocks of the previous FSM model.

Example 28-1

Write the VHDL code that models the FSM shown on the right. Use a dependent PS/NS coding style in your implementation.



Solution: This problem represents a basic FSM implementation. It is somewhat instructive to show the black box diagram which serves as an aid in the writing the entity description. Starting design problems by drawing a black box diagram is always a healthy approach particularly when dealing with FSMs. Often times with FSM problems, it sometimes becomes challenging to discern the FSM inputs from the outputs, particularly when the state diagrams become more complex[294]. Drawing a diagram partially alleviates this problem. Figure 28.3 shows a black box diagram for this example while Figure 28.4 shows a solution to this example.

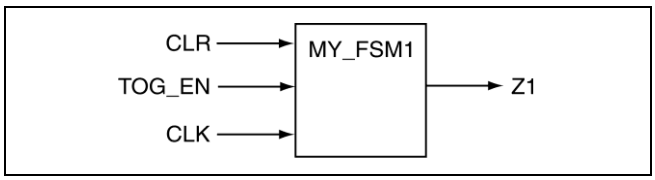


Figure 28.3: Black box diagram for the FSM of Example 28-1.

[294] Also, since there are so many ways to draw state diagrams, you may be dealing with an approach that you're not used to. In this case, draw a black box diagram for sure.

```
entity my_fsm1 is
    port (   TOG_EN : in  std_logic;
            CLK,CLR : in  std_logic;
                 Z1 : out std_logic);
end my_fsm1;

architecture fsm1 of my_fsm1 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     -- take care of the asynchronous input
     if (CLR = '1') then
        PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      Z1 <= '0';        -- pre-assign output
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;

   end process comb_proc;
end fsm1;
```

Figure 28.4: The final solution for Example 28-1.

And of course, this solution has many things worth noting in it. The more interesting things are listed below.

- We've declared a special VHDL *type*, state_type, to represent the states in this FSM. This is an example of how VHDL uses enumeration types. As with enumeration types in other higher-level computer languages, there are internal numerical representations for the listed state types but we only deal with the more expressive textual equivalent. In this case, the type we've created is called a state_type[295] and we've declared two variables of this type: PS and NS (which stand for *present state* and *next state*). The key thing to note here is that a state_type is a type that we've created and is not a native VHDL type.

- The synchronous process is generally equivalent in form and function to the simple D flip-flops we examined when we were dealing with basic storage element representations using VHDL. The only difference is we've substituted PS and NS for D and Q, respectively. The key thing to note here is that the storage element is associated with the PS signal only. Note that PS is not

[295] The name is arbitrary but it does nice describe the purpose of the new type.

specified for every possible combination of inputs (it has no catch-all statement) which is why VHDL models PS as a storage element.

- Even though this is about the simplest FSM you could hope for, the code looks somewhat complicated. However, if you examine it closely, you can see that the solution nicely compartmentalizes everything. There are two processes. The synchronous process handles the asynchronous reset and the assignment of a new state upon the arrival of the system clock. The combinatorial process handles the outputs not handled in the synchronous process, the outputs, and the generation of the next state of the FSM.

- Because the two processes operate concurrently, they are working in a lock-step manner. Changes to the NS signal generated in the combinatorial process forces an evaluation of the synchronous process because NS is in the sensitivity list of the combinatorial process. When the synchronous process institutes those changes on the next clock edge, the changes in the PS signal causes a new evaluation of the combinatorial process because PS is in the sensitivity list of the combinatorial process. And so on and so forth.

- The case statement in the combinatorial process provides a ***when*** clause for each individual state of the FSM. This is the standard approach for the dependent PS/NS coding style. A ***when others*** clause is also provided; the signal assignments that are part this catch-all clause is arbitrary since the code should never actually make it there. This statement provides a sense of completeness and represents good VHDL coding practice[296]. In reality, catch-all statements are quite useful for debugging purposes as you can force signals to be at strange values if they make it to parts of the VHDL model that they should not be in. Thus, if you see these strange values on a Logic Analyzer or simulator output, you know right where to look to fix the problem.

- The Moore output is a function of only the present state. This is expressed by the fact that the assignment of the Z1 output is unconditionally evaluated in each ***when*** clause of the case statement in the combinatorial process. In other words, the Z1 variable is inside the ***when*** clause but outside of the ***if*** statement in the when clause. This is because the Moore outputs are only a function of the present and not the external inputs. Note that the external input that controls how the FSM transitions from a given state. You'll see later that Mealy outputs, due their nature, are assigned inside the ***if*** statement.

- The first step of the combinatorial process is to pre-assigned the Z1 output. Pre-assigning it in this fashion prevents the unexpected latch generation for the Z1 signal. When dealing with FSMs, there is a natural tendency for the FSM designer to forget to specify an output for the Z1 variable in each of the states. Pre-assigning these outputs helps prevents latch generation and can arguably make the VHDL source code seem neater. The pre-assignment does not change the function of the VHDL model because if the process makes multiple assignments within the process, only the final assignment takes affect when the process evaluation completes. In other words, only the final assignment is effective once the process terminates[297].

There is one final thing to note about the solution shown in Figure 28.4. In an effort to keep the example simple, we disregarded the true digital values of the state variables. The black box diagram of Figure 28.3 indicates the fact that the only output of the FSM is signal Z1. This is reasonable in that it could be the case where only one output was required in order to control some other device or circuit. The state variables have an internal representation so the precise representation of the state variables is not important since the FSM does not provide them as output. So if someone were to ask you how many

[296] Catch-all statements used in this manner are also helpful when used to debug VHDL models.

[297] Keep in mind that process statement is a concurrent statement. Despite the fact that process statements are filled with sequential statements, the "execution" of the process statement occur concurrently. It's a tough concept, but you'll get used to it with practice and contemplation.

flip-flops were associated with a VHDL model such as this one, you would not know. Once again, this is because we used a VHDL enumerated type to represent the state variables and have thus handed control over to the VHDL synthesizer[298].

Some FSM designs use the state variables as outputs. To show this situation, we'll provide a solution to Example 28-1 where the state variables are "modeled" as outputs. Figure 28.5 shows the black box diagram of this solution while Figure 28.6 shows the alternate VHDL model.

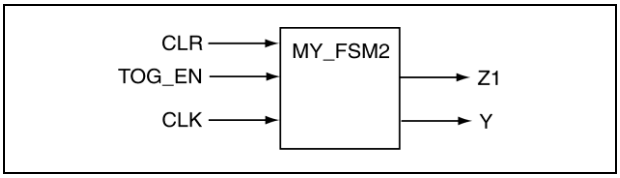


Figure 28.5: Black box diagram of Example 28-1 including the state variable as an output.

[298] In reality, this is a problem of "how to encode the state variables". This is a giant issue as there are many ramifications regarding how exactly you encode the state variables. But, this is less of an issue in a beginning text such as this one because modeling things to be massively efficient is simply a more advanced concept and is beyond the scope of this text.

```
entity my_fsm2 is
    port (   TOG_EN : in  std_logic;
            CLK,CLR : in  std_logic;
               Y,Z1 : out std_logic);
end my_fsm2;

architecture fsm2 of my_fsm2 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     if (CLR = '1') then
        PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      case PS is
      Z1 <= '0';

         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;
   end process comb_proc;

   -- assign values representing the state variables
   with PS select
      Y <= '0' when ST0,
           '1' when ST1,
           '0' when others;

end fsm2;
```

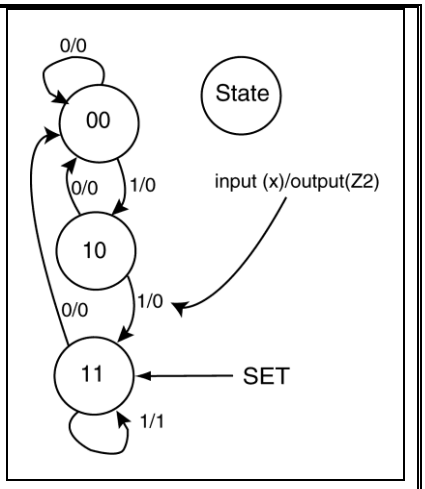
Figure 28.6: Solution for Example 28-1 including state variable as an output.

Note that the VHDL code in shown in Figure 28.6 differs in only two areas from the code shown in Figure 28.4. The first area is the modification of the entity declaration to account for the state variable output Y. The second area is the inclusion of the selective signal assignment statement, which assigns a value of state variable output Y based on the condition of the state variable. The selective signal assignment statement evaluates each time the PS signal changes.

Once again, since we have declared an enumeration type for the state variables, we have no way of knowing exactly how the synthesizer has opted to represent the state variable. The selective signal assignment statement in the code of Figure 28.6 only makes it appear as if we used only one state variable to represent the two states shown in the original state diagram. In reality, there are methods we can use to control how the state variables are represented and we'll deal with those soon. Lastly, be sure to note that there are three concurrent statements in the VHDL code shown in Figure 28.6: two process statements and a selective signal assignment statement.

Example 28-2

Write the VHDL code that implements the FSM shown on the right. Use a dependent PS/NS coding style in your implementation. Consider the state variables as outputs of the FSM.



Solution: The state diagram shown in the problem description indicates that this is a three-state FSM with one Mealy-type external output (Z2) and one external input (X). Since there are three states, the solution requires at we use at least two signals to model the state variables, which is sufficient to handle the three states. These state variables are for output purposes only, as the VHDL synthesizer handles the true state variable representation. Figure 28.7 shows the black box diagram for this example while Figure 28.7 shows the full solution. Note that the two state variables are handled as a bundled signal, which is arbitrary.

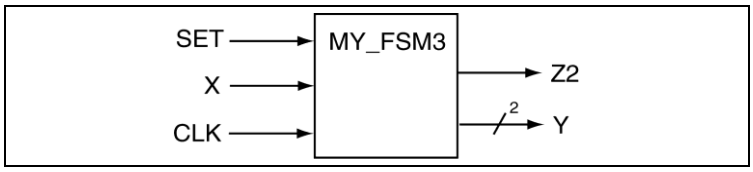


Figure 28.7: Black box diagram for the FSM of Example 28-2.

```
entity my_fsm3 is
    port ( X,CLK,SET : in  std_logic;
                   Y : out std_logic_vector(1 downto 0);
                  Z2 : out std_logic);
end my_fsm3;

architecture fsm3 of my_fsm3 is
   type state_type is (ST0,ST1,ST2);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,SET)
   begin
     if (SET = '1') then
        PS <= ST2;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      case PS is
         Z2 <= '0';     -- pre-assign FSM outputs
         when ST0 =>    -- items regarding state ST0
            Z2 <= '0';  -- Mealy output always 0
            if (X = '0') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z2 <= '0';  -- Mealy output always 0
            if (X = '0') then NS <= ST0;
            else  NS <= ST2;
            end if;
         when ST2 =>    -- items regarding state ST2
            -- Mealy output handled in the if statement
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST2;  Z2 <= '1';
            end if;
         when others => -- the catch all condition
             Z2 <= '1'; NS < ST0;
      end case;
   end process comb_proc;

   -- faking some state variable outputs
   with PS select
      Y <= "00" when ST0,
           "10" when ST1,
           "11" when ST2,
           "00" when others;
end fsm3;
```

Figure 28.8: Solution for Example 28-2.

As usual, there are a couple of fun things to note about the solution for Example 28-2. Most importantly, you should note the similarities between this solution and the solution to the previous example.

- The FSM has one Mealy-type output. The solution essentially treats this output as a Moore-type output in the first two ***when*** clauses of the ***case*** statement. In the final ***when*** clause, the Z2 output appears in both sections of the ***if*** statement. The fact that the Z2 output is different in state ST2 (depending on the value of the X input) makes it a Mealy-type output and therefore a Mealy-type FSM. This is always something you should be aware of; it's easy to think that since the VHDL

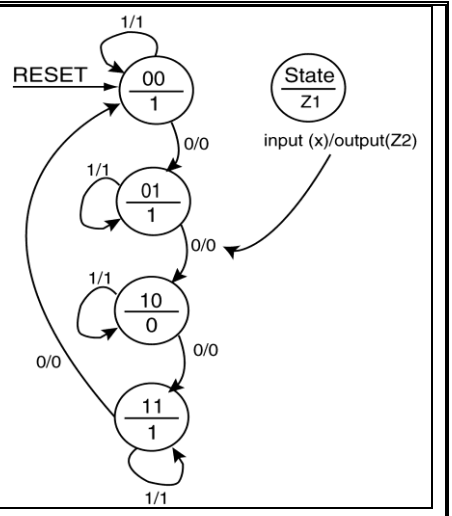
model is using an ***if*** statement in the when clause for the state, it's automatically going to be a Mealy type external output. Never make this assumption.

- This original state diagram has a mysterious state transition arrow. The SET arrow on the lower right of the state diagram seems to enter the lower state out of nowhere. State diagrams use this symbology to indicate asynchronous inputs to FSMs. This input is a "parallel input" as previously described since the SET signal simultaneously acts on all of the storage elements. Modeling the SET signal in the synchronous process uses the same approach as modeling asynchronous inputs in basic storage elements. Therefore, the approach used to model the SET signal is nothing new and strange.

- When faking the state variable outputs (keeping in mind that enumeration types represent the actual state variables), two signals are required since the state diagram contains more than two states (and less than five states). The solution opted to represent these outputs as a bundle, which has the effect of slightly changing the form of the selected signal assignment statement appearing at the end of the architecture description.

Example 28-3

Write the VHDL code that models the FSM shown on the right. Use a dependent PS/NS coding style in your model. Consider the listed state variables as output.



Solution: The state diagram indicates that the solution contains four states, one external input, and two external outputs. This is a hybrid FSM in that the ***if*** contains both a Mealy and Moore-type output but in this case, the FSM would be considered a Mealy-type FSM. Figure 28.9 shows the black box diagram for the solution while Figure 28.10 shows the associated VHDL model.

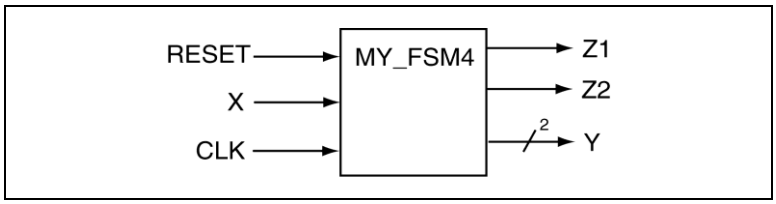


Figure 28.9: Black Box diagram for the FSM of Example 28-3.

```
entity my_fsm4 is
    port ( X,CLK,RESET : in  std_logic;
                       Y : out std_logic_vector(1 downto 0);
                  Z1,Z2 : out std_logic);
end my_fsm4;

architecture fsm4 of my_fsm4 is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then  PS <= ST0;
     elsif (rising_edge(CLK)) then  PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      -- Z1: the Moore output; Z2: the Mealy output
      Z1 <= '0';   Z2 <= '0'; -- pre-assign the outputs
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST1; Z2 <= '0';
            else  NS <= ST0; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all condition
            Z1 <= '1';  Z2 <= '0';  NS <= ST0;
      end case;
   end process comb_proc;

   with PS select
      Y <= "00" when ST0,
           "01" when ST1,
           "10" when ST2,
           "11" when ST3,
           "00" when others;
end fsm4;
```

Figure 28.10: Solution for Example 28-3.

So if you've haven't noticed by now, implementing FSMs using VHDL behavioral modeling is remarkably straightforward. It's actually a cookbook approach it's so straightforward. In reality, you'll rarely find yourself having to code VHDL FSM models from scratch. The better approach is to grab a previously coded model and use that. In other words, there is no need to reinvent the wheel when using VHDL to model FSMs: use the cut and paste[299] feature of your text editor instead. Keep in mind that

[299] Particularly useful if you're a CPE: "cut and paste engineer".

real engineering is rarely cookbook and therefore modeling FSM using VHDL is not true engineering. For FSM problems, the engineering is in the testing and creation of the state diagram. So don't get too comfortable with behavioral modeling of FSMs; the real fun is generating a FSM that solves a given problem which we'll start doing in the next chapter.

Finally, Figure 28.11 shows a copy of the famous VHDL FSM implementation cheat sheet. I use this when I'm unable to find an old VHDL FSM model that I can and paste from.

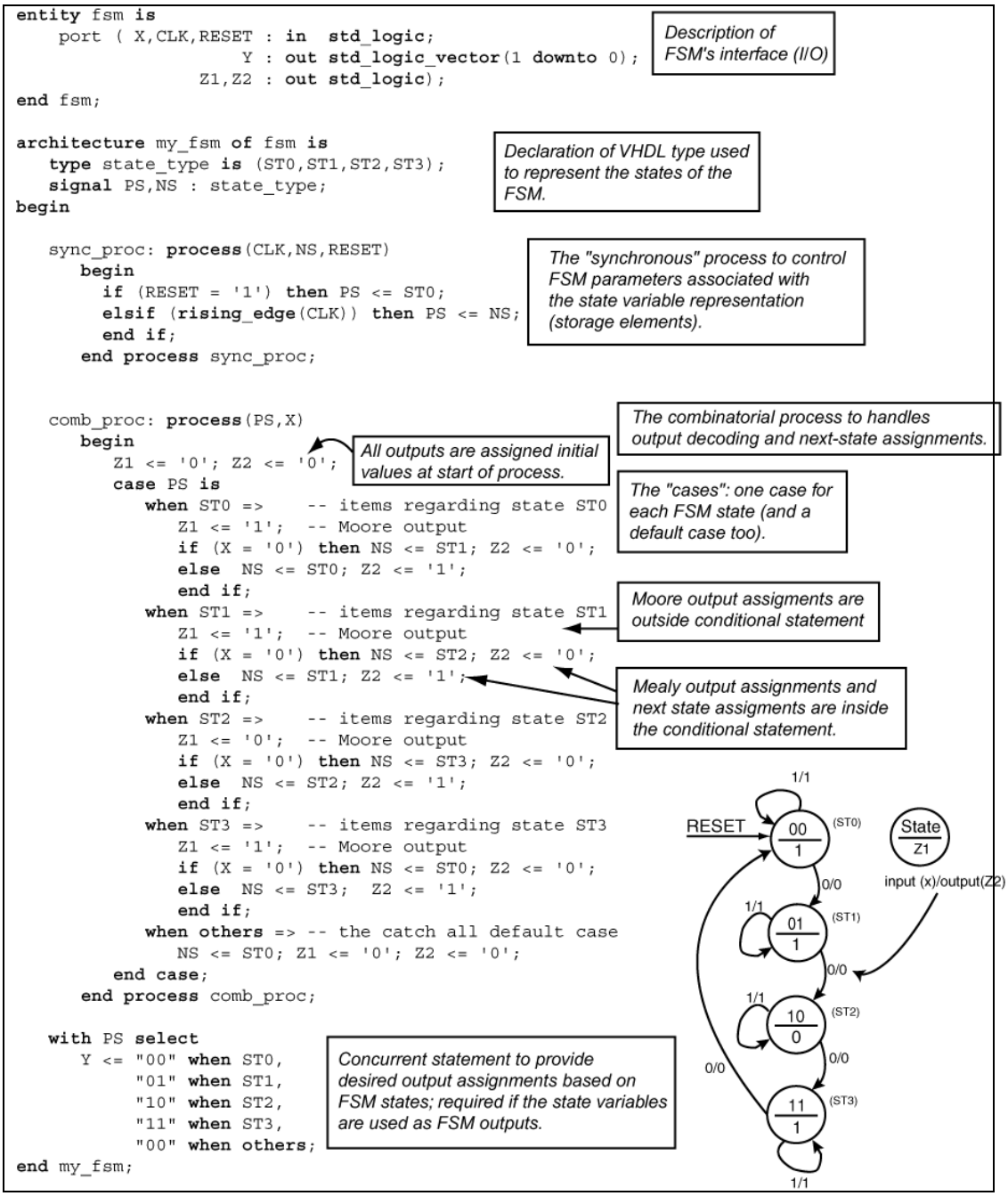


Figure 28.11: Possibly the ultimate VHDL FSM behavioral modeling cheat sheet.

28.3 State Variable Encoding and One-Hot Encoding

In our quest to conquer the wild FSM, we did not pay much attention to the 1's and 0's used to represent the state variables. Our only concern in this matter was to have a unique state variable assignment for each state in the FSM. The only thought you may have given to deciding upon what state variables to use was to use as few as possible in an effort to reduce the complexity of the resulting

hardware (if you actually had to implement the FSM). By using as few state variables as possible, you minimized the amount of hardware dedicated to storing the state variables (thus minimizing the number of flip-flops). This is a viable approach, and it was especially more viable in days where people actually implemented FSMs using discrete ICs. This focus on minimizing hardware is somewhat less the case today in non-academic environments, particularly with the current violent revolution in PLDs, and particularly FPGAs.

You can actually use many different approaches to assign state variables. While is comfortable to make just any assignment and have the world be happy (which is generally what is done in academia), the reality is that there is much more involved. You may someday be lucky enough to take a course or to learn these underlying details from your employer, but for now, we can continue not putting much thought or effort into the topic[300]. For now, we need to look at one other type of state variable assignment since it has deep ramifications in the realm of FPGA-based FSM implementations.

28.3.1 Binary and One-Hot Encoding of State Variables

The approach we've been using up until now of implementing FSMs was to use as few flip-flops as possible to encode the state variables. This approach to state variable encoding is referred to as *binary* or *full encoding*. In this approach, as the name implies, the state variable assignments use binary-type values[301]. This is the typical approach taken in introductory FSM courses. In this approach, there is an intuitively binary relation between the number of flip-flops (bit storage elements) and the number of states in the FSM.

There are two ways to look at the flip-flops vs. states relationship as shown by the two relationships of Equation 28-1. These equations show the relationship between the number of states and the number of flip-flops required to uniquely encode the states when using binary encoding. The operator used in the right-hand equation is the *ceiling* function which is defined as the smallest integer greater than or equal to its argument.

$$2^{\#\text{flip-flops}} \geq \#\text{states} \qquad \#\text{flip-flops} = \log_2\lceil \#\text{states} \rceil$$

Equation 28-1: Relating the number of states and the number of flip-flops for binary encoding.

Many the approaches you can use to encode state variables are based upon the availability of hardware. Back in the days when people commonly used discrete ICs to implement FSMs, common approaches were to use hardware such as counters, shift registers, ROMs). Each of these approaches typically had some advantage or disadvantage over the other approaches. Some of these approaches include gray codes, unit distance codes, sequential codes, Johnson counts, twisted-ring counts, and one-hot encoding. We'll only be looking at one-hot encoding since it is well suited for use implementing FSMs on FPGAs.

One-hot encoding places no emphasis on minimizing the number of flip-flops as was a characteristic of binary encoding. One-hot encoding instead places emphasis on minimizing the next-state logic. The contrast here when compared to binary encoding is where in the FSM that the hardware is applied. In

[300] When FSMs are pushed to their operational limits by fast clock speeds, strange stuff can happen. This strangeness can somewhat be controlled by an intelligent assignment of state variables. If you know the black magic of generating intelligent state variable assignments, your FSM will operate more robustly at high clock speeds.

[301] Or to put it more precisely, binary values were used when the number of states requiring representation was a power of two. If the number of states is not a power of two, some subset of a full set of binary values was used.

binary coding, the hardware used to encode the state variables is minimal, but this minimization comes at the cost of creating more logic in the excitation equations. In one-hot encoding, the hardware used to encode the state variables is not minimized but it has the overall effect of minimizing the next-state logic. This is a definite trade-off; there's a lot more to say about this that we won't say here.

Equation 28-2 shows the closed-form relationship between the number of flip-flops and the number of states in one-hot encoding. As indicated by the equation below, there is one flip-flop for each state in the FSM. Pretty straightforward, huh?

# of flip-flops = # of states

Equation 28-2: Relating the number of states and the number of flip-flops for one-hot encoding.

One-hot encoding differs from binary encoding in that it has constraints on both the characteristics and number of state variables. In a valid one-hot state, only *one* of the flip-flops can be in a '1' state at any given time, which is how the name *one-hot* is derived. Table 28.1 shows the valid one-hot encoded state assignments for a few different state counts. It should be evident from Table 28.1 that it's fairly simple to extrapolate the one-hot code for any number of states. The ordering of the individual codes makes no difference; the important feature of these codes is the fact that only one state variable is a '1' for all the states in any codeword.

Though it sounds like a cool name, there is nothing complex about one-hot encoding. Note that the codes look a lot like the active-high outputs of a standard decoder. The reality is that in using one-hot encoding, you are increasing the number of flip-flops required to implement the design as opposed to the number of flip-flops that binary encoding would use for the same FSM. This sounds bad, very bad. However, the payoff is that you'll also see a reduction in the amount of logic required to implement the next state decoder.

Finally, FPGAs have a relatively large number of flip-flops that are readily available for state variable representation. In this case, using one-hot encoding actually does reduce the overall size of the circuit. So, all is not bad in FSM-land. The example in Table 28.1 shows that implementing designs using one-hot encoded state variables is nothing overly complicated.

# of states in FSM	Valid Codewords (valid one-hot state assignments)
2	"10", "01"
3	"100", "010", "001"
4	"1000", "0100", "0010", "0001"
5	"10000", "01000", "00100", "00010", "00001"

Table 28.1: Valid one-hot encoded state variable assignments for state counts (2-5).

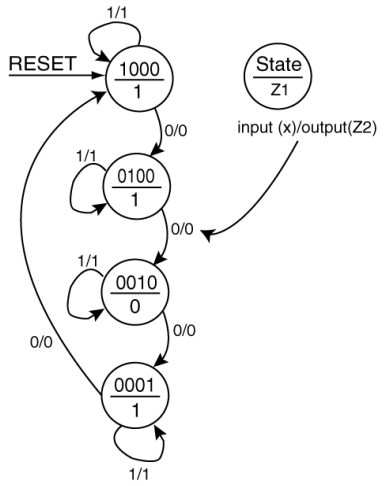
28.4 VHDL Topics: One-Hot Encoding in FSM Behavioral Modeling

The question naturally arises as to how VHDL implements one-hot encoded FSMs. If you want total control of the process, you'll need to grab control away from the synthesizer. In addition, since we're concerned with learning VHDL, we need to look at the process of explicitly encoding one-hot FSMs. The other good thing to note here is that this discussion is not really about the one-hot encoding of FSM state variables; it's actually about having the ability to encode the state variables any way you please. One-hot encoding just happens to be a popular approach to encoding state variables particularly suite to PLD-based implementations[302].

The modular approach we used to implement FSMs expedites the conversion process from using enumeration types to actually specifying the representation of the state variables. The changes required from our previous approach are limited how the VHDL places constraints on the encoding of the state variables. Modifications to the binary encoded approach are thus limited to a few lines in the VHDL state variable declaration process. If you need to provide the state variables as outputs, the entity declaration (you'll need more variables to represent the states) and the VHDL code controlling the assignment of output variables will also need modification.

Example 28-4

Write the VHDL code that implements the FSM modeled by the state diagram shown on the right. Use a dependent PS/NS coding style in your implementation. Consider the listed state variables as output. Use one-hot encoding for the state variables; provide the state variables as outputs to the FSM. This problem is identical to a previous example but one-hot encoding for the state variables in this example.



Solution: The state diagram shows four states, one external input X, two external outputs Z1 and Z2 with the Z2 output being a Mealy output. This is a Mealy machine and the state diagram indicates that the state variables must be encoded using one-hot encoding. We'll approach this solution in pieces, bits and pieces.

Figure 28.12 shows the modifications to the entity declaration required to convert the binary encoding used in standard VHDL behavioral modeling to one-hot encoding. Figure 28.13 shows the required modifications to the state variable output assignment in order to move from enumeration types to a special form of assigned types. Forcing the state variables to be truly encoded using one-hot encoding requires these two extra lines of code as is shown in Figure 28.13.

Most synthesis tool vendors provide the ENUM_ENCODING attribute to allow the digital designer to specific the binary encoding to that is used by each object of enumerated types. These two lines of code essentially force the VHDL synthesizer to represent each state of the FSM with its own storage element.

[302] Once again, the notion of how to encode the state variables is a concept involved with advanced FSM design and is not covered here.

In other words, the VHDL code shows that each state is represented by the associated "string" modifier. In this particular example, the code essentially forces the FSM implementation to remember four bits per state, which essentially requires four flip-flops. You should strongly consider comparing and contrasting these three figures. Figure 28.14 shows the total solution for this example.

```
-- full encoded approach
entity my_fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                Z1,Z2 : out std_logic);
end my_fsm;
```

```
-- one-hot encoding approach
entity my_fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(3 downto 0);
                Z1,Z2 : out std_logic);
end my_fsm;
```

Figure 28.12: Modifications to convert entity associated with Error! Reference source not found. **to one-hot encoding.**

```
-- the approach to for enumeration types
type state_type is (ST0,ST1,ST2,ST3);
signal PS,NS : state_type;
```

```
-- the approach used for explicitly specifying state bit patterns
type state_type is (ST0,ST1,ST2,ST3);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
signal PS,NS : state_type;
```

Figure 28.13: Modifications to convert state variables to use one-hot encoding.

```
entity my_fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(3 downto 0);
                 Z1,Z2 : out std_logic);
end my_fsm;

architecture fsm of my_fsm is
   type state_type is (ST0,ST1,ST2,ST3);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then  PS <= ST0;
     elsif (rising_edge(CLK)) then  PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      -- Z1: the Moore output; Z2: the Mealy output
      Z1 <= '0';   Z2 <= '0'; -- pre-assign the outputs
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST1; Z2 <= '0';
            else  NS <= ST0; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all condition
            Z1 <= '1';  Z2 <= '0';  NS <= ST0;
      end case;
   end process comb_proc;

   -- one-hot encoded approach
   with PS select
      Y <= "1000" when ST0,
           "0100" when ST1,
           "0010" when ST2,
           "0001" when ST3,
           "1000" when others;
end fsm;
```

Figure 28.14: The final solution to Example 28-4.

Chapter Summary

- Modeling FSMs from a state diagram is a straightforward process using VHDL behavioral modeling. The process is so straightforward that it is cookie cutter. The real engineering involved in implementing FSM is in the generation of the state diagram that solves the problem at hand.

- Due to the general versatility of VHDL, you can use many approaches to model FSMs using VHDL. The approach used in this chapter is referred to as the dependent style of FSM model. This approach used two processes to model FSM behavior: one process for the sequential elements of an FSM and one process for the combinatorial elements.

- The actual encoding of the FSM's state variables when enumeration types are used is left up to the synthesis tool. If you prefer some particular method of variable encoding, using the attribute approach detail in this section is a simple but viable alternative.

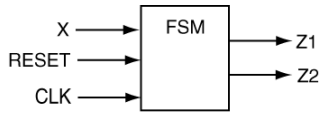
- When enumeration types are used to represent state variables, external outputs of FSM can be assigned by including concurrent signal assignment statements in the FSM model. Although you can simulate state variables this approach, the VHDL synthesizer has the ultimate control as to how the state variables are actually assigned.

- There are VHDL constructs available to force the VHDL synthesizer to encode FSM behavioral models using a one-hot code. These constructs allow the digital designer to directly control the state variable encoding. Generally speaking, there are options that can be chosen in the synthesizer toolset that allow the user to indirectly select the exact flavor of state variable encoding; there are many standard flavors to choose from though only two were discussed in this chapter.

- You can encode the state variables of FSMs using many different styles. The "binary" encoding approach minimized the number of flip-flops required to implement a given FSM while the "one-hot" encoding approach minimized the next-state decoding logic at the expense of using more flip-flops.

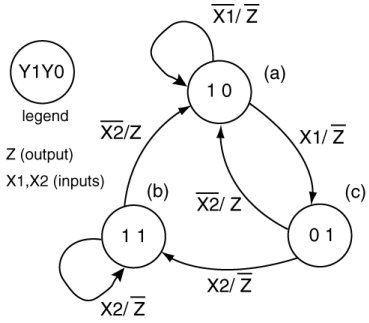
Chapter Problems

1) Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and completely label everything.

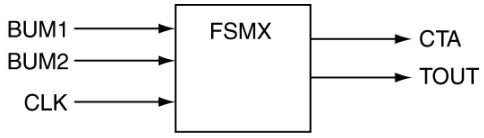



```
entity fsm is
   port (  X,CLK : in  std_logic;
           RESET : in  std_logic;
           Z1,Z2 : out std_logic;
end fsm;

architecture fsm of fsm is
   type state_type is (A,B,C);
   signal PS,NS : state_type;
begin

   sync_proc: process(CLK,NS,RESET)
   begin
      if (RESET = '0') then PS <= C;
      elsif (rising_edge(CLK)) then PS <= NS;
      end if;
   end process sync_proc;

   comb_proc: process(PS,X)
   begin
      case PS is
         Z1 <= '0';    Z2 <= '0';

         when A =>
            Z1 <= '0';
            if (X = '0') then NS <= A; Z2 <= '1';
            else  NS <= B; Z2 <= '0';
            end if;

         when B =>
            Z1 <= '1';
            if (X = '0') then NS <= A; Z2 <= '0';
            else  NS <= C; Z2 <= '1';
            end if;

         when C =>
            Z1 <= '1';
            if (X = '0') then NS <= B; Z2 <= '1';
            else  NS <= A; Z2 <= '0';
            end if;

         when others =>
            Z1 <= '1'; NS <= A; Z2 <= '0';

      end case;
   end process comb_proc;

end fsm;
```

2) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.

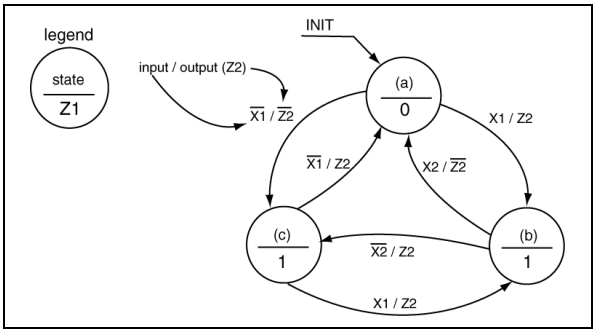


3) Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and completely label everything.




```
entity fsmx is
    Port ( BUM1,BUM2 : in std_logic;
                 CLK : in std_logic;
           TOUT,CTA : out std_logic);
end fsmx;


architecture my_fsmx of fsmx is
   type state_type is (S1,S2,S3);
    signal PS,NS : state_type;
begin
   sync_p: process (CLK,NS)
   begin
      if (rising_edge(CLK)) then
         PS <= NS;
      end if;
   end process sync_p;

   comb_p: process (CLK,BUM1,BUM2)
   begin
      case PS is

         when S1 =>
            CTA <= '0';
            if (BUM1 = '0')  then
               TOUT <= '0';
               NS <= S1;
            elsif (BUM1 = '1') then
               TOUT <= '1';
               NS <= S2;
            end if;

         when S2 =>
            CTA <= '0';
            TOUT <= '0';
            NS <= S3;

         when S3 =>
            CTA <= '1';
            TOUT <= '0';
            if (BUM2 = '1')  then
               NS <= S1;
            elsif (BUM2 = '0') then
               NS <= S2;
            end if;

         when others =>  CTA <= '0'; TOUT <= '0'; NS <= S1;
      end case;
   end process comb_p;

end my_fsmx;
```

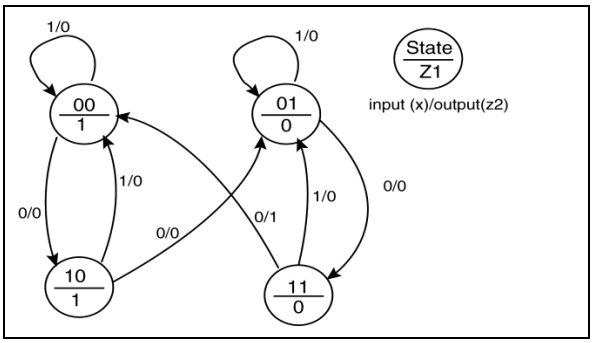
4) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right.



5) Draw the state diagram associated with the following VHDL code. Consider the outputs Y to be representative of the state variables. Be sure to provide a legend. Indicate the states with both the state variables and their symbolic equivalents.

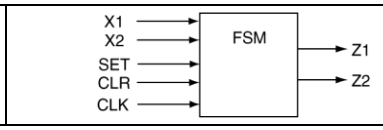
```
entity fsm is
port (     X,CLK : in  std_logic;
           RESET : in  std_logic;
           Z1,Z2 : out std_logic;
               Y : out std_logic_vector(2 downto 0));
end fsm;

architecture my_fsm of fsm is
   type state_type is (A,B,C);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type is "001 010 100";
   signal PS,NS : state_type;
begin

sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '0') then PS <= C;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;

comb_proc: process(PS,X)
   begin
      case PS is
         when A =>
            Z1 <= '0';
            if (X = '0') then NS <= A; Z2 <= '1';
            else  NS <= B; Z2 <= '0';
            end if;
         when B =>
            Z1 <= '1';
            if (X = '0') then NS <= A; Z2 <= '0';
            else  NS <= C; Z2 <= '1';
            end if;
         when C =>
            Z1 <= '1';
            if (X = '0') then NS <= B; Z2 <= '1';
            else  NS <= A; Z2 <= '0';
            end if;
         when others =>
            Z1 <= '1'; NS <= A; Z2 <= '0';
      end case;
   end process comb_proc;

with PS select
   Y <= "001" when A,
        "010" when B,
        "100" when C,
        "001" when others;
end my_fsm;
```

6) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. Encode the state variables as listed and also provided them as outputs of the FSM.



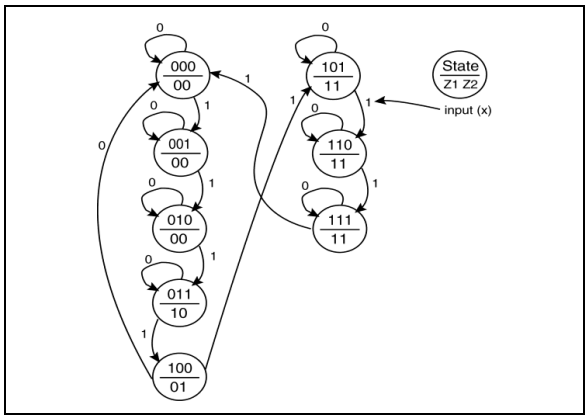
7) Draw the state diagram that corresponds to the following VHDL model and s*tate whether the FSM is a **Mealy** or **Moore** machine*. Be sure to label everything.

```
entity fsm is
    Port ( CLK,CLR,SET,X1,X2 : in std_logic;
                       Z1,Z2 : out std_logic);
end fsm;
```

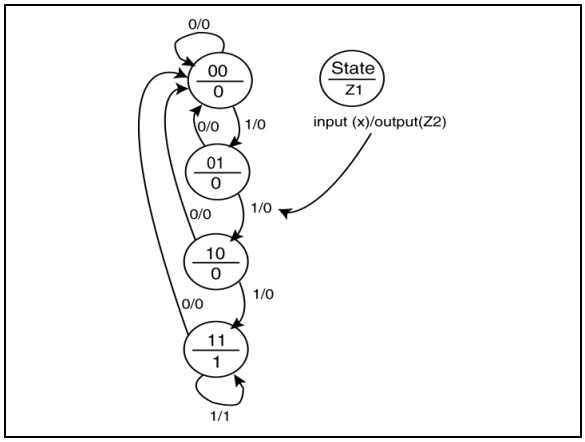



```
architecture my_fsm of fsm is
   type state_type is (sA,sB,sC,sD);
   attribute ENUM_ENCODING: STRING;
   attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
   signal PS,NS : state_type;
begin
   sync_p: process (CLK,NS,CLR,SET)
   begin
      if (CLR = '1' and SET = '0') then
         PS <= sA;
       elsif (CLR = '0' and SET = '1') then
          PS <= sD;
      elsif (rising_edge(CLK)) then
         PS <= NS;
      end if;
   end process sync_p;

   comb_p: process (X1,X2,PS)
   begin
      case PS is
         when sA =>
            if ( X1 = '1')  then
               Z1 <= '0'; Z2 <= '0';
               NS <= sA;
            else
               Z1 <= '0'; Z2 <= '0';
               NS <= sB;
            end if;

         when sB =>
            if ( X2 = '1')  then
               Z1 <= '1'; Z2 <= '1';
               NS <= sC;
            else
               Z1 <= '1'; Z2 <= '0';
               NS <= sB;
            end if;

         when sC =>
            if ( X2 = '1')  then
               Z1 <= '0'; Z2 <= '0';
               NS <= sB;
            else
               Z1 <= '0'; Z2 <= '1';
               NS <= sC;
            end if;

         when sD =>
            if ( X1 = '1')  then
               Z1 <= '1'; Z2 <= '1';
               NS <= sD;
            else
               Z1 <= '1'; Z2 <= '1';
               NS <= sC;
            end if;
      end case;

   end process comb_p;
end my_fsm;
```

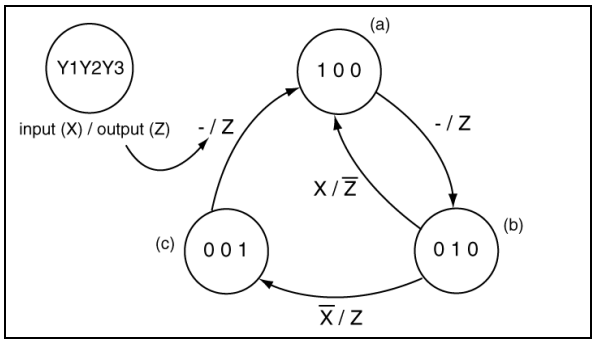
8) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



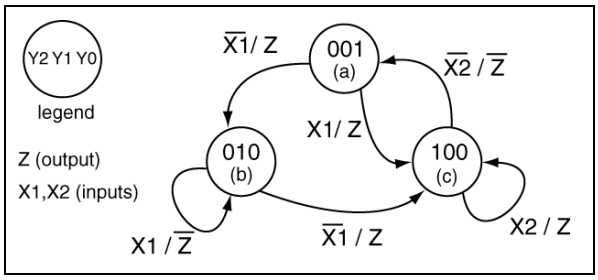
9) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



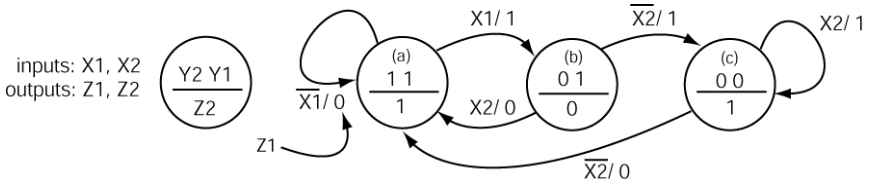
10) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



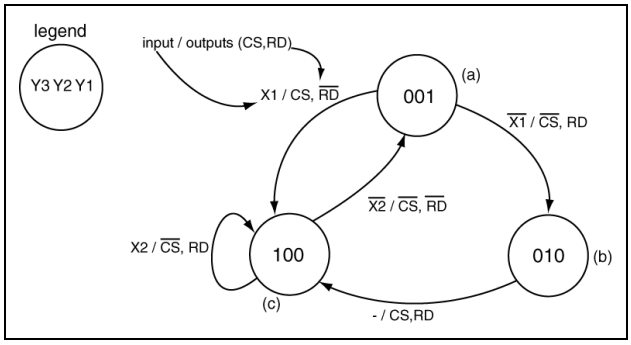
11) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



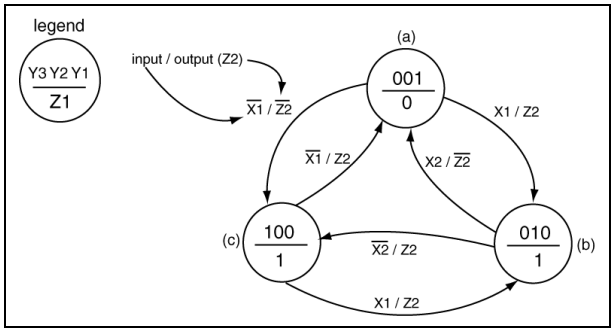
12) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



13) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



14) Write a VHDL behavioral model that could be used to implement the state diagram on shown in the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



29 Chapter Twenty-Nine

(Bryan Mealy 2012 ©)

29.1 Chapter Overview

Implementing FSMs from a given state diagram was one of our initial focuses when dealing with FSMs. The skills required to implement the FSM with either discrete logic components (gate-level implementations) or VHDL behavioral models was not overly challenging once you did a few examples[303]. Now that we have conquered the mechanics of FSM implementation, we'll change our focus to state diagram generation. Afterall, it's the generation of the state diagram that represents 99.9% of the engineering involved in FSM design. The idea here is that anyone can implement a FSM from a given state diagram while it takes a 100% understanding of FSMs and the problem at hand in order to generate a state diagram for a given problem.

This chapter provides an intuitive look at state diagrams and their associated timing diagrams. Having an intuitive feel for state diagrams and being familiar with the mechanics of implementing FSM, you'll be ready to handle just about any control problem known to both humans and academic administrators alike. Some of the presented information will seem like review, but it should help solidifying the various concepts associated with FSMs.

Main Chapter Topics

- **THE BASICS OF STATE DIAGRAMS:** This chapter presents an intuitive view of all aspects of state diagram. The key to generating state diagrams is understanding the basic state diagram symbology and terminology and how they relate to designing a solution to solving the problem at hand.
- **FSM Problem Solving**: This chapter introduces basic state diagram generation in the context of sequence detectors. Sequece detectors provided relatively simple problems to understand which allows you to focus your efforts on generating the associated state diagram.

Why This Chapter is Important

This chapter is important because it describes the low level details of representing state diagrams and also the differences in timing diagrams associated with Mealy and Moore-type FSM.

[303] But it was a giant pain in the ass.

29.2 The Big FSM Picture

The world progressed nicely for billions of years without having the concept of finite state machines or any other such important items. In recent history, we've developed a need for low-level control of just about everything in our lives, particularly control by tiny electronic things. In regards to FSMs, the following verbage provides an overview of the control path that has led us to where we are today (a few details are missing):

- In relatively recent history, digital stuff (computers and things) started happening. It actually started happening a long time ago, but until relatively recently, the cost of digital stuff was such that the average human could not afford to take notice.

- All the new digital stuff required some circuitry to control it; FSMs were the logical[304] option. The thought here is that maybe you could use a computer to control a computer, but, these were still the days where computers were actually expensive (and big) and had names like "HAL". The main problem with software-based control was that the required software increased the complexity of your project (and thus the length of the program) and increased the overall memory requirements of your design[305]. In order to deal with these issues, these projects used FSM implemented with discrete components to control things. This approach was a pure hardware approach that could be implemented with readily available and relatively inexpensive hardware. This was good.

- Integrated circuits (ICs) started taking over. There generally had been many ICs out there, but all of a sudden, there were many more ICs out there. These new ICs were providing more complex functionality, which meant that some of the control functions handles by FSMs were being built into the various ICs. There were also ICs dedicated to controlling specific devices, which were essentially required because devices were becoming complex and control requirements were growing in complexity. In other words, if you could purchase an IC that controlled some other digital device, you would not need to design a FSM that controlled the device.

- Microcontrollers (MCUs) started becoming prevalent[306]. This meant that the FSM-controlled hardware was now controllable by MCUs. This meant that hardware devices could now essentially be under software control (what drives the MCU) rather than hardware (what FSMs are constructed from). The upside of this software control is the flexibility in software (namely its re-programmability characteristic). The downside is that using the MCU to control hardware requires processing time from the MCU or dedicating an entire MCU to the control task. This option also requires someone who possesses the skill to design and program a MCU-based system. Although MCUs nicely handle some control tasks, they are not appropriate for all such tasks, particulary as the number of control tasks in a given system increase.

- Programmable Logic Devices (PLDs) such as FPGAs and CPLDs hit the market and became BIG (in size at least). This means that you could use the PLD to handle logic functions required by your circuit. Since you may have already been using a PLD, it makes sense to use that PLD to implement a FSM while you're at it. And what the heck, you could use the PLD to implement the entire MCU (known as a soft-core MCU). In

[304] A true play on words.

[305] Keep in mind that back in these days, memory was much more expensive than it was today.

[306] They had actually been around for awhile, but they were now less expensive. More importantly, the development environments (primarily PC-based) and associated CAD tools were significantly less expensive also.

other words, transferring control from the MCU to the FSM was no longer too costly. The advent of relatively inexpensive but powerful PLDs as well as the relatively inexpensive IC fabrication technology[307] allows the offloading of control tasks from the system software to some form of external hardware.

In the end, one of the downsides of MCUs is that they are typically limited by the number of pins they can use to interface to the outside world. The pin count is generally related to the cost of the MCU also: the more pins you have on your MCU, the more you're going to pay for it[308]. Now days, MCUs can do many tasks (generally at the same time, sort of), which is good. The downside of having MCUs do many tasks is that the associated software becomes more complicated and error prone based on the number of tasks it is required to control. The type of errors associated with digital systems such as these are intermittent and hard to reproduce and thus repair.

So the good news is that FSMs are not quite dead; they are still used quite often to avoid some of the hassles created by complicating the software associated with the controlling circuits using MCUs. Although you probably don't know it, there are most likely quite a few FSMs embedded in the amazingly complex ICs that control everyday devices such as cell phones, MP3 players, bowling balls and other such useless devices that we can't seem to live without.

FSMs generally simplify required control tasks by off-loading the software-based control requirements to non-software-based circuitry, namely FSMs. In addition, FSMs can help reduce the I/O pin count requirements in MCU-based applications. In other words, FSMs are massively useful as well as massively interesting. What a deal! Use them where you can to make your world nicer. The following sections provide an intuitive overview of FSMs and include a few examples where somewhat real-world control problems can use FSMs.

The question that arises is: ***How do I use a FSM to control something?*** The answer to this question is based on whether you understand the following:

1. Understand how the FSM operates in terms of the underlying hardware (such as the storage elements, excitation logic, output decoding logic, next state decoding logic)

2. Understand the various lingo used when dealing with FSM (such as present state, next state, state transitions, external inputs, external outputs, state variables, next state decoder, output decoder, Mealy/Moore machine, strike, spare, etc.)

3. Understand the symbology used to describe the FSM (namely, the state diagram symbology and the PS/NS table content)

4. Understand how to implement the FSM (either flip-flops and discrete logic or some type of programmable logic device)

5. Understanding the many timing issues involvled in actual FSM implementations. FSM implementations have all the issues associated with real circuits plus some other issues associated with non-trivial sequential circuits.

This chapter describes the basic knowledge associated with designing FSMs that act as controller circuits. Without doubt, you'll find that FSMs are actually quite intuitive once you get comfortable with the items listed above. Most of the major issues dealing with the first three

[307] The thought here is that if you're going to fabricate an IC, including FSMs in on the IC requires a relatively small amount of real estate which makes them cost effective.
[308] Other things such as speed, memory size, and package affect the cost also.

points listed above are mapped out in the following sections; the fourth point is the least "engineering" related of the four points and amounts to what I refer to as grunt work. This grunt work includes discrete and VHDL behavioral FSM implementations.
The fifth point is something you'll need to deal with at some point[309]. The toughest part of any FSM design is to generate the state diagram; everything beyond that point is straightforward[310] to the point of being tedious grunt work.

29.3 The FSM: An Intuitive Over-Review

Figure 29.1 shows the general model of the FSM acting as a controller circuit. The things that are important to a controller circuit are the control signals (outputs that do the actual control) and the status signals (inputs to let you know what's going on). In the FSM model shown in Figure 29.1, the external inputs act as the status signals to the circuit your controlling while the external outputs act as the control signals that interface with hardware outside of the FSM. You generally need a clock signal to keep things synchronized[311]. The point we're trying to make here is that, in theory, FSMs are actually quite intuitive. The only real obstacle to designing FSMs is learning to represent your intuition with the standard FSM lingo and symbology.

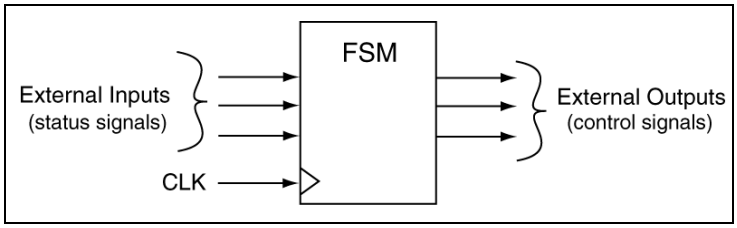


Figure 29.1: The general view of a FSM used as a controller circuit.

29.3.1 The State Bubble

The state bubble is one of the major features of a state diagram-based FSM description. FSMs use the state bubble to represent a particular "state" in an FSM. Figure 29.2(a) shows a typical state bubble. The following verbage lists some of the key features regarding the state bubble:

- A state needs some way to delineate it from other states, which is why the state bubble generally contains some type of identifying information. State bubble indentifying information includes either a symbolic name, or the actual state variable values used to encode the FSM. Most state diagrams use some form of symbolic representation except for sometimes FSMs describing counters. Using symbolic names delivers more information to the human reader if the state names are chosen such that they describe the significance of the state (it's called self-commenting; do it). For example, a state name such as "WAIT_FOR_SIGNAL" conveys a lot more information than "10". The symbolic state names should be unique for state diagrams to disambiguate the states.

[309] A later chapter presents more information regarding sequential circuits, namely set-up and hold times.
[310] Timing issues can be somewhat challenging, however.
[311] There are "clockless" FSMs out there; as a matter of fact, there is a big area of digital design that deals exclusively with asynchronous FSMs. It's interesting stuff; I hope some of you make it there someday.

- The current values of the underlying storage elements are what officially define the states themselves. In other words, each different state in a state diagram has a unique set of bits being stored in the storage elements. Although you could technically provide duplicate symbolic names for states, the implemented bit-level representations of the states must be unique.

- Timing diagrams represent the states by the time slots representing the possible state values. Figure 29.2(b) shows that the boundaries of these time slots delineated the associated active edges of the system clock. The synchronizing signal used by the storage elements bound the state timing slots. In the case of the diagram in Figure 29.2(b), the elements are rising-edge triggered (RET) storage elements because the rising clock edge defines the state boundaries.

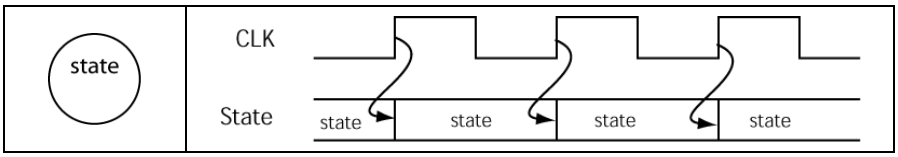


Figure 29.2: The State Bubble and associated timing diagram.

29.3.2 The State Diagram

The state diagram is one of many methods used to model a FSM. The particularly pleasing aspect of the state diagram is that their main purpose is to convey meaning and understanding to the human viewer (as opposed to facilitating the actual implementation of the FSM as a VHDL model generally does).

There are three forms of information presented by state diagrams: 1) the various states in the FSM, 2) the input conditions controlling the state-to-state transitions, and, 3) the output values associated with the various states. This section deals primarily with the state-to-state transitions associated with a state diagram. Figure 29.3(a) shows a typical (and overly generic) state diagram. The following verbage describes some of the key features of this state diagram.

- The terminology used to describe how a FSM goes from one state to another is referred to as a *state transition* or just *transition*. State diagrams use a "state transition arrow", or just "arrow", directed from the source state to the destination state to represent state transitions. Roughly speaking, there are only two possible state transitions in a state diagram from a given state. On the associated clock edge, a transition can occur from, 1) one state to another state (indicated by the "state change" label in Figure 29.3(a)), or, 2) the FSM can remain in the same state (indicated by the "no state change" label in Figure 29.3(a)). As we previously discussed, the "no state change" arrow is the now classic "self-loop". These are truly the only possible transitions[312] in a state diagram relative to a given state.

[312] We've dealt with asynchronous inputs in a previous chapter. The state transitions caused by asynchronous inputs (such as presets and clears) are simple to model and implement due to the fact they represent "special" circumstances in a FSM. What we're interested now is the typical operation of an FSM.

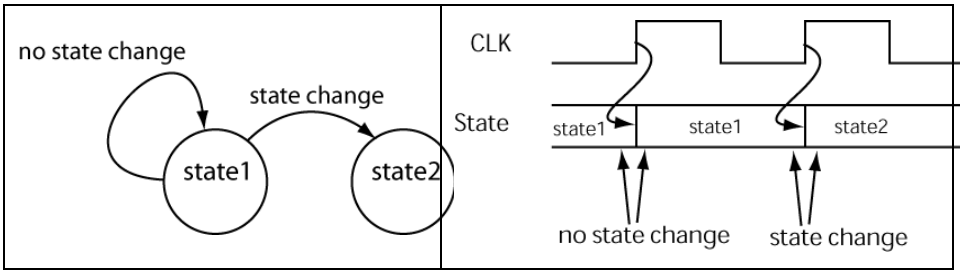
- The state diagram contains no clock signal. State diagrams never show system clock signal even though the system clock is an integral part of a FSM. The only part of the clock signal we're interested in is the active clock edge; the state transition arrows represent what action occurs on the active clock edge associated with a given FSM implementation. There is clocking information present in a state diagram, but it's only implied as opposed to specifically listed.

- The two states shown in Figure 29.3(a) have unique names. In real life, you would want to give these more meaningful names such as something to indicate why the state exists (or what is going on in that state).

- The state names provided Figure 29.3(a) give no indication as to how the states will be represented when the FSM is actually implemented. In other words, the state diagram provides no commitment to the actual state variable assignment used to disambiguate the states on a hardware level.

- The relation between the timing diagram shown in Figure 29.3(b) and the state diagram in Figure 29.3(a) is the key to understanding state diagrams in general. When we talk of state, we're talking about all the time in-between the active edges of the clock. In other words, the state bubble essentially represents all the time between any two active edges of the system clock. On the other hand, the state transition arrow represents what happens on each of the active clock edges. On each clock edge, one of two things must necessarily occur: the FSM transitions either to another state or the FSM remains in the same state. A more general way of saying this is that a state transition occurs on every active clock edge, but sometimes it transitions back to the same state. So once again, the state transition arrows in Figure 29.3(a) is thought of as the minute piece of time between two states in the associated timing diagram. The arrows help us model FSM behaviors but have no true significance in the associated hardware.

- The concept of Present State (PS) and Next State (NS) is somewhat hard to pin down in a timing diagram such as the one shown in Figure 29.3(b). The problem is that the present state (and hence the next state) is constantly changing as you travel from left to right on the time axis. If you declare one state as the present state, then you can declare the following state as the next state relative to the present state. This definition of course changes as you traverse the timing diagram. PS/NS tables do a better job of presenting present and next state information.



(a) (b)

Figure 29.3: A state diagram (a) and the associated timing diagram (b) with some interesting details.

29.3.3 Conditions Controlling State Transitions

As you would guess from examining the state diagram shown in Figure 29.3(a), there must be some mechanism that decides on which transition will occur from a given state on the next active clock edge. Note in Figure 29.3(a) that **state1** has two arrows leaving the state; this mean there are conditions associated with those arrows that decide on which transition actually occurs.

There are two forms of information that decide on what transitions a FSM will take from any given state: 1) at least one of the external inputs to the FSM, and, 2) the given state the FSM is currently in (otherwise known as the present state). The second condition is somewhat too general because when we're talking about state transitions, we talk about them from the context of being in one state and transitioning to another state. Each state has it own set of conditions that govern transitions, so in terms of state diagrams, we're more so concerned on a state-by-state basis as to what external input conditions control the state transitions from a given state. Figure 29.4 shows the way we indicate these conditions. From Figure 29.4 you can see that state diagrams list the conditions governing transitions by placing the conditions next to the state transition arrows.

In general, every state transition arrow must have conditions associated with it that describes what governs the transition[313]. On this note, there are three important things to keep in mind:

1. The conditions associated with the state transition arrows from a given state must be mutually exclusive. This means that there can never be the same set of input conditions associated with two different transitions arrows leaving the same state. If this condition did exist, the FSM would not be able to decide into which state is should transition.

2. The set of conditions associated with a particular state must be complete. If there is a set of conditions from a given state not covered by the associated state transition arrows, the FSM will once again not know what to do[314]. Your state diagram should leave no room for guessing. When you implement FSM using VHDL, you truly must cover all the cases or else the VHDL synthesizer or some other development tool will decide for you. Sometimes you may not like the tool's decision. Worst of all, you may have created an ugly, hard to trace error. You're boss will hate you and your modest raise will become even more modest.

3. If the state transition arrow has no conditions listed with it, this usually means the state transition is unconditional[315]. Once again, it is generally a better ideal to provide a "don't care" indication in the condition portion of the state diagram.

[313] This is sort of not always true. As we saw with counters, when the transitions were unconditional, we often listed no conditions. In reality, maybe we should have listed a "don't care" for those particular transitions.

[314] In cases such as these, the tools you're working with will generally not tell you about such conditions and will arbitrarily decide what it wants to do. In general, software design tools are generally make the assumption you know what you're doing and that you always do the right thing. With that assumption, the tools gladly fill in any details that you have have unintentially forgotten.

[315] Such transitions should be explicity noted in the state diagram; doing so will ensure the person reading the state diagram that you intented on having an unconditional transition.

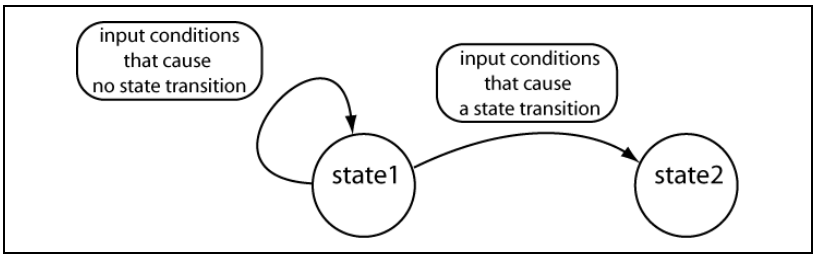


Figure 29.4: How state diagrams indicate the conditions associated with state transitions.

One thing to keep in mind here is that the FSM is a piece of hardware that controls another piece of hardware. The external inputs to the FSM are status inputs from the circuit that the FSM is controlling. The idea here is that, depending on the current status of the hardware that is being controlled, the FSM will transition to one state or another. The thing we haven't mentioned yet is that there are also some external outputs from the FSM which the FSM uses to control inputs to the circuit the FSM is controlling.

29.3.4 External Outputs from the FSM

The external outputs from a FSM are generally "control signals" used to control other circuits. In that the external input signals serve as status inputs to the FSM, the external output signals directly control other circuits. The state diagram will have different states and thus the control signals output from one state are generally not the same as control signals output from other states. This is because in general, the FSM is performing different control functions based on the different states in the FSM. If your FSM issues the same control signals from different states, there is a chance your FSM has redundant states (which is only a problem because redundant states waste hardware). Once again, the external inputs control the state transitions while the external outputs are issued based on the individual FSM states (and on external inputs in the case of Mealy-type outputs).

There are two different types of outputs in a FSM: Mealy-type outputs and Moore-type outputs. Although these two types of outputs are similar in most aspects, particularly in their controlling functions, they have one major difference. The outputs Moore-type outputs are a function of the state variables only while the Mealy-type outputs are a function of both the state variables and the current external inputs[316]. The common terminology used is to describe you FSM as either a Mealy-type FSM, or Mealy machine, or a Moore-type FSM, or Moore machine. There are FSMs that have both Mealy and Moore-type outputs; we generally consider these FSMs to be Mealy-machines since the overall machine contains external outputs that are a function of external inputs. This notion is somewhat intuitive in that the Moore-type output is really a subset of the Mealy-type output. For your viewing pleasure, Figure 29.5 and Figure 29.6 provide block diagrams of a Mealy and Moore-type FSMs, respectively.

[316] How many times do you think you'll hear about these characteristics?

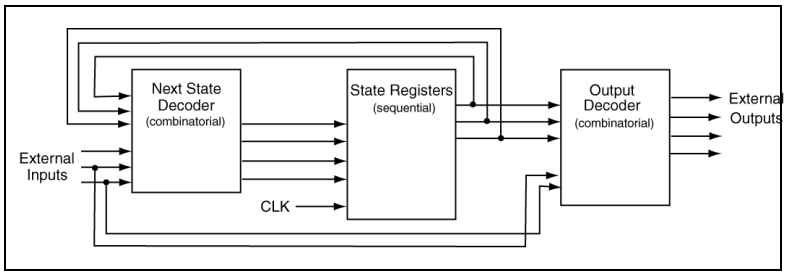


Figure 29.5: Block diagram of Mealy-type FSM.

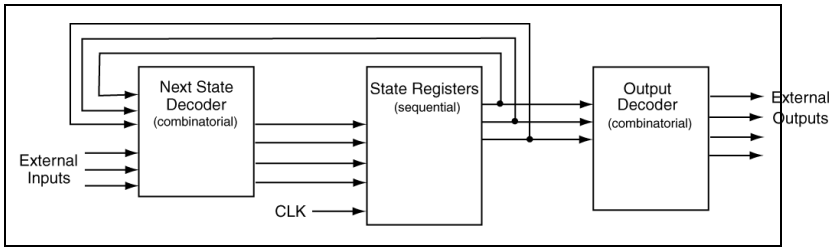


Figure 29.6: Block diagram of a Moore-type FSM.

What we're concerned about in this section is how we're going to represent the Mealy and Moore-type outputs on the state diagram. Although there are probably many ways to represent these outputs, the world's most intelligent people you the approach we describe here (so you should use it too). The key to understanding any state diagram is the legend that tells the viewer how to interpret what they're looking at. So if you deviate from the approach we'll describe, be sure to provide a detailed legend or appropriate annotations in order to appease the FSM Gods (as well as whoever is looking at your circuit).

Since Moore-type outputs are a function of the state variables only, we represent them by placing their values inside the state bubble. Figure 29.7 shows a state diagram that uses this approach. There can be any number of outputs represented inside the bubble. A comma usually delineates different outputs but you can use whatever method you choose[317]. Don't be afraid to increase the size of your bubble on your state diagram in order to include all the outputs. Clarity and readability takes home the prize when drawing state diagrams. Also, you should always use symbolic names rather than meaningless "1's" and "0's" in an effort to make you state diagrams more readable. The legend only goes so far when describing your state diagram; using symbolic names is simply a better approach[318].

[317] As always, make an effort to be as clear as possible.

[318] Yes, these examples often use 1's and 0's; but these are instructional problems. We we start doing something closer to real problems, we'll shift over to symbolic names.

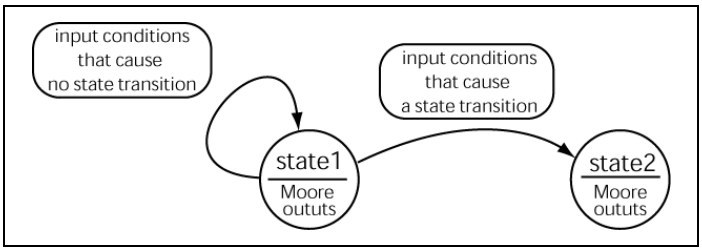


Figure 29.7: The State Bubble with associated Moore outputs.

The state bubble does not represent Mealy-type outputs inside of the state bubble because they are a function the external inputs as well as the state variables. To account for these characteristics in a state diagram, we list the Mealy-type outputs next to the external inputs associated with the individual state transition arrows and differentiated by the addition of the forward slash. Figure 29.8 shows an example of this approach. Once again, if a particular FSM has multiple Mealy-type outputs, these should be represented with either a comma separated list or something equally as readable.

There is a massively important feature shown in Figure 29.8 that can sometimes go without notice. Note that there are two sets of Mealy outputs shown in Figure 29.8 because there are two transitions from **state1**. The arrows are associated with the state transitions, which are based exclusively upon the current external inputs. But then again, the current Mealy-type outputs are also a function of those same inputs. Since the Mealy-type outputs are a function of the external inputs, they are represented by placing them next to the particular external inputs and associated with a given state transition arrow. Nevertheless, ***the listed condition of the Mealy-type outputs is always associated with the state the arrow is leaving*** (and not the state the arrow is entering). Figure 29.8 lists this notion if you by chance look close enough, so be sure to look close enough. Although this is not a complex point, understanding the symbology that state diagrams use to represent the state transition arrows and their association with the Mealy-type outputs takes getting used to. Representing the Mealy-type outputs in this manner may not be the clearest possible way, but if you can think of a better approach, knock yourself out.

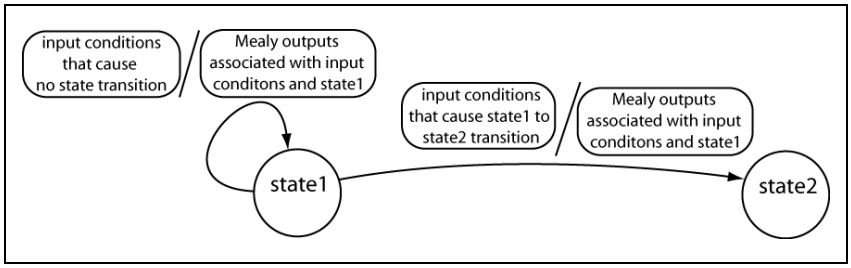


Figure 29.8: Representing Mealy-type outputs in a state diagram.

In addition, it should come as not surprise that you can represent both Mealy and Moore-type outputs in the same state diagram. Figure 29.9 shows an example of a state diagram that contains both Mealy and Moore-type outputs. Be sure to note the similarity between the state diagrams of Figure 29.8 and Figure 29.9.

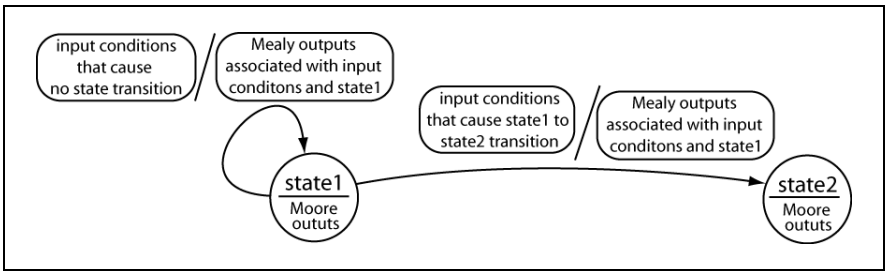


Figure 29.9: A state diagram that has both Mealy and Moore-type outputs.

As a final note in this section, the general rule with listing outputs is that you only list the "important" outputs for a given state. There are generally many outputs from a FSM, but not all of the outputs need to be assigned for every state. If in any state a given output is not assigned, it is assumed to be a "don't care". In terms of the circuit that the FSM is controlling, the outputs that werew omitted from a state will not have any effect on that circuit that the FSM is controlling. As you would imagine, the presence of *don't cares* simplifies the output decoding logic but has no negative side-effects on the FSM as was the case when not all external input conditions were accounted for in the state transitions from a given state. It is not bad practice to list all the external output for each state, but if you have many outputs, your state diagram becomes cluttered, thus causing the FSM Goddesses to shudder and frown upon your state diagram.

The example state diagrams we've work with so far seem to indicate the FSM states are somehow limited in the number or transition arrows that can leave (or enter) the state. There is actually no limit as is somewhat shown in the state diagram of Figure 29.10. The point we're attempting to make with the state diagram of Figure 29.10 is that there is no limit to the number of transition arrows leaving a given state. There are two key issues to be aware of regarding the transition arrows exiting a given state. First, make sure all of the conditions associated with each state transition are unique (no two arrows can have the same conditions). Secondly, make sure that the state diagram represents all possible conditions based on the inputs associated with the state transition arrows leaving the state (don't assume the FSM will magically stay in the same state if you don't explicitly specify all conditions).

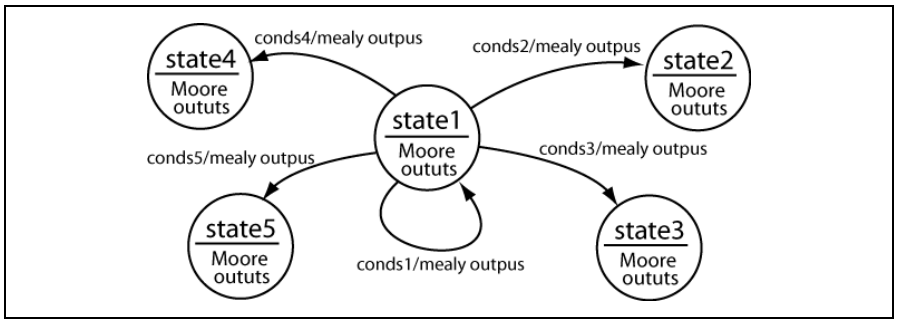


Figure 29.10: The State Bubble.

29.3.5 The Final State Diagram Summary

Figure 29.11 provides a quick overview of the relation between the FSM black box and the example state diagrams we've been working with in this section. What you should be gathering

from this diagram is that properly designed state diagrams have a particular structure that use a particular symbology. As Figure 29.11 is trying to show you, there is not that much to it once you understand a few simple points.

- Singly directed arrows represent state transitions.
- The FSM has external inputs that govern the state transitions.
- Each transition arrow lists the external inputs that control its transistion.
- Tthe state bubbles list the Moore outputs since they are only a function of state
- Mealy-type outputs are listed with the internal inputs (and hence the state transitions) since they are a function of both the present state and the external inputs.

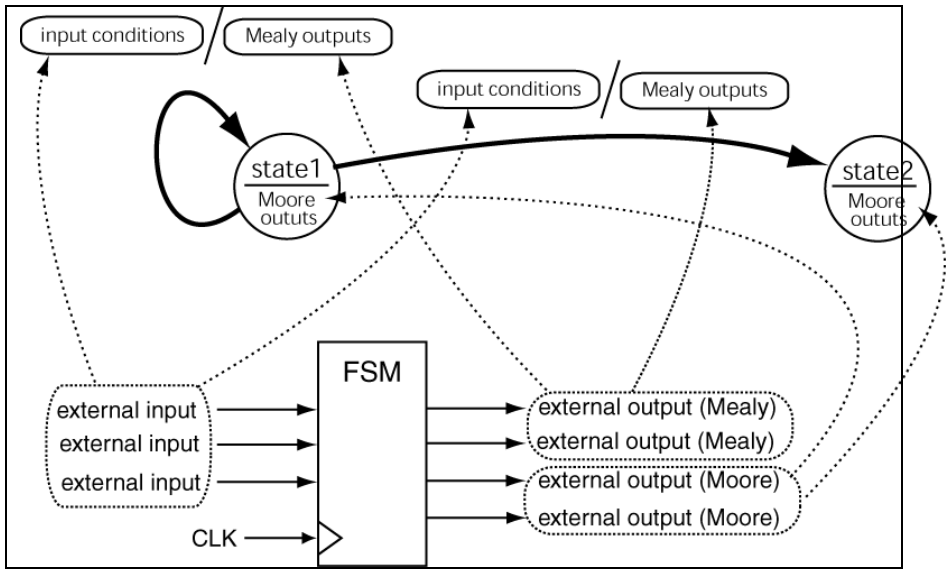


Figure 29.11: The relation between the state diagram and the high-level FSM.

Once final comment is in order here… There always seems to be question of how and where to start problems that require the generation of state diagram. There is no good answer for you; but there is a small suggestion: start somewhere and start right away. Drawing a black box diagram listing the FSM's inputs and outputs is always the best place to start because this allows you to get a feel for the system's status and control signals. From that point, try to look for a logical starting point and immediately draw a state bubble and some signals. The least complicated state (the state where the least is going on) is always a good initial state in your burgeoning state diagram. Having something or anything down on paper (even if it's crap) can often point to a viable solution[319]. Technically speaking, you can arrive at any viable solution regardless of the starting point. In other words, the final state diagram should be equivalent, or at least functionally equivalent, regardless of the initial starting point. Remember, the journey of a multi-state state diagram begins with the drawing of a single state[320].

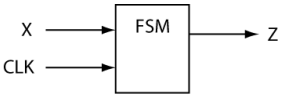
[319] Consider using an eraser or a piece of scratch paper.
[320] This has something to do with the Zen of digital logic design.

29.4 Sequence Detectors Using FSMs

The design of sequence detectors is usually one of the earliest topics discussed in FSM design. This is because the design of sequence detectors is highly instructive and spiritually enriching while not being overly complicated. In all of the problems we've discussed up to now, a state diagram was provided and it was up to you implement it using either classical FSM design or VHDL behavioral modeling techniques. You have many methods to choose from to implement FSM, but you're still developing skills for generating the original state diagram. The topic of sequence detectors is primarily a technique to generate the original state diagram; from there you can you any technique you actually choose to implement the FSM.

Many digital logic books include a "set of rules" to lead you through the process of designing FSMs that act as sequence detectors. The reality is that you'll spend more time trying to figure out the rules than you would spend generating the state diagram. Once again, this is a situation in which having an understanding of the processes involved ensures that you'll have the ability to generate the final state diagram. After you do a few of these problems, you'll most assuredly agree that they're not too compicated.

Figure 29.12(a) shows the general form of a simple sequence detector. In this problem, there is one external input **X** and one external output **Z**. Since this is a FSM, there is also a clock input. This particular example searches for the sequence "101" to appear on the **X** input[321]. Figure 29.12(b) shows a sample input sequence for the **X** input and the resulting outputs for two different types of outputs. In other words, the value of the **Z** output is in response to the sequence of **X** inputs. The data shown in Figure 29.12(b) is the data present when the active clock edge on the FSM appears (each column represents one clock edge).[322]



(a)

```
X:                0 0 0 0 1 1 0 1 0 0 1 0 1 0 1 0 0
Z (no reset):     0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0
Z (with reset):   0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0
Time →
```

(b)

Figure 29.12: A black box diagram (a), and sample inputs/outputs for finding "101" sequence (b).

Figure 29.12(b) lists two types of outputs: one where the FSM does not reset (non-resetting) after finding the correct sequence and the other where the Z "does reset" ("resetting") after finding the correct. In this context, resetting refers to the ability of the output to reuse past inputs regardless of whether they were part of a previoiusly successful[323] sequence or not. In the case where there is no reset, the Z output will be a '1' anytime the previous three X inputs[324] are the

[321] This statement is not entirely true and needs some qualification. We'll come back to this point and provide much need clarification later in this chapter.
[322] Once again, we will clarify this point later.
[323] Meaning that the sequence led to the finding of desired sequence.
[324] Once again, the most correct wording is that '1' was present on the X input when an active edge clock edge arrived.

sequence "101". For this case, the FSM can use previous X input values from other "101" sequences that were previously successfully detected. For the case where the Z does reset, no values from other sequences that were successfully detected can reused in the new sequence. These two conditions represent two possibilities in specifying this type of problem. Neither way is overly taxing and both provide equal satisfaction to the user. The previous sentence has no meaning whatsoever, but it sure sounds good.

One other consideration regarding sequence detectors is the fact that we can design the Z output as either a Mealy or a Moore-type output. Therefore, in the end, this type of problem can easily have one of four solutions based on the type of output (Mealy or Moore) and whether the machine resets or not after finding the correct sequence.

That's some of the overhead issues; now let's crank these things out. The following diagrams works through the example shown in Figure 29.12(b) thus producing a result in the four different methods ("Mealy" vs. "Moore" and "resetting" vs. "non-resetting"). We list less detail in some digrams due to the similarities in the development process. The next three figures have the solutions that represent all the possible conditions for the reset/no reset and Mealy/Moore options. Once again, as they say in real textbooks, generating the full solutions for the other forms of this example *are left up to the reader*[325]. This necessarily means you. If you understand the basics of these problems, you'll be infallible in the underground sequence detector network.

[325] Which is my way of saying I don't feel like drawing any more diagrams.

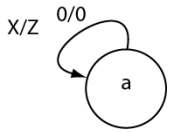
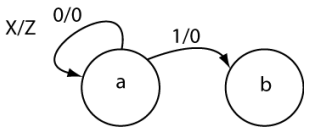
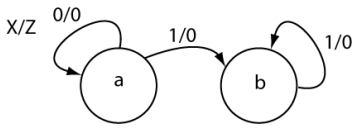
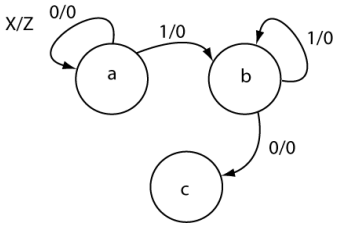
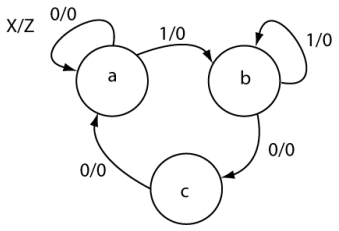
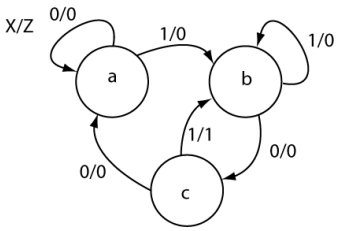
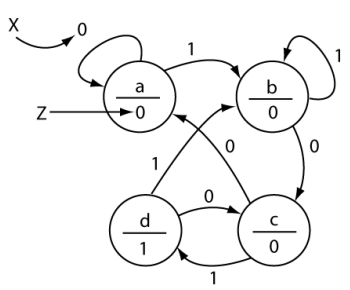
	You need to start somewhere… the best place to start is the beginning state where no correct values towards the finding the desired sequence have been found. The transition is the case where the undesired input of '0' is found and it stays in the state looking for the desired input of '1'. In this case, since the correct sequence has not been found, the Z output is a '0'. We assign the state bubble a symbolic value of an 'a'.
	Each state bubble must account for two arrows leaving the state (representing the two possible values of the X input). In this state, a '1' on the X input causes a transition to the new state. If you find yourself in state (b), then you know you've seen the first desired value of '1' on the X input. In essence, being in state (b) advertises the fact that the output of sequential circuits do indeed depend upon the sequence of inputs as opposed to just the raw inputs as they do with combinatorial circuits. It's a cool sort of memory characteristic.
	As long as the FSM receives a '1' on the X input in state (b), you must stay in the same state as indicated by the self-looping arrow. A '1' indicates the first value in the desired sequence. If it repeats, there is no reason to exit this state. No matter how many '1's you receive while in state (b), you'll never leave the state since the '1' still represents the first state in value in the desired sequence.
	Receiving a '0' in state (b) represents the second correct value in the sequence. In this case, you must transition to a new state. The output Z is still '0' because the complete correct sequence is yet to be found. Note that state (b) is complete now that there are two arrows exiting the state.
	Being in state (c) indicates you've found the first two values in the desired sequence. If at this point you were to receive a '0', you would essentially need to start the search sequence over which would result in a transition back to state (a). Remember, anytime you receive two contiguous '0's, you must start over again because two zeros are not part of the desired sequence.
	Finally, to finish off the diagram, if the FSM receives a '1' in state C, two things happen. First, you've found the desired "101" sequence and the output Z is set to '1'. Second, because the machine does not reset, you can reuse the one that made the "101" sequence a success as the first '1' in a new sequence; the transition to state (b) accomplishes this. You would not have been able to transition back to (b) if the machine was to reset after finding the correct sequence.

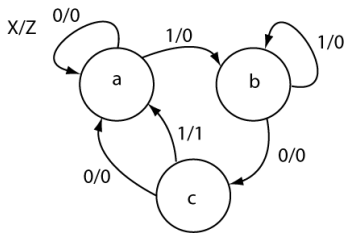
Figure 29.13: Generation of a state diagram that detects a "101" sequence without resetting.



The Moore state diagram from the example problem is similar to the Mealy state diagram. The main difference is the Mealy state diagram divides the outputs from the Mealy version of this problem into two states. Each of the two state bubbles includes a different output Z because they're Moore outputs. This solution shows one of the differences between Mealy and Moore-type FSMs: the Moore-type FSMs will have more states than a Mealy-type FSM implementing the same functionality. This is not good or bad, it just is. It's the Zen of FSMs.

(a) (b)

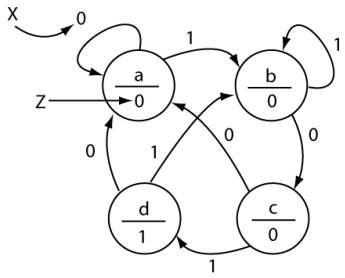
Figure 29.14: State diagram (a) and explanation (b) for Moore-output (no reset) for "101" sequence.



Only one state is different from this diagram and the non-reset diagram. All cases from state (c) return to the start case with one output being a '0' to indicate failure and one output being a '1' to indicate roaring success.

(a) (b)

Figure 29.15: State diagram (a) and explanation (b) for Mealy-output (with reset) for "101" sequence.



Again, this state diagram is similar to the diagram of the Figure 29.14. The two differences are associated with state (d). It is interesting to note the strange similarities between this state diagram and the state diagram of Figure 29.15.

(a) (b)

Figure 29.16: State diagram (a) and explanation (b) for Moore-output (with reset) for "101" sequence.

29.4.1 Sequence Detector Post-Mortem

Even though you should never simply "follow rules" when you're doing these problems, here are a few "suggestions" to chew on. As you do more of these designs, you'll develop you own style and collect your own set of tricks that make doing these problems easier.

1. Construct a sample input to clarify problem description.
2. Construct a path for the correct sequence first; then go back and fill missing transitions.
3. Try to add new arrows to existing states before adding new states.
4. Verify each state has one exit path for each value of the input variable(s).
5. Apply sample sequences to final state diagram to verify proper state diagram operation.

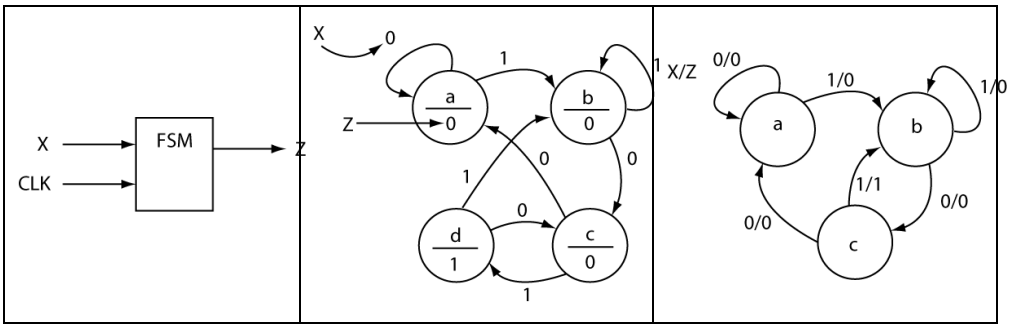
OK, items two and three are the opposite of each other; choose one or the other or somewhere in between. And the final thing to keep in mind here is that you can easily generate your own sequence detector practice problems. If you try out some strange sequences, you'll find that you run across some unique cases. Be sure to get creative and include some strange conditions such as decision points based other inputs along the way. You'll certainly be developing a strong foundation for future state diagram designs.

29.5 Timing Diagrams: The Mealy and Moore-type Output Story

The final step in developing a true understanding of FSMs is to understand the relationship between the state diagram and the timing diagram. This is generally not too hard but the sticking point always seems to be the how differences in the Mealy and Moore-type outputs show up on the timing diagram. In order to clarify these points, let's use the sequence detector FSM that we previously designed.

The FSM we were previously working with asserted the Z output when the sequence "101" appeared on the X input. We also added the constraint that the X input only changes no more than once per clock cycle[326]; Figure 29.17(a) provides a block diagram of this FSM. Figure 29.17(b) and Figure 29.17(c) show the state diagrams for the Moore-type and Mealy-type FSMs for this problem, respectively. The previous section derived these FSMs in a systematic manner and we'll spare the detail here. What interests us in this problem is how the Mealy and Moore-type outputs affect the timing diagram.

[326] We had to do this in order to simplify the problems. It's not too far of a stretch. We'll speak more of this before the end of this chapter.



(a) (b) (c)

Figure 29.17: The block diagram of the sequence detector FSM (a), the associated Moore-machine approach (b), and the associated Mealy-machine approach (c).

The difference between the Mealy and Moore-type state diagrams is readily evident in that the Mealy-type has one less state than the Moore-type. If you stare at these to state diagrams, you'll see that the main difference between these two diagrams is centered about the final two states in Moore-type state diagram and the final state in the Mealy-type state diagram. One approach to describing this difference is to say that the Mealy-type diagram divided state **c** into states **c** and **d** in the Moore-type state diagram. If you think about it, we had to do it this way because in the Moore-type state diagram, we required a state to indicate when the final bit to indicate the detection of the desired sequence.

For the case of the Moore-type FSM, the output Z is asserted for the duration of the state (state **d** Figure 29.17(b)) once the final bit in the sequence is detected. The corresponding state in the Mealy-type state diagram is state **c**. From this state, the Z output can be either a '1' or a '0' depending on the value of the X input. Because the output can be either a '1' or a '0' in state **c**, there is no need to break the single state **c** into two states (states **c** and **d**) as we did in the Moore-type state diagram. In other words, the output of state **c** in the Mealy-type state diagram can immediately indicate when the FSM detects the final bit of the sequence in the third state in the state diagram (state **c**). This is worth saying again; when the X input changes to a '1' in the **c** state, the correct sequence is "found" and the Z output indicates this by transitioning from '0' to '1'. Conversely, in the Moore-type state diagram, the output had to wait for the next clock edge to transition to state **d** which has an associated Moore-type output of '1'.

Figure 29.18 shows two example timing diagrams associated with the state diagrams of Figure 29.17(b) and Figure 29.17(c). For these two timing diagrams, assume that the FSM's active clock edge is the rising edge. By inspection, you can see that the top timing diagram must be the one associated with the Moore FSM because changes in the Z output are synchronized with state changes[327]. The arrows in the top timing diagram of Figure 29.18 show this synchronization. Once again, the external outputs of a Moore machine are a function of state only and therefore can only change when the state changes (which can only happen on active clock edges)[328].

In contrast, the lower timing diagram of Figure 29.18 shows that the output associated with the Mealy-type machine. Note that in the timing diagram for the Mealy-type machine, the output Z changes at times other than at the same time as the rising edge of the clock. More specifically, in state **c** of the low timing diagram of Figure 29.18, the Z input follows the change in the X input. Figure 29.17(c) show this characteristic by the two state transitions from state C in the Mealy-

[327] And the state changes are synchronized to the active clock edge.
[328] Repetition is a good good thing.

type state diagram. Note that transition associated with the X=0 input has an associated output of '0' while the transition associated the condition that X=1 has an associated output of '1'. In other words, the Z output has two possible values when the FSM is in state **c** as indicated by the state diagram. The actual output is a function of the X input as well as the state which is what makes it a Mealy-type FSM. This fine and important point is by far the most complicated issues when dealing with FSMs.

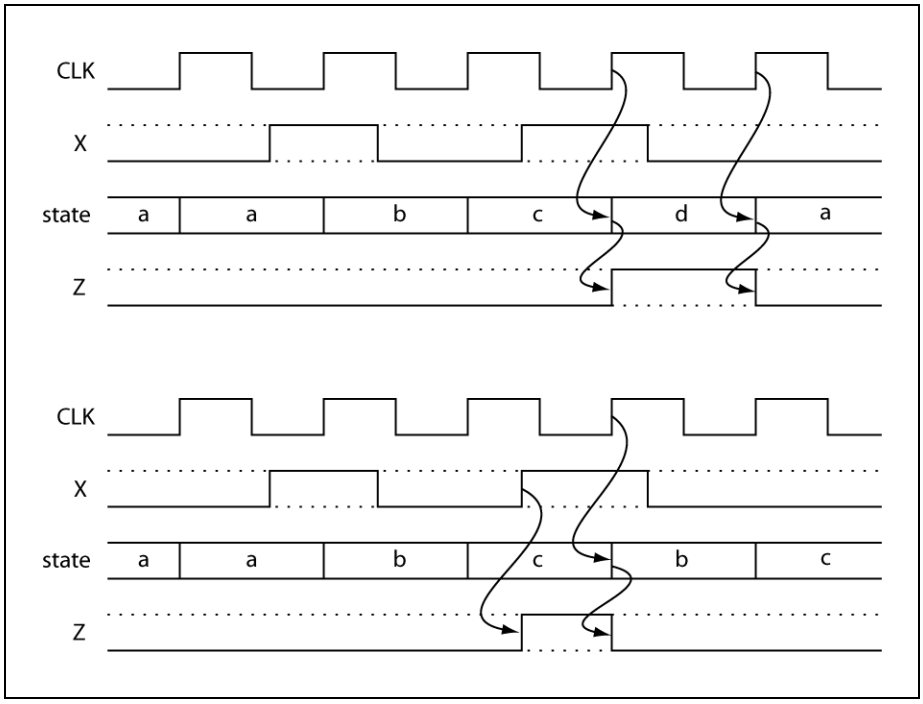


Figure 29.18: The timing diagram associated with the Moore-type machine (top) and the Mealy-type machine (bottom). Figure 29.17(b) shows the state diagram for the Moore-type FSM while Figure 29.17(c) shows the state diagram for the Mealy-type machine.

29.6 Sequence Detector: Mealy vs. Moore-type Clarification

Earlier in the chapter, we referred to an issue involed with sequence detectors as they relate to Mealy vs. Moore-type FSMs. The issue is that technically speaking, we must specify problems involving Mealy and Moore-type FSMs slightly differently in order for them to make 100% sense. The problem involves the notion that a given sequence becomes "valid" at different times based on whether the FSM is a Mealy or Moore-type machine.

For Moore-type FSMs, we only check the external input bit-stream on the active clock edge of the FSM. This fact makes the problem simpler because the FSM can officially ignore all changes in the sequence input that do not occur at the active clock edge. So for Moore-type FSM problems, we don't need to state extra clarification details in the given problem.

There are, however, some special considerations when dealing with a Mealy-type FSM. The notion in a Mealy-type FSM is that the bit-stream that is providing the sequence must be

monitored continuously because changes in the sequence input are always valid. The idea here is that any changes in the sequence input officially change the sequence. Therefore, if the sequence input toggles many times during a particular state, the sequence is officially providing a "0-1-0-1-0-1-0-1…" sequence. The associated FSM would not easily handle this characteristic of the sequence bit-stream. In order to simplify sequence detector problems, the approach we take is to state in the problem description that the sequence input changes (toggles) no more than once per state. Stating this constraint essentially simplifies the generation of the Mealy-type FSM.

Chapter Summary

- A FSM is generally used as a controller for some other hardware device. The external inputs to the FSM are status signals from the circuit being controlled while external outputs from the FSM are used as control signals to the device being controlled.

- State diagrams use state bubbles to represent the various states of the FSM. The state bubbles generally contain a symbolic name that represents the purpose of a given state.

- State diagrams use arrow notion to represent state transitions. Arrows can either be from one state to another state or from one state to itself (a self-loop indicating no state change, or a state change from a given state back into that state). State transitions generally occur on the active edge of the clock.

- External FSM inputs control state transitions in an FSM. From any given state, transitions are only a function of the external inputs. Transitions in the overall FSM are a function of both the external inputs to the FSM and the present state of the FSM.

- FSM can contain both Mealy and Moore-type external outputs. State diagrams represent Moore-type outputs inside the state bubble since they are only a function of the current state. State diagrams represent Mealy-type outputs as functions of the external inputs by placing them next to the state transitions arrows.

- All transitions from a given state must be mutually exclusive from all other transitions from that state. This means that there can be no combinations of external inputs that are represented in more than one transition arrow exiting a given state.

- The state transition arrows must represent all possible external input combinations exiting a given state. Not specifying every possible condition causes undefined FSM behavior.

- Sequence detector design is one of the most basic FSM design problems since they are instructive and can be relatively easy to do using state diagrams as a starting point.

Design Problems

1) Provide a state diagram that can be used to implement a FSM that indicates when the sequence "1011" appears on the FSM input (X). The FSM has two inputs (CLK,X) and one output (Z). This FSM does not reset when a '1' occurs on the output. The Z output is '1' only when the desired sequence is detected. Implement the state diagram two times: one time the output is a Mealy-type, the other time it is a Moore-type.

2) Repeat the previous problem but make the FSM reset when a '1' occurs on the output.

3) Provide a state diagram that can be used to implement a FSM that indicates when the sequence "01011" appears on the FSM input (X). The FSM has two inputs (CLK,X) and one output (Z). This FSM does not reset when a '1' occurs on the output. The Z output is '1' only when the desired sequence is detected. Implement the state diagram two times: one time the output is a Mealy-type, the other time it is a Moore-type.

4) Repeat the previous problem but make the FSM reset when a '1' occurs on the output.

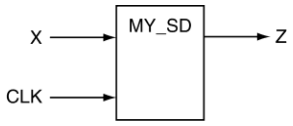
5) Provide a state diagram that can be used to implement a FSM that indicates when the number of '1's received at the FSM input (X) is divisible by 3. (0,3,6,9… are divisible by 3). This FSM has two inputs (CLK,X) and one Mealy-type output (Z). The Z output is '1' only when the desired sequence is detected.

6) Provide a state diagram that can be used to implement a FSM that indicates when the sequence "101" or "110" appears on the FSM input (X). This FSM has two inputs (CLK,X) and one output (Z). This FSM does not reset when one of the two given sequences appears. The Z output is '1' only when the desired sequence is detected. Implement the state diagram two times: one time the output is a Mealy-type, the other time it is a Moore-type.

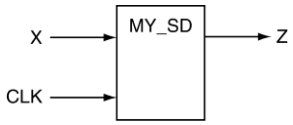
7) Provide a state diagram that can be used to implement a FSM that outputs the following sequence: "0100 110 110 110 …". This FSM has one input (CLK) and one Mealy-type output (Z).

8) Provide a state diagram that can be used to implement a FSM that indicates when at least two '1's and two '0's have appeared on the FSM input (X). Design the state diagram such that the order of occurrence of the inputs does not matter. The FSM has two inputs (CLK,X) and one Moore-type output (Z). The Z output is '1' only when the desired number of '1's and '0's has occurred.

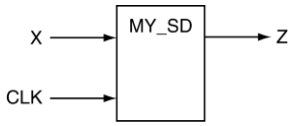
9) Provide a state diagram that describes a FSM that indicates when the sequence "1101" appears on the FSM input (X). The output (Z) is '1' only when this condition is detected. Implement this design as either a Mealy or Moore machine. Assume the **X** input is stable when each clock edge arrives and that **X** can change no more than once per clock period. Disregard all setup and hold-time issues.



10) Provide a state diagram that describes a FSM that indicates when the sequence "11001" appears on the FSM input (X). The output (Z) is '1' only when this condition is detected. Implement this design as ***both*** a Mealy and Moore machine. Design your state diagram so that the FSM reset once the correct sequence is detected. Assume the **X** input is stable when each clock edge arrives and that **X** can change no more than once per clock period. Disregard all setup and hold-time issues. Minimize the number of states in your design.



11) Provide a state diagram that describes a FSM that indicates when the sequence "10011" appears on the FSM input (X). The output (Z) is '1' only when this condition is detected. Implement this design as a ***Mealy*** machine. Design you state diagram so that the FSM ***resets*** once the correct sequence is detected. Assume the **X** input is stable when each clock edge arrives and that **X** can change no more than once per clock period. Disregard all setup and hold-time issues. Minimize the number of states in your design.

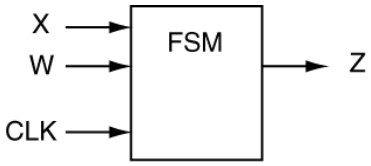


12) Provide a *state diagram* that describes a FSM that indicates when either one of the following two sequences are detected on the X input. Your design must use a **Moore-type** FSM that **resets** if either sequence is found. The Z output is asserted only when either sequence is found. *Minimize the number of states you use in your solution.*

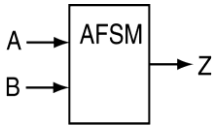
Assume:

- the **X** input is stable when each clock edge arrives
- the **W** input can change when only when the proper sequence is found
- full encoding with three flip-flops will be used to encode the FSM (limits state diagram to eight states!)

W	**Sequence searched for on X Input**
0	**1 0 1 1 0**
1	**1 0 1 1 1**



12) Design a fundamental mode, asynchronous finite state machine (AFSM) that detects the following sequence: AB = 11, 10, 11. The AFSM has two inputs, (A & B) and one output (Z) as indicated by the circuit diagram. The output Z equals 1 only when the proper sequence is detected. The AFSM ***does not reset*** when the correct sequence is found. Provide the equations describing the required state variables and output Z. Your design should be critical race and oscillation free. Don't bother removing static logic hazards.



30 Chapter Thirty

(Bryan Mealy 2012 ©)

30.1 Chapter Overview

Up until now, our work with FSMs has primarily been an academic exercise. Previous chapters presented the low-level FSM details, VHDL behavioral models of FSMs, and timing aspects of FSMs. The examples we completed in previous chapters were not overly close to real world applications.

Now that you know all the mechanics of FSMs and them implementations, you're ready to attack some actual real-world example problems. Solving "real" problems with FSMs will provide you with a great view of the awesome power of problem solving with FSMs. If you're still not sure about how to generate state diagrams, this chapter should tie together the missing pieces. Consequently, there is much less useless verbage in this chapter as we keep the main focus on basic problem solving techniques.

Main Chapter Topics

- **FSM Problem Solving**: This chapter finally introduces the true purpose of the FSM with their ability to solve real design problems based on their ability to act as controllers. The wait is finally over.

Why This Chapter is Important

This chapter is important because it describes techniques for solving actual design problems using FSMs.

30.2 FSM Overview

The Finite State Machine makes a great little controller circuit; Figure 29.1 shows the general model of the FSM acting as a controller circuit. The things that are important to a controller circuit are the control signals (to do the actual control) and the status signals (to let you know what's going on). In the FSM model shown in Figure 29.1, the external inputs act as the status signals to the circuit your controlling while the external outputs act as the control signals that interface with hardware outside of the FSM.

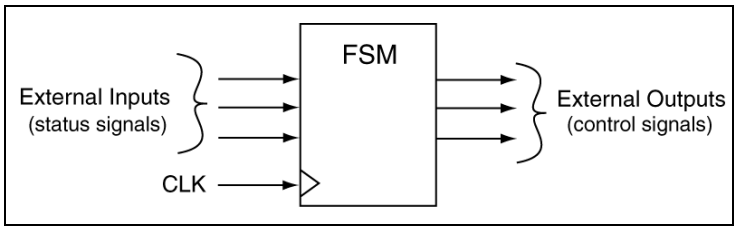


Figure 30.1: The general view of a FSM used as a controller circuit.

Your mission when faced with designing a FSM is to generate a state diagram that is able to solve the given problem. Once you complete the state diagram, the implementation of the FSM is not a big deal: the only true engineering required by the design is in generating the state diagram[329]. Truthfully, FSM design is an art form all its own; or maybe better stated, it's a language of its own. This language allows you to develop a state diagram that can implement a FSM that solves the problem at hand. What follows are some example problems of state diagram generation in painful detail. Take what you need. As you'll see in the following examples, developing an understanding of the problem's requirements is often times more timing consuming than the actual generation of the state diagram.

We all have a tendency to "look for the rules" on how to do new types of problems when they arise. I'm not sure is there is a set of rules on how to generate state diagrams to solve problems, but I sincerely doubt it. The issue is that problems such as these that we solve using FSMs come in many different forms; thus, there is no set of rules handles all cases. In an effort to toss something out at you, Figure 30.2 shows a wicked list of the only "rules" I can think of.

Finally, don't hesitate on any of the listed steps below. For that matter, don't allow yourself to get stuck on anywhere in any digital design. Always keep moving; if you go down the wrong path, toss out your design and start over with the items you learned from going down the wrong path. Don't hesitate to toss something down; if it makes no sense, tear it up and start over. The act of writing something down generally encourages your brain to traverse down various paths to solving the problem. Doing something is always better than doing nothing[330].

[329] A group of professors at Stanford was actually able to teach a troupe of wild apes to implement FSMs. These apes were subsequently given advanced degrees and now parade as academic administrators.
[330] Despite the examples being set by academic administrators.

Rule 0: Understand all aspects of FSMs. This includes Mealy vs. Moore machines, the associated timing diagram characteristics, the terminology associated with FSM, the various clocking aspects of FSMs, and the symbology associated with state diagrams as they relate to modeling FSMs.

Rule 1: Completely understand the problem at hand before doing anything. Most likely means that you'll have to read the problem many times as problems are typically not stated well, particularly problems in this text[331].

Rule 2: Draw a black-box diagram of the problem that clearly shows the inputs to (status signals) and output from (control signals) the FSM. Refer back to this diagram often.

Rule 3: Make a decision on whether you'll implement your FSM as a Mealy or Moore machine. This means you'll need to decide whether each external output is going to be implemented as a Mealy or Moore output.

Rule 4: After you put some thought into Rule 3, you should then draw some sort of legend that is consistent with the decisions you made in Rule 3. So get out a clean piece of paper[332], draw a big circle somewhere on it, and call it your legend.

Rule 5: Draw another big circuit somewhere on your paper and call that circle your starting state for the FSM. The best starting state is the state of the FSM with the least going on (such as the state where no outputs are asserted). From there, you'll need to start filling in the details. The biggest issue here is simply getting started.

Rule 6: Don't follow any of these rules.

Figure 30.2: A feeble attempt at "rules" for solving FSM-type control problems.

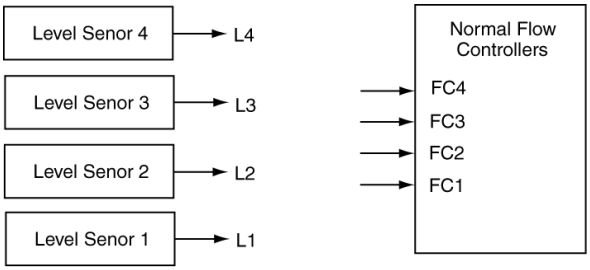
30.3 FSM Design Example Problems

Here are bunches of problems presented in varying amounts of detail. Have fun, and lots of it.

[331] Part of being an engineer is figuring out how to interpret poorly written problem descriptions. You'll receive a lot or practice with such problems in this text. Poor writing is part of the plan.
[332] Scratch paper (writing with one blank side and nothing important on the other side) works well here.

Example 30-1: Dam Waterflow Controller: Version I

A given dam has four water-level sensors with L1 being the lowest sensor placed vertically (placed at a greater depth in the dam). A given sensor is "on" (asserted high) when it senses water at that level and the sensor is low when no water is sensed at that level. The output of the four sensors drives four water flow controllers to which control the outflow of water from the dam. There is a one particular flow control associated with each level; for example, the actuation of Level Sensor 4 (L4) will activate Flow Control 4 (FC4). When the water reaches a particular Level Sensor, it activates the Flow Controller associated with that level. The flow controllers associated with the level sensors stay activated as long as the associated level sensor is activated. In other words, when the L4 sensor activates, all the other sensors are activated also. Provide a state diagram that describes a solution to this problem. Assume all inputs and outputs use positive logic.



Solution: A fine start on any solution is drawing the black box diagram associated with the problem. Figure 30.3 shows a diagram that clearly shows the inputs and outputs of the proposed FSM. This diagram also helps if you need to model the subsequent FSM in VHDL.

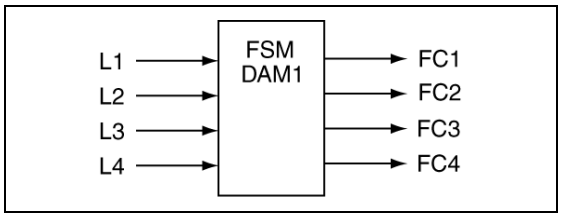


Figure 30.3: Black box diagram for solution of Example 30-1.

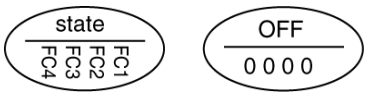
A fine place to start this state diagram is by drawing a legend and listing the seemingly "do nothing" state. The diagram on the right shows this starting point. The first state describes a condition where all the level sensors in the non-actuated state and the associated flow controllers turned off.	

Figure 30.4: Step 1) State Diagram design.

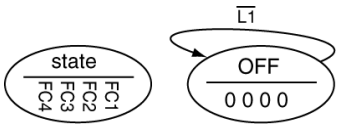
The FSM stays in the OFF state as long as the L1 sensor is not actuated; the complimented L1 in the state diagram represents this characteristic. This condition could also have been represented by nomenclature such as "L1 = 0". The approach we chose is technically uses PLC[333] (positive logic convention) which often leads to more readable state diagrams. Note that the act of not listed the L2, L3, and L4 inputs indicate that these inputs are officially "don't cares". "Not caring" about these inputs makes sense in the context of this problem because if the L1 sensor is not actuated, none of the other sensors will be actuated since all sensors are based on the level of water behind the dam.	

Figure 30.5: Step 2) State Diagram design.

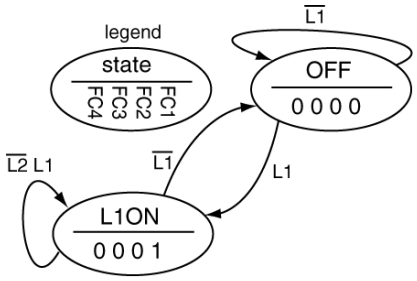
The first meaningful state is when the first sensor actuates. In this case, the FSM transitions to the L1ON state. The FSM will stay in this state as long as the L2 sensor is not actuated and the L1 sensor remains activated[334] (as indicated by the self-loop). If the L1 sensor were to de-activate, the state transitions back to the OFF state, as indicated by the complemented L1. Note that the FC1 flow controller is actuated when in the L1ON state as indicated with the '1' in the FC1 position; the other outputs remain off. Once again, not listing all the sensor inputs in the new state indicate that they are in "don't cares"	

Figure 30.6: Step 3) State Diagram design.

[333] The use of PLC in this case represents a shorthand notation of sorts. We know that L1 is positive logic so the complement of L1 is therefore the non-active state for that signal.

[334] Note that this notation does not use an explicit AND operator. This is typical of state diagrams: the AND operator is understood to be there.

From the L1ON state, a transition is made to the L2ON state when the L2 sensor is actuated. The FSM remains in the L2ON state so long as the L2 sensor remains activated and the L3 sensor is not actuated. If the L2 sensor turns off, the state transitions back to the L1ON state. Note that the FC2 flow controller is actuated in the L2ON state in addition to the FC1 flow controller. One other thing of interest here is that the conditions associated with the self-loop in the L2ON state do not include the L1 variable. This is officially a "don't care" with the reasoning that if the L2 sensor is one, then the L1 sensor must be on. You could list the L1 input in the condition but it would clutter your state diagram.

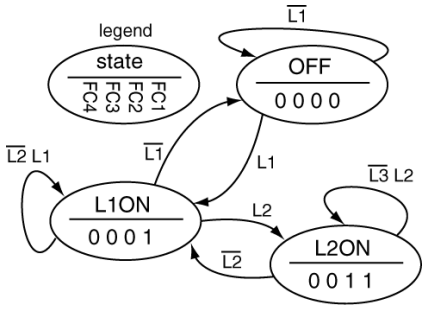


Figure 30.7: Step 4) State Diagram design.

This state is similar to the description of the previous state so we'll save the verbage here. You'll typically be able to exploit symmetry in the state diagram in problems such as these; as such, it should be something you stay on the lookout for. You can also exploit this symmetry to ensure you written your equations correctly. Be sure to exploit this symmetry.

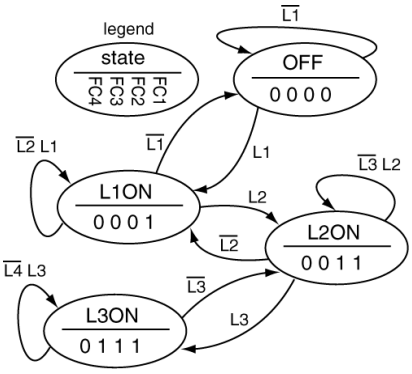


Figure 30.8: Step 5) State Diagram design.

Finally, let's consider the L4ON state. There is not much new here. Actuation of the L4 sensor sends the FSM in to the L4ON state. The FSM remains in that state so long as the L4 sensor remains actuated. The FSM transitions back to the L3ON state from the L4ON state when the L4 sensor deactivates.

Was this solution trivial? Yeah, probably. You're probably thinking that you should simply attach the output of the water level sensors other flow controller input. Great idea! Consider this problem a confidence builder. We'll make it a bit more challenging in after the next example.

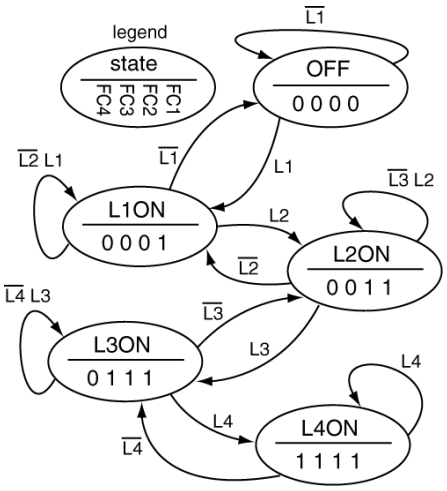


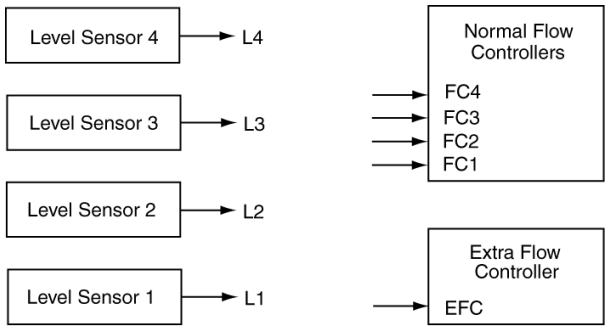
Figure 30.9: Step 6) State Diagram design.

We need to revisit this Dam Waterflow control problem. Unfortunately, the last dam failed miserably in a giant storm (the FSM design was fine but the dam design was bad, particularly the flow controllers). The town below was flooded, which generated a massive number of lawsuits.

For the upcoming problem, the dam contains an extra flow controller. This flow controller requires somewhat specialized control features outlined in the problem description that follows. Once again, this design acts alone and there is no pressing need to interface the FSM with a MCU. Once again, this is truly a case where understanding the problem requires more time than doing the actual design.

Example 30-2: Dam Waterflow Controller: Version II

A given dam has four water-level sensors with L1 being the lowest sensor placed vertically (placed at a greater depth in the dam). A given sensor is "on" (asserted high) when it senses water at that level and the sensor is low when no water is sensed at that level. The output of the four sensors drives five water flow controllers to which control the outflow of water from the dam. There is a one particular flow control associated with each level. The fifth controller actuates whenever the transition of a state goes from a lower water level to a higher water level; it is not activated when the water level transitions from a higher level to a lower level. The flow controllers associated with the level sensors stay activated as long as the associated level sensor is activated. In other words, when the L4 sensor activates, all the other sensors are activated also. Assume all inputs and outputs are positive logic.



Solution: As always, a good place to start with this solution is to draw a block diagram of the FSM. This gives you meager clues and points you to the road of assured success. Figure 30.10 shows the associated black box diagram. The solution with explanation follows of the developing state diagram happily follows.

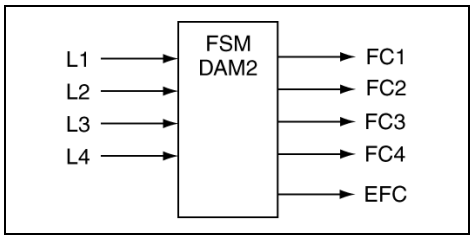


Figure 30.10: Black box diagram for the solution of Example 30-2.

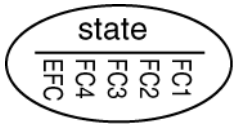
A worthy place to start is with the legend. This is always good place to start especially if you can't figure anything else to do. Besides, it's the next logical step from generating the FSM black box diagram of Figure 30.10 (in case you're running short of ideas).	

Figure 30.11: Step 1) State Diagram design.

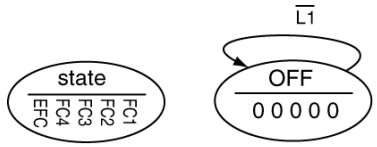
The first real state is generally the "waiting for something meaningful to happen" state. For this example, we'll call if the OFF state. In this step, the FSM stays in this state so long as the L1 sensor remains off as indicated in the state diagram with the complemented L1 on the self-loop. Note that all of the flow controllers remain off in this state (thus the off name). Very clever.	

Figure 30.12: Step 2) State Diagram design.

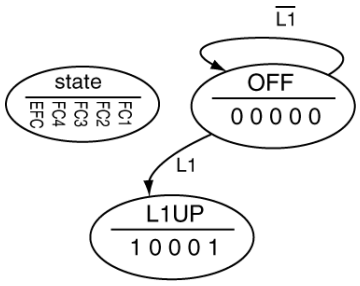
From the OFF state, the state transitions to the L1UP state if the L1 sensor is actuated as indicated by an uncomplimented L1 signal associated with the transition to the L1UP state. In the L1UP state, both the EFC and FC1 flow controllers actuate as indicated by the 1's in the respected positions in the state bubbles. Note that the water level is rising in this state. In other words, to arrive in the L1UP state, the water level must have risen.	

Figure 30.13: Step 3) State Diagram design.

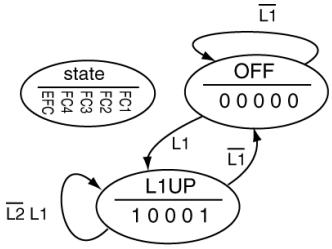
The state diagram on the right shows two new transitions. The FSM stays in state L1UP under the condition that the L2 sensor is not actuated but the L1 sensor is activated. Likewise, the state diagram leaves the L1UP state under the condition that the L1 sensor deactivates. In that case, the FSM transitions back to the OFF state.	

Figure 30.14: Step 4) State Diagram design.

The next logical state is the L2UP state. Transition to this state occurs when the L2 sensor activates. The corresponding output asserts the FC2 output, which actuates the FC2 flow controller. Because the water level has risen from the L1UP state to the L2UP state, the EFC output is remains asserted.

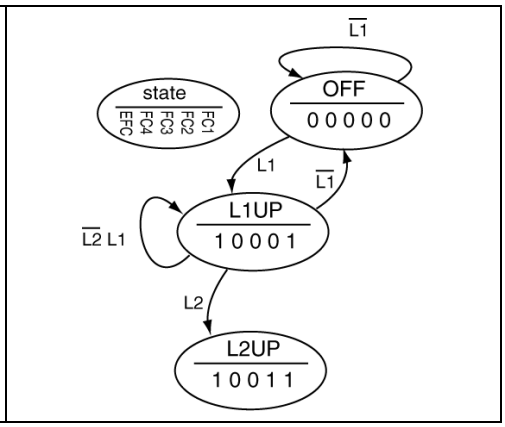


Figure 30.15: Step 5) State Diagram design.

The FSM stats in state L2UP so long as the L2 sensor remains activated and the L3 sensor is not activated. If the L2 sensor de-actuates in the L2UP state, the state transitions to the L1DN state. This state is similar to L1UP state except for the fact that that the EFC output is not actuated. In other words, if the water level is falling as is indicated for this transition, the EFC output is not asserted. The L1UP and L1DN states are essentially the same state aside from the fact that the EFC output is different in these states.

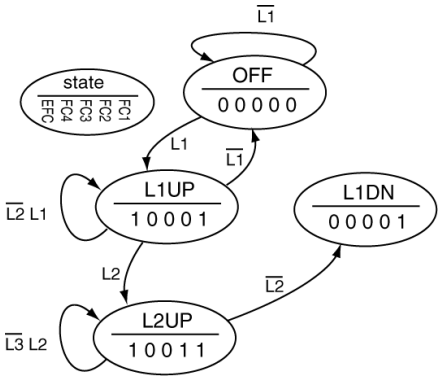


Figure 30.16: Step 6) State Diagram design.

The FSM stays in the L1DN state so long as L2 is not actuated and L1 remains actuated. Looking ahead to the diagram on the right, if L2 is re-actuates from the L1DN state, the state transitions back to the L2UP state and the EFC is turned on. Also from the L1DN state, if the L1 sensor deactivates, the state transitions back to the OFF state (not shown).

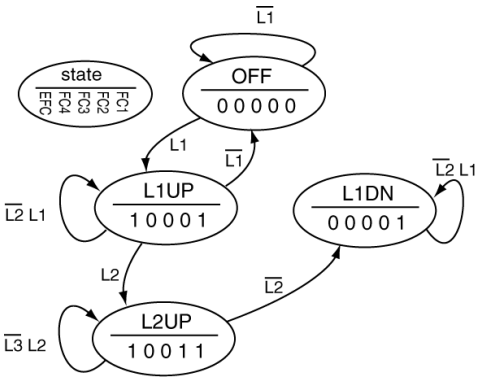


Figure 30.17: Step 7) State Diagram design.

From the L2UP state, if the L3 sensor actuates, the state transitions to the L3UP state. For the next few transitions, there is a certain symmetry is taking place in this state diagram. If you understand all the transitions up until now, the next few additions to the state diagram are nothing new. This being the case, we've ejected some of the boring explanatory details.

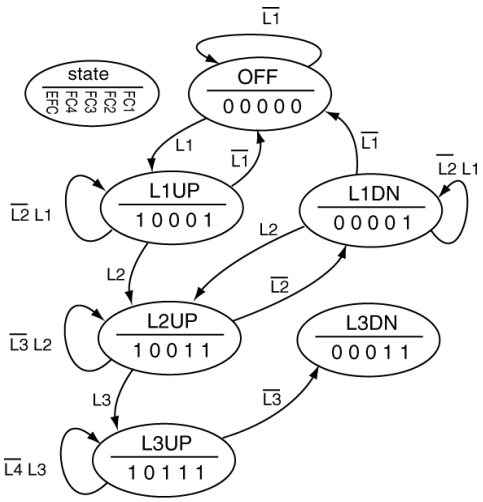


Figure 30.18: Step 8) State Diagram design.

Here's the addition of the final state which results from a transition from state L3UP under the conditions of the L4 sensor being activated. The addition of the L4UP state is similar to the L3UP and L2UP states. The "down" states associated with the "up" states are also similar so we do not include them here.

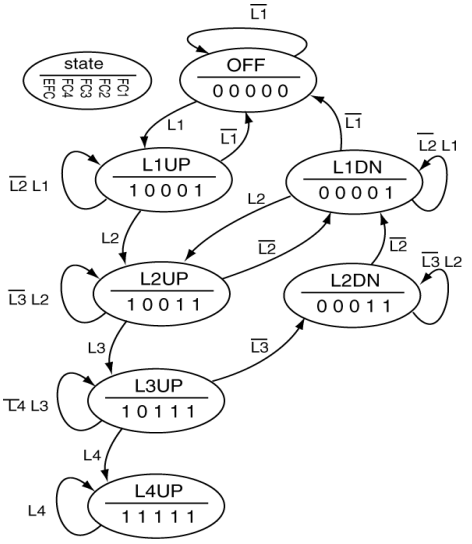


Figure 30.19: Step 9) State Diagram design.

Finally, bunches of more additions made to the state diagram but they are essentially nothing you have not already dealt with. The state diagram is complete. And it's beautiful. Note how the symmetry of the state diagram tugs on your soul strings.

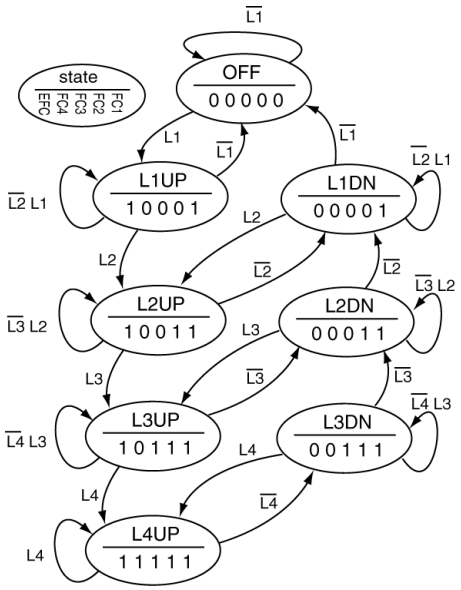


Figure 30.20: Step 10) State Diagram design.

Wow. The state diagram looks amazing. You can easily impress your friends[335] and acquaintances with this puppy, but your friends will know that it's not as amazing as it looks. Best yet, the dam works happily ever after; the town below is saved. You're a hero.

The next example problem is somewhat advanced in terms of the devices and ideas that it uses. Despite this notion, it is still understandable and you'll be better off if you put the time into understanding it. Many of the important underlying details of this problem will become clearer later in this text. The following verbage attempts to describe some of these details without too much splendor.

Since computers typically are required to interface with an analog world, there exists some type of analog-to-digital conversion (ADC) in every digital system. ADCs, generally speaking, generate a digital value (a set of bits) that corresponds to an analog input. The generated digital value is proportional to the analog value. The range of analog values that can be converted is a function of the ADC you're using as are the digital characteristics of the digital values. The bit-width of the converted value indicates the number range of the converted value. The ADCs themselves are typically rated by the bit-width, or *resolution*, of the converted value.

The resolution of ADCs typically runs from 6-bits to 24-bits (depending on how much you're willing to pay). ADC's typically are implemented using different approaches and algorithms but the following problem omits these details. Any one device will have only one resolution associated with it. There are lots more details involved in ADCs but they are not relevant to this discussion.

Generally speaking, a set of control signals are used to control ADCs. Any outside device such as a microcontroller or a FSM can tweak these control signals. Each ADC will have different

[335] And bowling buddies.

control requirements but they generally are somewhat similar. Once again, there are a lot more details on this but we'll pass by most of them and examine the simplified specification shown in Figure 30.21.

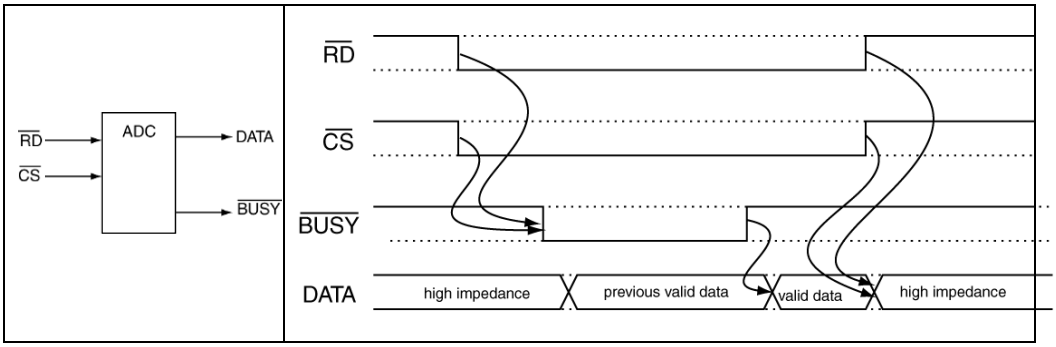
Figure 30.21(a) shows a black box diagram of the ADC we'll work with and Figure 30.21(b) shows the operating characteristics. To obtain an analog to digital conversion from this object, you must follow the following sequence of control signals. Your mission for the upcoming problem is to design a state diagram that will synthesize[336] the control signals in such a way as to make the ADC operate properly. Here are the important things to note about the timing diagram in Figure 30.21(b).

- **Drop the RD and CS signals low.** This notifies the ADC that you want to initiate a conversion. Before the conversion initiates, the ADC is considered to be in an *idle state*.

- **Wait for BUSY to drop low.** The ADC drops this signal low after a certain amount of time designated in the spec for the ADC device. Once this signal drops low, the ADC has started a conversion. Clock signals drive ADCs (either internal or external) which effectively times the conversion. The point in this example is that you don't know or care how long the clocking is going to take[337]. The important thing is that once the conversion is complete, the ADC drives the BUSY signal high.

- **Wait for the BUSY signal to return high.** This indicates that the data on the output lines is valid digital data from the more recently initiated conversion. The data is then available to be read and used by some other device in your system.

- **Return the RD and CS lines to the high state.** This has the effect of returning the device to an idle state. The ADC's idle state produces high-impedance[338] on the device's data output. Once the conversion is complete and the data associated from the conversion is read, the data outputs are driven back to high-impedance by bringing the RD and CS signals back to a logic high state.



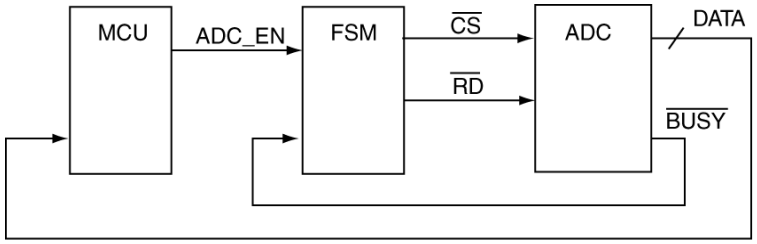
(a) (b)

Figure 30.21: Simplified black box diagram and typical timing specification for an ADC.

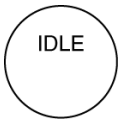
[336] The word "synthesize" is used often in this context. It simply means that you're going to need to design some hardware (the FSM) that produces the control signals as shown in the ADC's timing diagram.
[337] This clocking has nothing to do with the system clock driving the FSM.
[338] This is electronics stuff; it's really complicated (but you don't need to deal with it now). For now, you can consider this an on/off function with the high-impedance condition being equated with the off state.

Example 30-3: Pin-Limited MCU/ADC Interface

Design a FSM that initiates an analog-to-digital conversion using only one signal (ADC_EN). The controlling signal is the ADC_EN signal which connects to a microcontroller for this example. Transitioning the ADC_EN signal from low-to-high initiates the analog-to-digital conversion process. The FSM initiates the ADC process by dropping CS and RD signals, waiting for the BUSY signal to drop, and then waiting for the BUSY signal to rise indicating the ADC has completed the conversion. The high-to-low transition of the ACD_EN signal places the ADC back into its idle state. It is the job of the MCU to wait the proper amount of time for the conversion to complete before the data is read. A simplified black-box diagram of the circuit is shown below (signal such as the analog input are missing). You only need to generate the state diagram for this FSM.

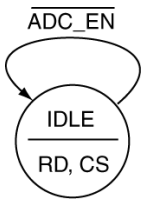


Solution: The following figures show the solution to Example 30-3 using a modest yet painful amount of detail.



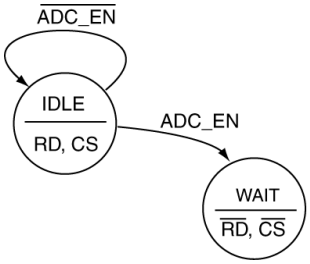
A good place to start in any FSM design is the state in which there is the least going on. For this design, we'll use symbolic mnemonics to represent the state. We'll call the beginning state of this diagram the idle state. This state signifies that no conversion is currently taking place.

Figure 30.22: Step 1) State Diagram design.



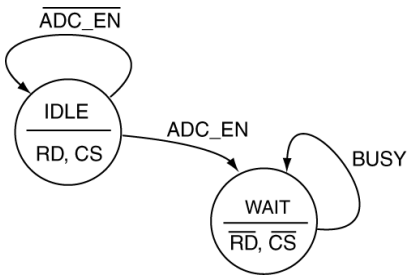
Moore-type FSM models can most easily represent the two control outputs (RD and CS). In the idle state, both of these outputs are high so the state lists them accordingly. The FSM will stay in this state so long as the ADC_EN signal remains low as indicated by the complemented self-loop in this state.

Figure 30.23: Step 2) State Diagram design.



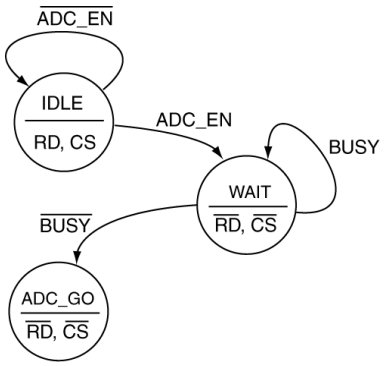
If the ADC_EN signal transitions from low-to-high, the MCU requested the initiation of an analog-to-digital conversion. Once this occurs, the FSM must actuate the RD and CS signals (these signals are active low and were "high" in the idle state). We must add a new state in order to handle these conditions; transition to this new state occurs when the ADC_EN signal is in the high state.

Figure 30.24: Step 3) State Diagram design.



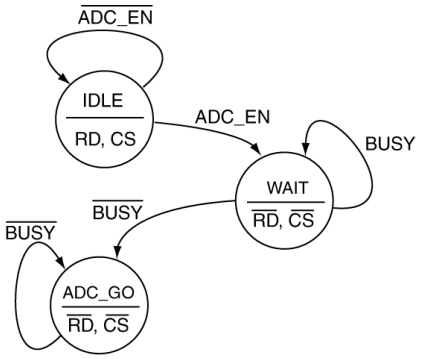
Once the analog-to-digital conversion initiates in the previous state, the FSM must wait for the BUSY signal to transition from high-to-low. The BUSY signal is controlled by the ADC and indicates the current state of the ADC (whether the ADC is currently doing a conversion or not). The FSM requires this state because the FSM needs to monitor this signal to know when the requested conversion is complete. The next step shows this condition.

Figure 30.25: Step 4) State Diagram design.



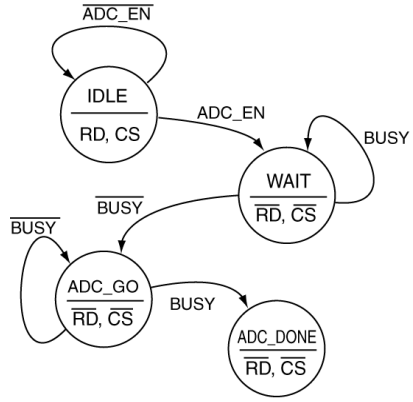
Once the BUSY signal drops low, the FSM transitions to a new state (symbolically labeled ADC_GO). In this new state, the analog-to-digital conversion has started; the results are forthcoming.

Figure 30.26: Step 5) State Diagram design.



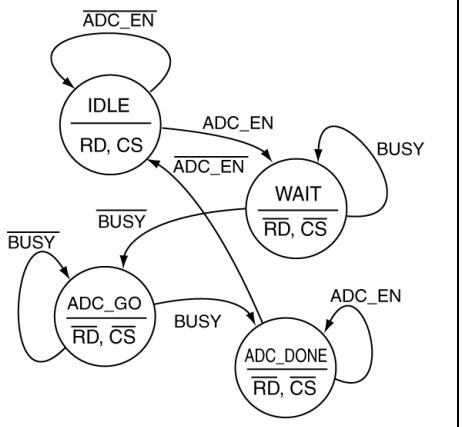
The FSM stays in the ADC_GO state as long as the BUSY signal remains low. Once the BUSY signal is driven high (by the ADC), the conversion is complete. One of the nice things about this design is this. Both the FSM and the ADC depend on system signals but they don't necessarily need to use the same clock signal. The purpose of the BUSY signal is to indicate to the world outside of the ADC device that the analog-to-digital conversion is complete. In this way, the two clocking signals do not need to match or synchronize in any way. This is a typical digital design approach as the synchronization of clocking signals is something everyone tries to avoid.

Figure 30.27: Step 6) State Diagram design.



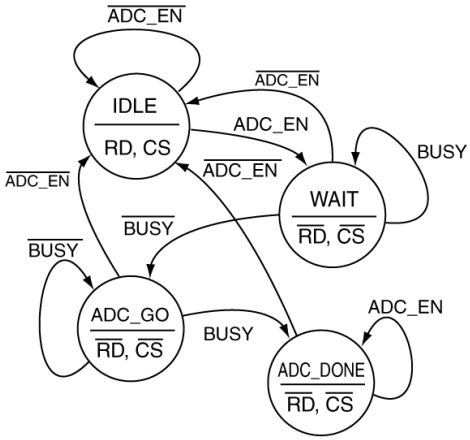
The BUSY signal is driven high once the analog-to-digital conversion has completed. This requires that we transition to a new state that we'll call ADC_DONE. The reason we need to add this new state will become apparent in the next box.

Figure 30.28: Step 7) State Diagram design.



Finally, in the ADC_DONE state, we must wait for the MCU to read the conversion data. The FSM keeps the ADC in this state as long as the ACD_EN signal remains high. The underlying thought here is that the MCU will eventually read the ADC data and then de-assert the ADC_EN signal. Once the ADC enable signal is de-asserted (driven to the high state), the FSM returns to the IDLE state.

Figure 30.29: Step 8) State Diagram design.



While the previous diagram initially appears complete, we could actually make it better. Our goal here is to make the FSM more robust. In order to do this, we technically need to cover the unexpected input cases. The question left unanswered in the previous state diagram is: what happens when an unexpected input sequence occurs? The design approach we took for this FSM was to generate the state diagram based on the correct input sequence occurring; but what would happen if the ADC_EN input were disabled soon after it was enabled? The answer to this depends primarily on the ADC hardware. Nevertheless, since we don't know this level of detail from the program description, we instead add some extra transitions to our state diagram. The final state diagram now has conditions that return the FSM to the IDLE state if the ADC_EN input is unasserted from any state.

Figure 30.30: Step 9) State Diagram design.

Example 30-4: Plant Watering Controller (Version 1)

Generate a state diagram that could be used by a finite state machine (FSM) to control can be used to control a plant watering device. This device has two outputs: one output, DRY, indicates when the planting medium is in need of water. The other output, FLOOD, indicates when the water level has reached a given point. The watering device has one input, WATER, that controls whether water is pumped into the planter or not. The operation of this device is such that if the planter is dry, the pump actuates until a flooded condition is present.

Solution: The key to understanding this problem is to draw an appropriate circuit diagram that shows your FSM as well as all the information (inputs and outputs) that you know about the watering device. Figure 30.31 provides the well-known black box diagram. For this particular problem, this diagram is massively important due to the wording used in the problem. What is not immediately apparent from the problem description is input/output description. You need to read the problem carefully to ascertain that the output from the watering device is a status input to the FSM while an input to the watering device is an output from the FSM. The black box diagram helps clarify the input/output specification.

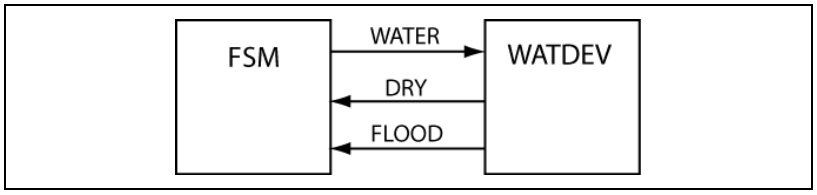


Figure 30.31: The block diagram of the final circuit.

The next thing you need to do is generate the required state diagram. For this diagram, there are two states: the OFF state and the ON state. A Moore-type output represents the WATER output, which controls the watering device. Figure 30.32 shows the final state diagram for this problem.

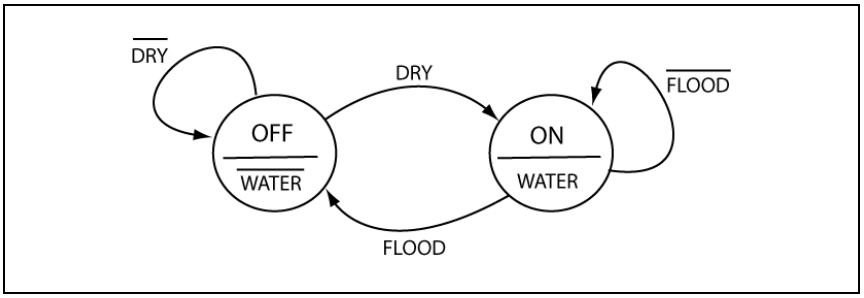


Figure 30.32: The state diagram describing the required FSM for Example 30-4.

Example 30-5: Plant Watering Controller (Version 2)

Design a state diagram that could be used by a finite state machine (FSM) to control can be used to control a plant watering device. This device has two outputs: one output, DRY, indicates when the planting medium is in need of water. The other output, FLOOD, indicates when the water level has reached a given point. The watering device has two inputs, WATER, that is used to control whether water is being pumped to the planter or not, and FERT, that is used to actuate a pump provides liquid fertilizer to the plant. The operation of this device is such that if the planter is dry, the pump actuates until a flooded condition is present. The plant need fertilizing every other time the plant is watered.

Solution: The key to this problem is noticing that it is relatively similar to the previous problem. As you probably noticed in the previous problem, you could have designed the required circuit without using a FSM (namely a combinatorial design). This problem is different because you must fertilize the plant at every other watering. Although you probably could dream up some combinatorial circuit that would be adequate for this design, a FSM is the best approach.

As is usually the case, a good place to start with this problem is generating a black box diagram of the solution circuitry. This diagram explicitly shows all the device and FSM inputs and outputs, and thusly provides you with an enlightened approach to the problem. Figure 30.33 shows the resulting black box diagram.

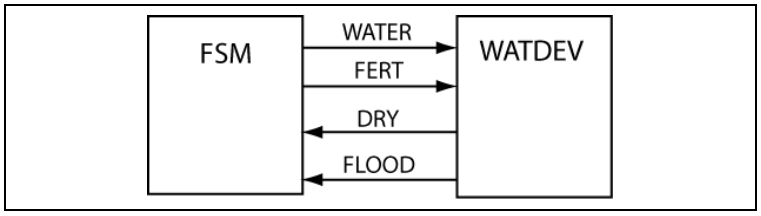


Figure 30.33: The block diagram of the final circuit.

The state diagram for this is similar to the state diagram in Example 30-4. In essence, we've repeated the state diagram twice and used one of the repetitions to turn on the FERT input. The state diagram indicates the FSM's is ability to "remember" whether it added fertilizer during the previous dry spell or not. This is pretty cool; the plants in your grow room will love you and provide you with endless hours of escape from academic drudgery. Figure 30.34 shows the final state diagram for Example 30-5.

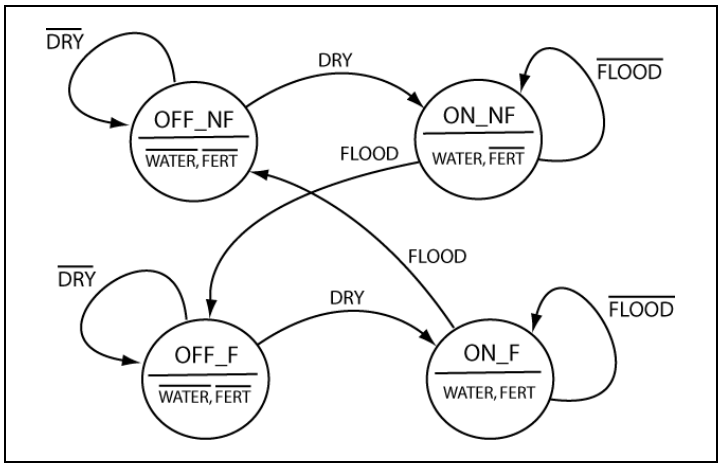
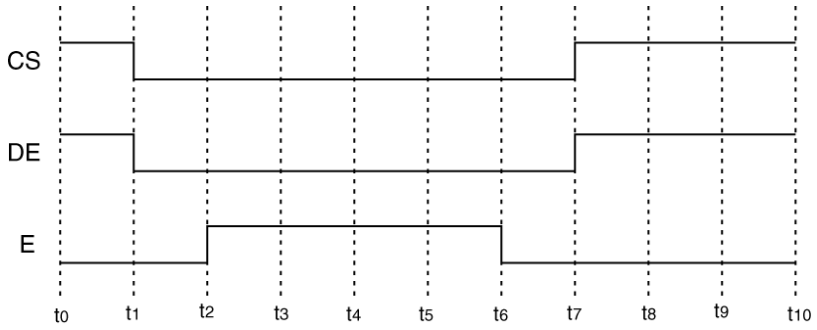


Figure 30.34: The state diagram describing the final FSM.

Example 30-6: Signal Synthesizer

Design a state diagram that models a finite state machine (FSM) that synthesizes the following signals. More specifically, if the FSM receives a GO, the CS, DE, and E, signals are output by the FSM according to the timing diagram provided below. In the diagram below, each of the time increments are equivalent and equal to 10μs (10×10^{-6} s). As part of this problem, you must state the system clock frequency of your FSM.



Solution: As with all problems, the best place to start is to draw a black box diagram of the final circuit. Figure 30.35 shows such a diagram. Note that this diagram shows the GO signal as an input and the three signals synthesized as outputs.

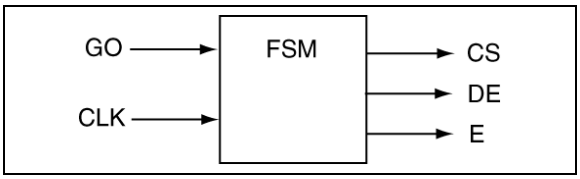


Figure 30.35: Black box diagram of final FSM circuit for this problem.

The next thing we need to establish is the clock frequency used by the FSM. You can calculate this based on the requirements of the problem, namely, the timing diagram provided in the problem description. The first thing to note is that all of the time increments are equal. The next thing to notice is that the output changes seem the fall on the clock edges only which heavily implies that we can use a Moore machine in for the FSM. Reproduction of the timing diagram can be satisfied with a clock period that is equal to the clock period of the time increment. The clock frequency is thus $(10 \times 10^{-6}\ s)^{-1}$, or 100kHZ.

The next part of the problem concerns itself with starting the state diagram. The best place to start is at the perceived beginning, when the FSM is waiting for the GO command (this is the state having the least is going on). So long as the GO command is unasserted, the FSM stays in the initial state. When the GO command asserts on an active clock edge of the FSM, the FSM then proceeds to implement the given signals. From there, each state in the state diagram essential creates one time increment in the state diagram. Figure 30.36 shows the final solution for Example 30-6. Although this is appears to be a large state diagram, hopefully you can see that it is all straightforward.

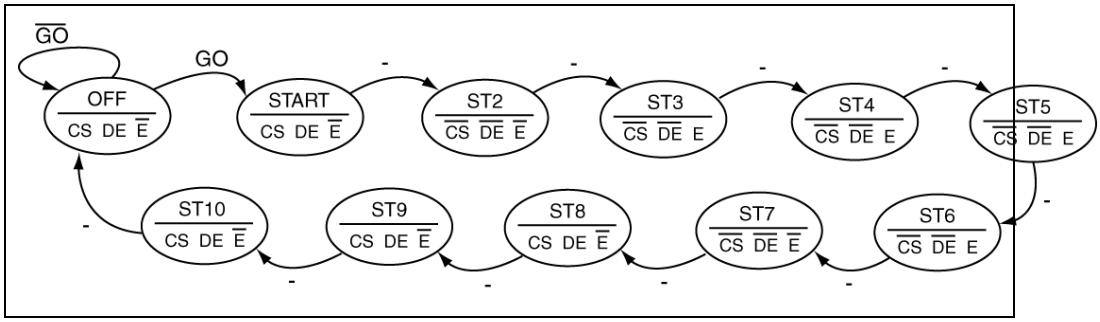


Figure 30.36: The state diagram describing the final solution for Example 30-6.

Problem Post mortem: A couple of worthwhile things to note here:

- First, the output signals in the OFF state are somewhat arbitrary, since the problem statement provided no information as to the state of these signals before the GO signal asserts, we can effectively state the outputs at any value.

- The conditions that govern the transition from the OFF state are based solely upon the state of the GO signal. All of the other state transitions are unconditional transitions. In other words, once the asserted GO signal starts the process, it will step through each of the states on subsequent clock cycles until the state diagram re-enters the OFF state.

- The START state represents the first time increment in the state diagram. Each subsequent state bubble represents individual time increments in the original state diagram.

Example 30-7: **Mixed Duty Cycle Signal Generator**

Design a state diagram that a FSM could use to generate an output signal of different duty cycles. The FSM has a two-bit input signal, SEL, that selects the duty cycles generated by the FSM according to the table below. The period of the final signal must be 40μs so you must state the clock frequency of your FSM. Minimize the number of states you use in your solution.

SEL Input	Duty Cycle
00	25%
01	50%
10	75%
11	100%

Solution: The key to this problem is to understand the problem. As always, your first mode of attack it so draw a black box diagram as shown in Figure 30.37. Note that there is only one output to this circuit: the SIG signal. This signal will exhibit one of the four duty cycles stated in the problem description. The SEL signal is a 2-bit bus, which contains SEL1 and SEL0 with SEL1, being the signal with higher precedence.

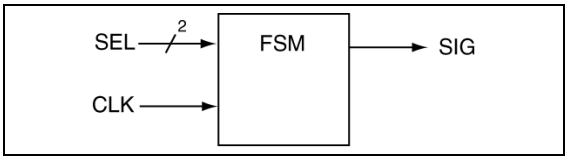


Figure 30.37: Black box diagram for Example 30-7.

The next thing to do is to draw the outputs of the FSM under each of the four conditions. This will once again aid in the understanding of this problem. It also helps you decide a clock frequency for the FSM. Figure 30.38 shows the FSM outputs based on the SEL input. Since the problem states a 40μs period for the outputs, and the required duty cycles effectively divide the 40μs period into four sections, the resultant FSM clock frequency is ¼ the total period, or 10μs. The system clock frequency is thusly the reciprocal of one of these increments or 100kHz.

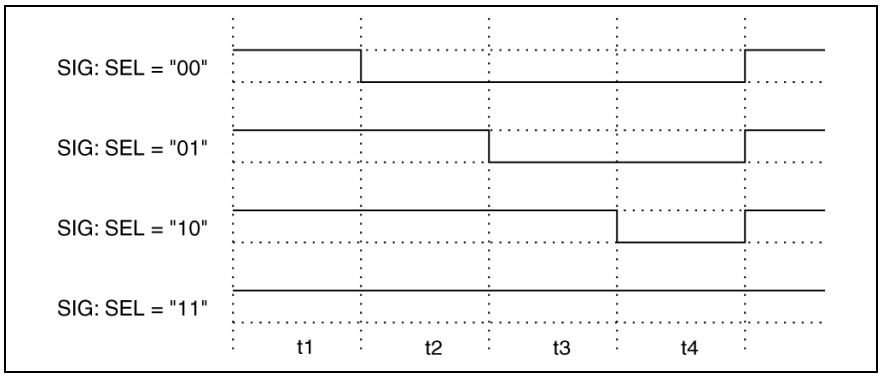


Figure 30.38: How the output should appear under given circuit conditions.

The next thing to do is to start drawing the state diagram. The best place to start on this problem is to pick one of the duty cycles and implement it. The partial state diagram in Figure 30.39 shows an implementation of the 25% duty cycle. Note that in Figure 30.39 there is only one conditional transition while all of the other state transitions are unconditional. This one condition is based directly on the state of the select signals. In other words, if both select signals are not asserted, the state diagram starts down the 25% duty cycle path.

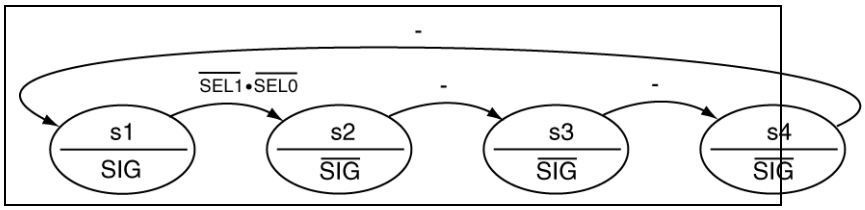


Figure 30.39: Implementation of the 25% duty cycle sequence.

The next step in this solution is to add the 50% duty cycle to the now emerging solution. Figure 30.40 happily shows the new additions to the state diagram. Note that the transition from state s1 to s5 is based on any condition not met in the transition condition from s1 to s2. From the s5 state, if the higher weighted select signal, SEL1, is not asserted, then the state diagram turns off the SIG by transitioning to state s3. State s3 is effectively the final half of the 25% duty cycle transition sequence.

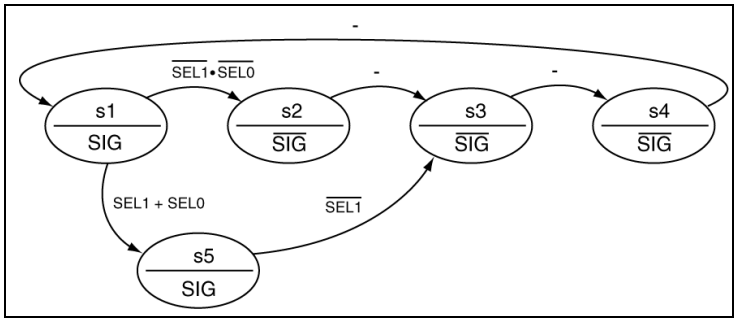


Figure 30.40: Addition of the 50% duty cycle sequence to the 25% duty cycle sequence.

The next step in this solution is to add the 75% duty cycle to the budding state diagram; Figure 30.41 soulfully shows this condition. From the s5 state, if the SEL1 signal asserts, then either the 75% or the 100% duty cycle is selected which as represented by the transition to the s6 state. From the s6 state, if the SEL0 signal is unasserted, the state diagram selects the 75% duty cycle as is indicated by the conditions on the transfer from state s6 to state s4. Note that s4 is the final state of the 25% duty cycle path.

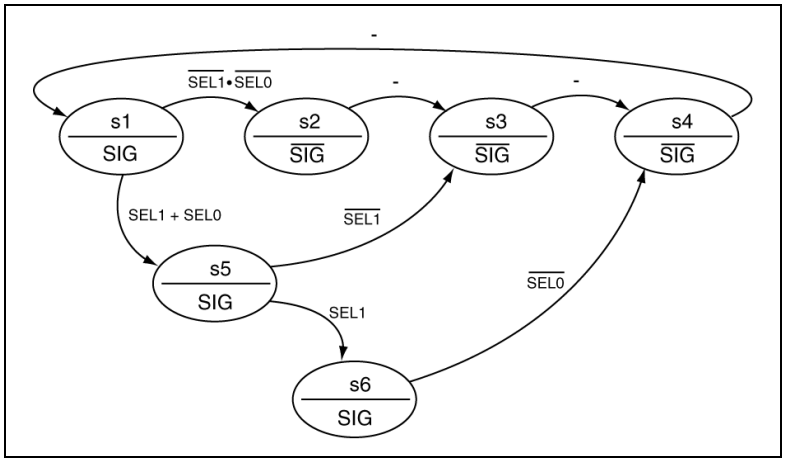


Figure 30.41: Addition of the 75% duty cycle to the emerging state diagram solution.

The final step in the solution is to add in the 100% duty cycle sequence. This requires only the addition of the s7 state. The value of the SEL0 signal controls entry to the s7 state. The FSM exits this state unconditionally and restarts the sequence. The state diagram is now complete and the rejoicing can now begin.

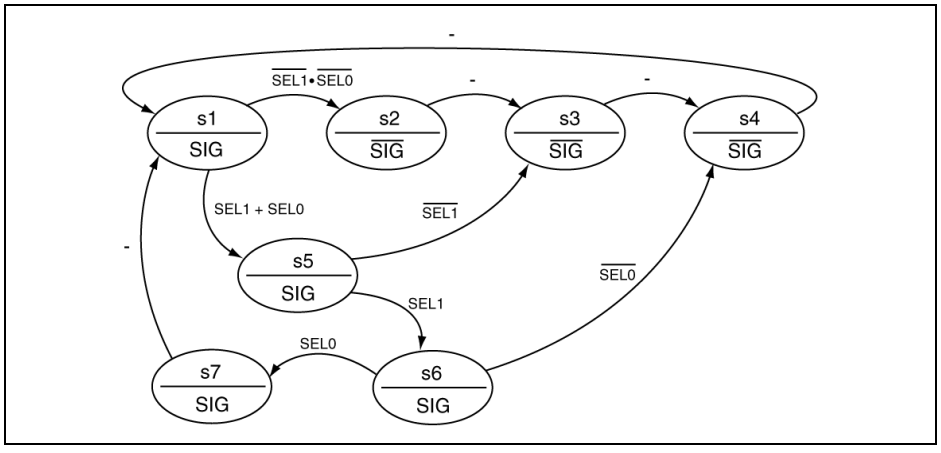


Figure 30.42: The final state diagram including the 100% duty cycle sequence.

Problem Post mortem: A couple things to note here:

- What would happen if the SEL signal changed midway through the state diagram? It would probably cause some glitching as the output settled. So why didn't we address

this condition as we did the problem? The answer is because the problem did not require us to do so. How could we have solved this problem if were asked to do so? The most straightforward solution I can see if to save (register) the state if the SEL input in state s1. The registered values of the SEL inputs would then control the subsequent transitions in the state diagram. This would probably have generated state diagram with more states but it would have been a straightforward solution.

Chapter Summary

- This chapter consisted primarily of example problems. The only verbage worthy of mention here is the set rules on how to get started on a given FSM problem. We list the rules here again in order to make sure this page is not left intentionally blank (figuratively speaking).

  Rule 0: Understand all aspects of FSMs including Mealy vs. Moore machines, timing diagram characteristics, FSM terminology, clocking aspects of FSMs, and state diagram.

  Rule 1: Understand the problem at hand; read the problem many times as problems are typically not stated well.

  Rule 2: Draw a black-box diagram of the problem that clearly shows the inputs to (status signals) and output from (control signals) the FSM.

  Rule 3: Decide whether you'll implement your FSM as a Mealy or Moore machine.

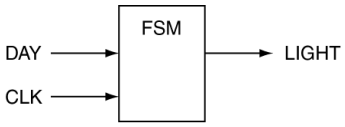
  Rule 4: After you put some thought into Rule 3, draw a legend that is consistent with the decisions you made in Rule 3.

  Rule 5: Draw big circle somewhere on your paper and call that circle your starting state for the FSM. From there, you'll need to start filling in the details. Don't hesitate to simply "get started".

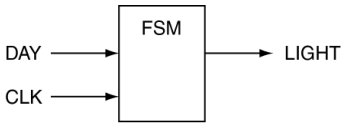
  Rule 6: Don't follow the rules; make up your own rules instead. Break those too.

Design Problems

1) Design a finite state machine that controls a lighting circuit. The FSM has one control input DAY, and one output, LIGHT. The DAY input is from a sensor that indicates whether it is daytime (DAY =' 1') or nighttime (DAY='0'). The LIGHT output turns on the light when equal to '1' and turns off the light then LIGHT = '0'. Design your FSM so that the light turns on every other night. For example, one night the light is on, the next night the light is off. Use either a Mealy or Moore machine.



2) Design a finite state machine that controls a lighting circuit. The FSM has one control input DAY, and one output, LIGHT. The DAY input is from a sensor that indicates whether it is daytime (DAY =' 1') or nighttime (DAY='0'). The LIGHT output turns on the light when equal to '1' and turns off the light then LIGHT = '0'. Design your FSM so that the light turns on every other night. For example, one night the light is on, the next night the light is off. Use either a Mealy or Moore machine.



3) Design a state diagram that describes the operation of a FSM that could be used to control a greenhouse temperature control system.

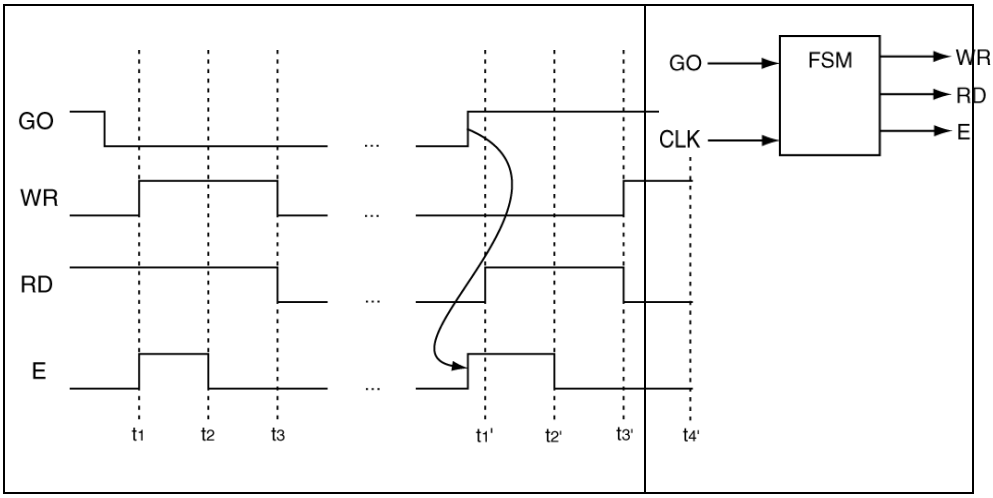
a. The system to be controlled has two outputs, HOT and COLD, which indicate when the temperature of greenhouse is too hot or too cold. If both of these inputs are not asserted, the temperature of the greenhouse requires no action be taken because the control vent is normally partially open. Assume the HOT and COLD inputs will never be simultaneously asserted.

b. The system to be controlled has two inputs, OPEN and SHUT, which are used to completely open or completely close the normally partially open vent. The vent should be opened when the temperature of the green house is too hot and closed when the temperature is too cold.

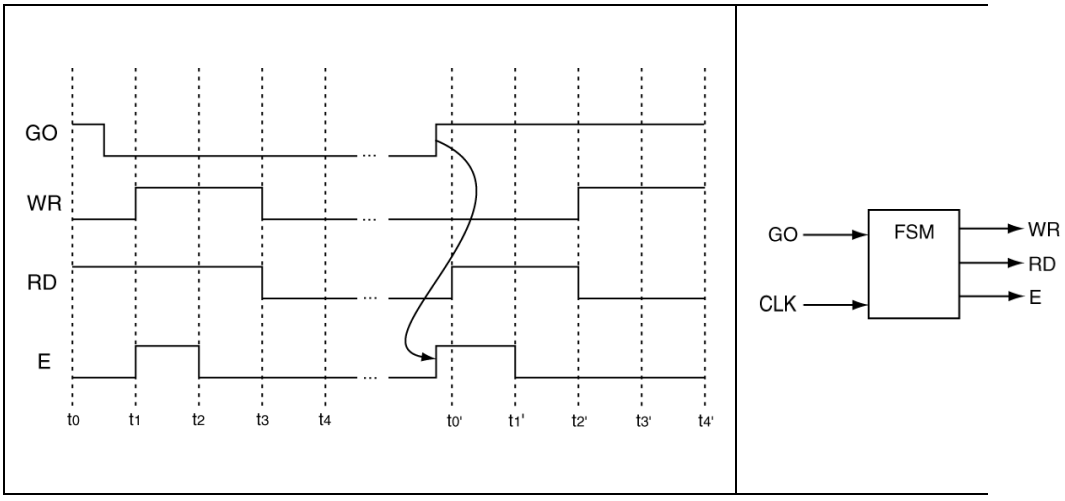
c. Anytime the vent fully opens or closes, it must remain in that state for at least two minutes.

d. The system clock speed of your FSM is real slow: it has a one minute period.

4) For this problem, specify a state diagram what will implement the following timing diagram. In this timing diagram, the GO signal (active low) initiates the synthesis of signals in time slots $t_1 \rightarrow t_3$. After t_3, the FSM waits for GO to be unasserted. When GO is unasserted, the signal in time slots $t_{1'} \rightarrow t_{4'}$ are synthesized. After $t_{4'}$, the FSM returns to the starting state with the outputs indicated prior to time t_1. Your state diagram should also generate the given outputs between t_3 and $t_{1'}$.



5) For this problem, specify a state diagram what will implement the following timing diagram. In this timing diagram, the GO signal initiates the synthesis of signals in time slots $t_1 \rightarrow t_4$. After t_4, the FSM waits for GO to be unasserted. When GO is unasserted, the signal in time slots $t_{0'} \rightarrow t_{4'}$ are synthesized. After $t_{4'}$, the FSM returns to the starting state with the outputs indicated at time t_0.



6) Design a state diagram that describes the operation of a FSM that could be used to control an automobile headlight control system that prevents drivers from unintentionally leaving their headlights on.

a) The FSM issues a control signal, LITES_OFF, when it has determined the driver has unintentionally left the lights on. This signal is used by some other device to turn off the headlights.

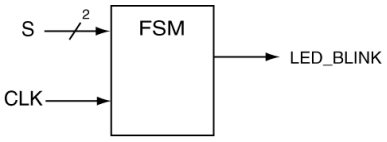
b) The FSM runs in two modes: the 2-minute mode and the 4-minute mode which is determined by the MODE input to the FSM (MODE='0' is the 2-minute mode). The LITES_OFF signal is issued after either two or four minutes depending on the mode.

c) The FSM will only issue the LITES_OFF signal when the both the engine status signal ENG_ON and the DRV_SEAT signals have been unasserted for the set amount of time. Both of these signals are positive logic. If the DVR_SEAT signal is asserted on a clock edge, the timer starts counting again. The ENG_ON signal indicates whether the engine is running (the lights are not turned off if the engine is running) and the DRV_SEAT indicates that someone is sitting in the driver's seat (the lights are not turned off if someone is sitting in the driver's seat).

d) The system clock speed of the FSM is slow: it has a one minute period.

7) Provide a state diagram for a FSM that controls a blinking LED according to the following specifications. The LED is used to visually show the speed setting of a 4-speed motor which is controlled by signal **S** according to the table below. The motor speeds can change asynchronously.

- The LED has two blink periods (both 50% duty cycles): 40ms & 20ms. When the motor is running at its fastest or slowest setting, the LED blink period is 20ms. Otherwise, the LED blink period is 40ms.
- At no time should the LED be ***on*** or ***off*** for more than 20ms.



S	Motor Speed
"00"	slowest
"01"	less slow
"10"	faster
"11"	fastest

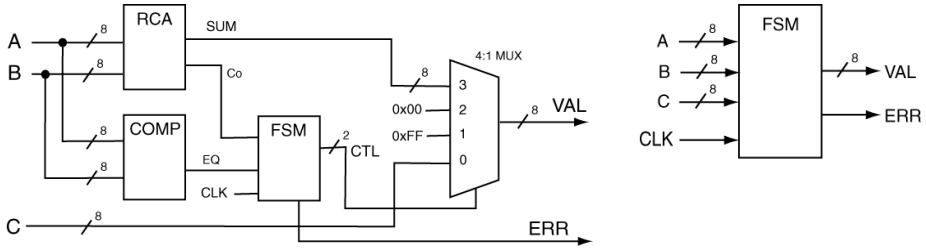
8) Design a FSM is required to control a set of four lights: L3, L2, L1, and L0. The FSM reads the value of five switches, MS, S3, S2, S1, and S0; the S3 switch turns on the L3 light, etc, under the following conditions:

- The other four switches have priorities assigned to them with S3 being the highest and S0 being the lowest.
- The MS switch is considered the master switch and must be actuated in order for any of the lights to be on. When the MS switch is turned off, any light that is on will turn off on the next clock edge.
- No more than one light can be on at any given time. The light that will be turned on at any given time is associated with the switch of highest priority.
- The FSM ensures that the lights must turn on in order. For example if the lowest priority light is on and the highest priority switch is activated, the two lights middle lights will turn on then off one at a time (synchronized to the clock edge) before the highest priority light turns on. The same is true for the opposite direction.

9) A FSM is required to control a sound activated beer dispenser. The system clock for this problem is quite slow: it has a ten-second period. In order to ensure you'll be able to dispense the beer, you thus need to scream loudly for at least ten seconds in order to be screaming on an active clock edge. If you're screaming when the clock edge arrives, the beer will start dispensing. The constraints on this problem are than in any 30-second period, the beer will flow for no more than 20 seconds. In other words, if you were screaming for the entire 30 seconds, you would only receive 20 seconds of beer. After 30 seconds, the system resets and eagerly awaits the next thirsty screamer. This FSM has two inputs: the system clock and the sound sensor. This FSM has only one output: the beer dispenser control. Provide a state diagram that could implement this FSM. Minimize the number of states in the state diagram.

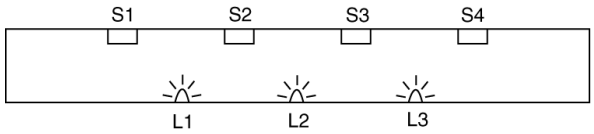
10) Using the listed circuit, design a FSM that outputs the sum of (A + B) as long as no carry is generated. If a carry is present on an active clock edge, the circuit outputs a 0x00 then 0xFF for one clock cycle each, then outputs the C value for at least two clock cycles but for as long as A does not equal B. If and when A equals B, the circuit once again displays the sum of (A+B), etc. The circuit also asserts ERR output whenever the circuit output is not the sum of (A + B).

Assume the FSM clock is much faster than then changes in A, B, and C. Disregard all setup and hold-time issues. You only need to provide a state diagram for this design. *Minimize the number of states in your design.*



11) Design a FSM that creates a power-saving control of a set of hallway lights. The hallway has four sensors (S1,S2,S3,S4) and three lights (L1,L2,L3) as indicated by the diagram below. The sensors indicate when a person is near and causes the nearest light(s) to turn on. When a person first enters the hallway, only one light turns on; as a person walks through the hallway, only the two nearest lights turn on. You need to provide a black box diagram and a state diagram for this design. *Minimize the number of states in your design.* For this problem, make the following assumptions:

- The sensors completely sense the hallway with no overlap in the coverage area
- Only one person at a time will be in the hallway
- When a person enters one side of the hallway, the person will eventually exit on the other side



31 Chapter Thirty-One

(Bryan Mealy 2012 ©)

31.1 Chapter Overview

Designing and analyzing finite state machines (FSMs) represents the majority of the work we've done in the last few chapters. Although we've dealt with some issues related to FSM timing, the emphasis was primarily on the timing details and differences of Mealy and Moore-type outputs. The topic of FSMs is a deep subject and there are many issues we have not dealt with and won't be dealing with in this text. However, there are some issues we'll present in order to provide you with a nice collection of FSM techniques.

The main topic of this chapter is the timing/clocking issues associated with FSM design. The good thing is that these topics apply to all sequential circuits, particularly circuits that use some sort of system clock signal for synchronization purposes. While none of these issues is overly complicated, they are important to creating FSMs that not only work, but also work with the fastest possible clock speeds. The thought is that if your circuit operates with a high clock speed, then it must be a good circuit[339].

Main Chapter Topics

- **SEQUENTIAL CIRCUIT ATTRIBUTES:** Many useful digital circuits contain a system clock. This chapter therefore describes some of the basic terminology associated with clocking signal.
- **PRACTICAL DEVICE ASPECTS:** Digital circuit elements are physical devices and therefore have basic limitations based on device physics. This chapter describes some of the attributes in the context of clocking basic FSM circuits.

Why This Chapter is Important

This chapter is important because it describes some of the more important timing aspects associated with sequential circuits and FSMs.

31.2 Clocking Waveforms

As you know from your previous experience with FSMs, the memory elements in FSM are synchronous circuits. The term synchronous refers to the fact that changes in the state of the flip-flops representing the state variables synchronized to the active clock edge. Up until now, we have not dealt with the clock signals much other than to acknowledge that they exist and that they synchronize changes in the FSM

[339] Although clock speed is a great selling point in digital design land, it generally has little to do with the quality or robustness of your circuit.

state. There are some common clocking terms that everyone in any technical field needs to know of in order to be able to converse with your friends (real and imaginary) at the many parties you generally attend (real and imaginary). This section introduces those terms.

31.2.1 Clocking Waveforms

Probably the most important aspect of clocking waveforms is that the clock signal is generally considered to be periodic. We'll define this in more technical terms later (once you know the more technical terms) but for now we'll define a periodic clock signal as one that does not change in form over time. In other words, no matter where in time you view the waveform, it always appears to have the same form. Figure 31.1 shows both a periodic (CLK1) and a non-periodic waveform (CLK2). Note that you could use either of these signals as the clock input to a flip-flop since they both contain the required rising and falling edge. However, in reality, most of the circuits we'll be dealing with use the more "predictable" waveform of CLK1 over the seemingly random waveform of CLK2.

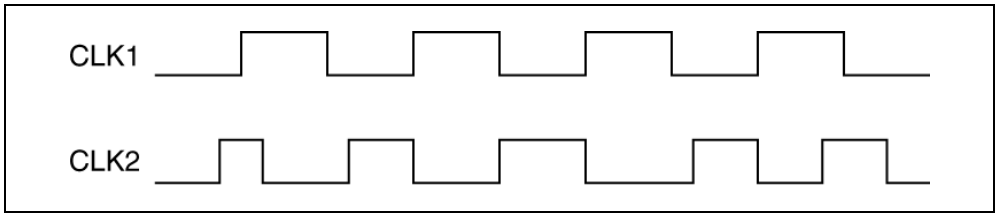


Figure 31.1: A periodic (CLK1) and non-periodic (CLK2) waveform.

31.2.2 The Period

The more technical definition for a periodic waveform is that the waveform repeats itself "every so often". The *period* of the waveform indicates the amount of time required for the waveform to repeat itself. Keep in mind that "time" is the units associated with period of a waveform. Figure 31.2 shows a periodic waveform with one of the periods clearly delineated. This waveform is considered periodic because the waveform between (a) and (b) is the same as the waveform between (b) and (c). Clocking waveforms generally use the variable T to represent the time required for the waveform to repeat itself as shown in Figure 31.2.

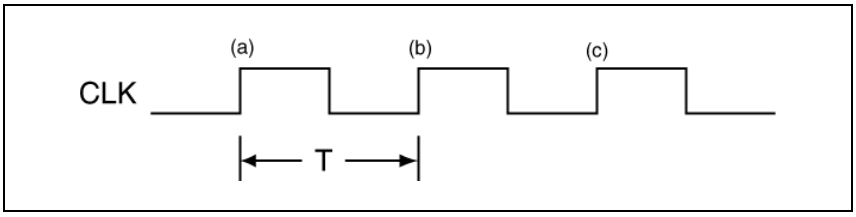


Figure 31.2: The variable T is typically used to represent the period of a waveform.

31.2.3 The Frequency

Although the period is a useful measurement, it is not always the best approach to describing a periodic signal. Often times we may not be specifically interested in the period of the waveform, but we are interested in how many times the waveform repeats itself in a given space of time. The *frequency* of the waveform represents the number of times a signal repeats itself over a given amount of time. This definition is actually somewhat more generally then we usually work with so we want to refine it somewhat to make it more usable. The space of time we're usually interested in is one second (1s); the

standard used most often in technical pursuits is the number of time a signal repeats itself in one second of time. Using this one second time slot simplifies the translation of period to frequency.

Period and frequency have a reciprocal relationship when the amount of time considered is one second; Figure 31.3 shows these relationships. The units used for frequency are generally Hertz, or Hz for short. The term Hertz is technically defined as the number of *cycles per second* (or just "cycles per second"), which refers to the number of times a given signal repeats itself over time. The term Hertz has units of $\mathbf{s^{-1}}$, which underscores its reciprocal relationship to the period, which is measured in units of time.

$Period = T = \frac{1}{frequency} = (frequency)^{-1}$	$frequency = \frac{1}{Period} = \frac{1}{T} = (T)^{-1}$
Units: time (seconds)	Units: Hz $(seconds)^{-1}$
(a)	(b)

Figure 31.3: The calculations and units for Period and Frequency.

Example 31-1

A given waveform has a 40ns period. What is the frequency of this waveform?

Solution: Taking the reciprocal of the period provides the frequency as shown by the following calculation:.

$$frequency = \frac{1}{40ns} = \frac{1}{40x10^{-9}} = 25x10^{6}\,Hz = 25MHz$$

Example 31-2

A given waveform has a 50M Hz frequency. What is the period of this waveform?

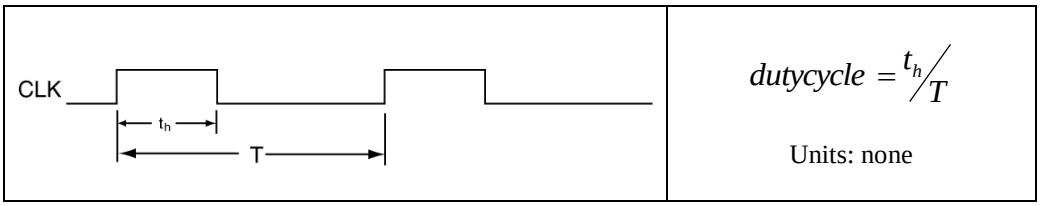
Solution: Taking the reciprocal of the frequency provides the period. You can find the entire calculation below.

$$Period = T = \frac{1}{50MHz} = \frac{1}{50x10^{6}\,s^{-1}} = 20x10^{-6}\,s = 20\mu s$$

31.2.4 Periodic Waveform Attributes

Now that we've established that we're interested in periodic waveforms; let's now describe some of the attributes associated with periodic waveforms. All the periodic waveforms we've dealt with up to now have been symmetrical; this means that the portion of time the signal was high was equal to the portion of time the signal was low. These equivalent times are not always the case. Sometimes the clock signal high times and the clock signal low times are not equivalent. In these cases, we use the term *duty cycle* to describe the waveform.

In rough terms, the duty cycle refers to the percentage of the period that the signal is in its high state. In technical terms, the duty cycle is the ratio of the time the signal is high to the period of the signal. Figure 31.4(a) shows the official looking equation for duty cycle. Note that since the duty cycle refers to a ratio, there are no units associated with duty cycle metric.



(a) (b)

Figure 31.4: Duty cycle calculations and units.

Example 31-3

A waveform with a 25% duty cycle is high for 12.5ns. Find the frequency of the waveform.

Solution: If the waveform is high 25% of the period, than 12.5ns represents ¼ of the period. The entire period is then four times longer than the amount of time the signal is high; therefore, the period of the waveform is 50ns. The frequency is the reciprocal of the period, or 20MHz.

31.3 Practical Flip-Flop Clocking

Most of our FSM discussion thus far dealt with the notion of idealized flip-flops, which allowed us to focus on the basic functioning of the devices. Now that we're familiar with the basic function of flip-flops, we need to take a look at some of the practical aspects of working with flip-flops. Namely, what we're interested in here are timing considerations that must be taken into account in order for our sequential circuits to work properly with increasing clock speeds. Many factors will prevent our circuits from working properly so our focus will be on two major timing considerations that you'll more than likely run into as you continue your journey deep into digital design-land.

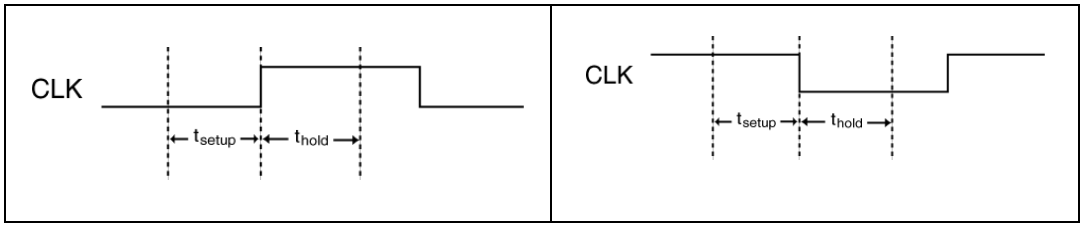
Recall that flip-flops generally have control inputs (namely the D, T, and JK inputs) and clock inputs. Flip-flops are synchronous circuits in that the change in the flip-flop's output are synchronized to an active clock edge. As it turns out, there are more factors involved when using actual flip-flops. Several issues can arise because flip-flops are actual semiconductor devices, which have many insidious factors

associated with them that will make the operation of the flip-flop less than ideal. In other words, things don't happen immediately with flip-flops; signals need to propagate through flip-flops, the flip-flop trolls need to be fed, etc.

One of the consequences of practical flip-flop clocking is that you need to be nice to the flip-flop's control inputs (temporally speaking) near the active clock edge. More specifically, the control input generally needs to remain stable for a given amount of time both before and after the active clock edge. The amount of time the control input needs to remain stable before the active clock edge is referred to as the *setup time* and the amount of time the control input needs to remain stable after the active clock edge is referred to as the *hold time*. A timing diagram best shows these metrics; Figure 31.5 shows a timing diagram associated with a flip-flop clocking signal.

Figure 31.5(a) and Figure 31.5(b) show the setup and hold times associated with a rising-edge and falling-edge triggered flip-flop, respectively. The control input (such as the "D" input of a D flip-flop) must be stable (it must not change) for the duration of the setup time. The control input of the flip-flop must also be stable for the duration of the hold time. If the control input were to change during these time intervals, the output, and thus, the state of the flip-flop would be indeterminate.

Out there in digital-land, it is well known that if you *violate* a setup or hold time, your flip-flop stands the chance of becoming *metastable*[340]. This means that the output of the device will be neither high nor low; it will be somewhere in-between and it may stay there for an extended length of time. In the context of a digital circuit, this would be un-good.



(a) (b)

Figure 31.5: Setup and hold time definitions for rising edge (a) and falling edge (b) triggered flip-flops.

Setup and hold times are associated with many different types of digital circuits. The idea is always the same: keep a signal stable for a given amount of time before and after some critical clock edge. Since flip-flops are what we know best, we'll center our discussion around them. There is not too much more to say about setup and hold times. What we'll do now is consider some other practical aspects of a sequential circuit which use the setup and hold times. But, mark my words… someday you'll be working on a circuit that does not seem to want to work properly. You'll toil over it for awhile and then it will hit you: *you violated a setup and/or hold time*. You'll do a quick redesign on your circuit, it will magically work, and you'll have saved the day once again.

31.4 Maximum Clock Frequencies of FSMs

In this modern age, faster is generally associated with better even though this is usually not the case in real life. In order to provide you with a deeper understanding of digital circuits, and in particular sequential circuits, we need to take a closer look at some of the timing aspects. Namely, for a given

[340] And yet again, a digital design word makes it out of digital design land. The word metastable is often used to describe people who are unpredictable; the type you'll do best to steer clear of.

circuit, there is always a question of how fast you can *clock* the circuit and still have the circuit operate properly. In other words, what is maximum frequency that the flip-flop clock can run at without hindering the operation of the circuit by violating nasty things such as setup and hold times.

As a handy reminder, Figure 31.6 shows a model of a Moore-type FSM. Each of the given boxes is comprised of either sequential or combinatorial logic. As you know, there are propagation delays associated with all types of logic[341]; as you just found out, there are factors such as setup and hold times associated with sequential logic. From the diagram of Figure 31.6, you should sense that the circuitry contained in the various boxes is going to lower the maximum rate at which the FSM can operate. This includes both the sequential elements and the Next State Decoder. The Output Decoder generally has no effect on the maximum clock frequency so we'll not need to consider it here. What does matter is the propagation delay though the Next State Decoder, the setup times associated with the flip-flops and some combination of the flip-flops hold time and/or the propagation delay through the flip-flop. These items require time: as the time accumulates, the shortest period possible for the clock signal becomes greater, and hence, the maximum clock frequency becomes lower.

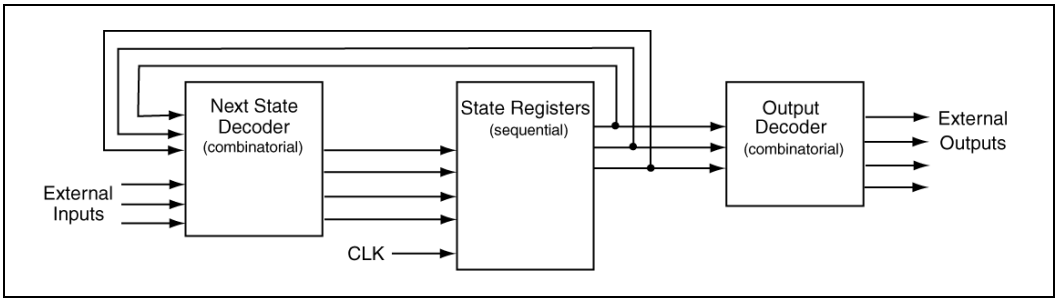


Figure 31.6: Model for a Moore-type FSM.

In order to simplify the analysis of FSM circuits, we'll also make some other assumptions about this circuit. For a given flip-flop, we know we have both a hold time and a propagation delay time that we need to deal with. For these problems, we'll assume that the propagation delay for the flip-flop is greater than the hold time. This allows the exclusion of the hold time from the calculation. Once again, the only factors affecting the maximum clock frequency (or minimum period) for the circuit are the setup time, the propagation delay through the Next State Decoder, and the propagation delay through the flip-flops. Figure 31.7 provides a visual representation of these attributes.

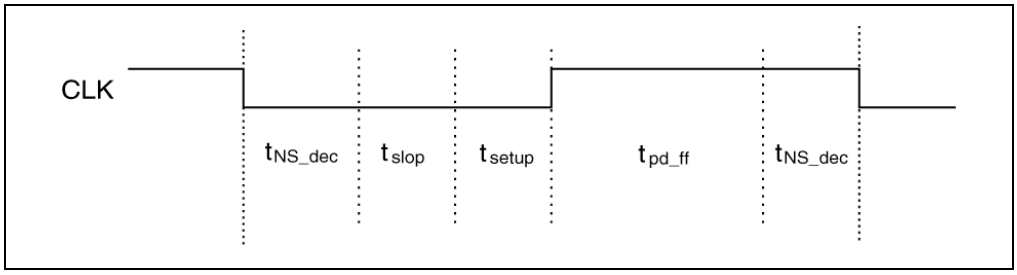


Figure 31.7: Model for a Moore-type FSM.

Figure 31.7 shows four time slices that we need to consider in the context of maximum clock frequencies. Despite being shown twice, there is only one t_{NS_dec}. We show this value twice because it is a continuation from the portion of the waveform ending with the falling edge on the right side of the

[341] Keep in mind that sequential circuits were basically combinatorial circuits that contained feedback paths from the circuit outputs to the circuit inputs.

diagram to the portion of the waveform starting with the falling edge on the left side of the diagram. Another factor include in this diagram is the t_{slop} value. The idea here is that you never want to design to the absolute operating boundaries of your circuit; you always want to throw in a safety margin to guard against circuit conditions that may adversely affect the circuit[342]. In the end, we'll use these four values to calculate the minimum period as shown in Figure 31.8. Figure 31.8 shows once again that the minimum period is the reciprocal of the maximum clock frequency.

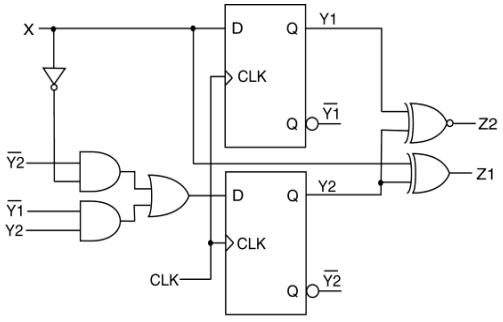
$$T_{\min} = t_{NS\_dec} + t_{slop} + t_{setup} + t_{pd\_ff}$$

$$Frequency_{\max} = 1 / T_{\min}$$

Figure 31.8: Official calculations for minimum period and maximum clock frequency.

Example 31-4

What is the maximum system clock frequency at which the following sequential circuit can operate? For this problem, the flip-flops have a setup time of 10ns and a propagation delay of 13ns. Inverters have propagation delays of 6ns and logic gates have propagation delays of 8ns. For this problem, add a safety margin of 12ns. Assume the propagation delay for the flip-flops is greater than the hold time. Assume the X input is stable and the Z1 and Z2 outputs drive a circuit that is not sensitive to the maximum clock frequency.



Solution: The first thing to notice about this problem is that we don't need to worry about the X input because the problem states that the X input value could be considered stable. The problem also stated that the Z outputs are yet another item we don't need to worry about. What we need to do for this problem is total up the gate delays on the longest path in the excitation logic in order to give us the t_{NS_dec} value. Since there are two gates (one AND gate and one OR gate), in the longest path through the Next State Decoder, the t_{NS_dec} value is twice a standard gate delay. The safety margin of 12ns makes of the t_{slop} value. Figure 31.9 shows the final solution for this example.

[342] These factors would include ambient temperature variations and variations in the device itself.

$$T_{\min} = t_{NS\_dec} + t_{slop} + t_{setup} + t_{pd\_ff}$$

$$T_{\min} = (8ns + 8ns) + (12ns) + (10ns) + (13ns) = 51ns$$

$$Frequency_{\max} = 1/T_{\min} = 1/51ns = 19.6MHz$$

Figure 31.9: The calculations: plug and chug.

Chapter Summary

- Waveforms in digital design are usually periodic in nature. Periodic signals can be described by a given waveform that repeats itself after a given amount of time referred to as the period of the signal. Periodic signals are also described by the frequency which is defined to be the reciprocal of the period.

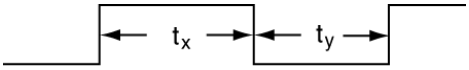
- Periodic waveforms are also described by their duty cycles which is defined to be the ratio of the time in the period that the signal is in a high state to the period of the signal.

- All clocked digital devices have physical attributes that govern their performance. Two of the attributes typically associated with sequential digital circuits are the setup and hold times. The setup time is the amount of time that an input signal needs to remain stable before the active clock edge of a device. The hold time is the amount of time that the input signal needs to remain stable after the active clock edge.

- On major concern of FSMs is the maximum clocking frequency that the FSM can use while not compromising the operation of the FSM. Using a simple model, the maximum clock frequency is a function of the propagation delay of the next state decoder, the propagation delay of the flip-flop, the setup time of the flip-flop, and usually some margin of safety.

Chapter Problems

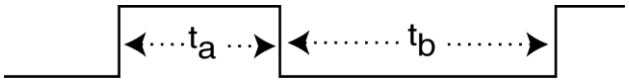
1) For the system clock signal displayed below with t_x=30ns and t_y=25ns, find the period, frequency, and duty cycle of the waveform. (1ns = $1x10^{-9}$ seconds)



2) A system clock signal with a 70% duty cycle is in a high state for 14ns of its period. What is the period and frequency of the clock? (1ns = $1x10^{-9}$ seconds).

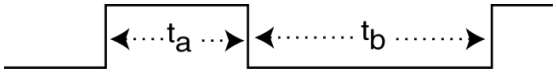
3) A system clock if running at 50M Hertz. What amount of time is the signal high if the system clock has a 40% duty cycle? (1 M Hertz = $1x10^{6}$ Hertz)

4) The following clock waveform is in a low state for a 80% of the period. Find the duty cycle, period, and frequency (its OK to only setup the frequency calculation).



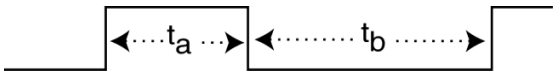
t_b = 20ns

5) The following clock waveform is in a *high* state for a 40% of the period. Find the duty cycle, period, and frequency (its OK to only setup the frequency calculation).



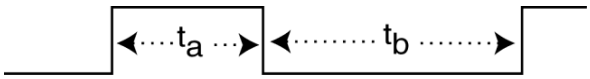
t_a = 20ns

6) The following clock waveform is in a *high* state for a 40% of the period. Find the duty cycle, period, and frequency (its OK to only setup the frequency calculation).



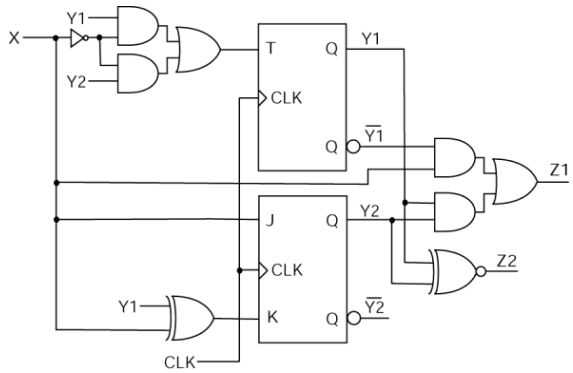
t_a = 20ns

7) The following clock waveform is in a *low* state for a 20% of the period. Find the duty cycle, period, and frequency (its OK to only setup the frequency calculation). The diagram is not drawn to scale.



t_b = 60ns

8) What is the maximum clock frequency that can be used by the following circuit? For this problem, add a safety margin that is 10% of the minimum clock period based on the timing values stated below. Assume the output Z drives a circuit that is not sensitive to the maximum clock frequency. Use the listed circuit parameters for this problem:



flip-flop propagation delay: **20ns**

inverter propagation delay: **4ns**

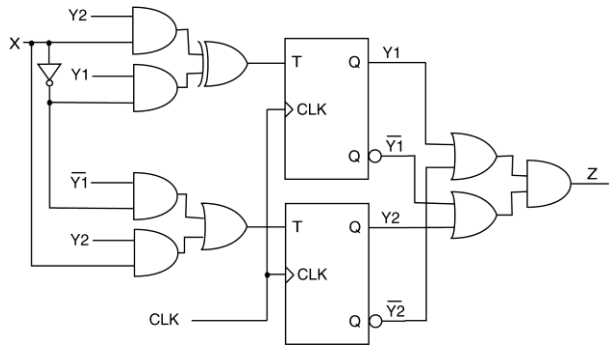
gate propagation delay: **10ns**

flip-flop set-up time: **6ns**

flip-flop hold time: **7ns**

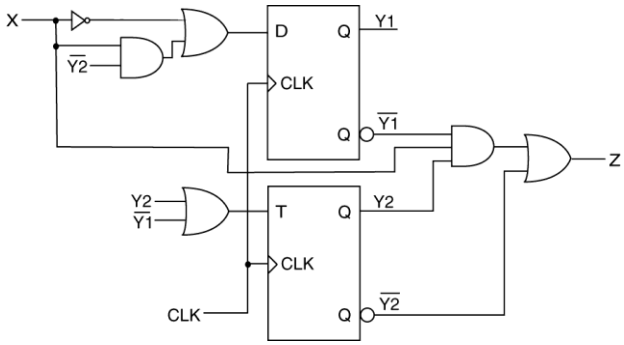
9) For the previous problem, you now need to add a margin of safety to the clocking operation of the circuit. Redo problem 7 and add a 20ns margin of safety, t_{slop}, to the minimum clock period. What is the new minimum clock period and new maximum clock frequency?

10) The following circuit was designed to operate at 20MHz ($20x10^6$Hz). Under these conditions, how much of a safety margin (*if any*) has been added to the circuit? Assume the X input is stable and the output Z drives a circuit that is not sensitive to the maximum clock frequency. Also assume that the propagation delay of the flip-flops is much greater than the flip-flops set-up time. Use the listed circuit parameters for this problem:



flip-flop propagation delay: **17ns**
inverter propagation delay: **4ns**
gate propagation delay: **9ns**
flip-flop set-up time: **8ns**
flip-flop hold time: **7ns**

11) What is the maximum clock frequency that can be used by the following circuit? For this problem, add a safety margin that is 20% of the minimum clock period based on the timing values stated below. Assume the output Z drives a circuit that is not sensitive to the maximum clock frequency. Use the listed circuit parameters for this problem:



flip-flop propagation delay: **20ns**
inverter propagation delay: **4ns**
gate propagation delay: **8ns**
flip-flop set-up time: **5ns**
flip-flop hold time: **7ns**

32 Chapter Thirty-Two

(Bryan Mealy 2012 ©)

32.1 Chapter Overview

In a perfect world, we would all know exactly what we'll need to know for that next interview question. In addition, if we knew just what we needed to know, we would have no use for stuff we didn't need to know (because in a perfect world, no one would be impressed with the trivial crap that we typically blurt out[343]). It may turn out that you never need to know the information in this chapter and thus this is all an academic exercise. Then again, it may turn out that you do need to know this material, either for an interview or for an actual FSM implementation in a place where it really matters (such as on the job).

On a better note, if you need to learn this material, it will no doubt add to your basic knowledge of FSM implementations as well as reinforcing your basic understanding of the standard types of flip-flops. As with everything, it all can't be all that bad. Once again, the only challenging part about designing state machines is generating the state diagram; but low-level implementation details are equally as fun.

Main Chapter Topics

- **"NEW" FSM IMPLEMENTATION TECHNIQUES:** These techniques overcome some of the basic limitations in the "classical" approach we've used until now. There are a few drawbacks of these techniques but there are also a few techniques to minimize these drawbacks.

Why This Chapter is Important

This chapter is not that important; it's sort of interesting because provides some interesting information regarding FSM implementations and associated low level details.

32.2 FSM Modeling Using New Techniques

Up until now, we have taken two approaches to learning the ins and outs (pun intended) of finite state machines: analysis and design. The approach we took in these types of problems was well structured which hopefully helped to offer some insights into the workings of the FSM. As you probably also noticed, the approach was somewhat tedious. The only redeeming feature for this approach to FSM problems was the fact that the problems were not overly complex. The key word in the previous sentence is "complex". The characteristic that made the approach complex was the fact that all of the

[343] Academic administrators, however, are always impressed with trivial crap, particularly self-generated trivial crap.

inputs to the Next State Decoder (see Figure 32.1) appeared as independent variables in the resulting PS/NS table.

Figure 32.1 shows a block diagram of a FSM (Moore-type). So long as the inputs to the Next State Decoder block remained relatively few, the resulting PS/NS table remained relatively small and the FSM problem remained relatively doable. The problem here is that complex FSMs are simply not doable using this technique; as you know, even simple FSM implementations using this technique were somewhat error prone due to the many steps involved in the problem.

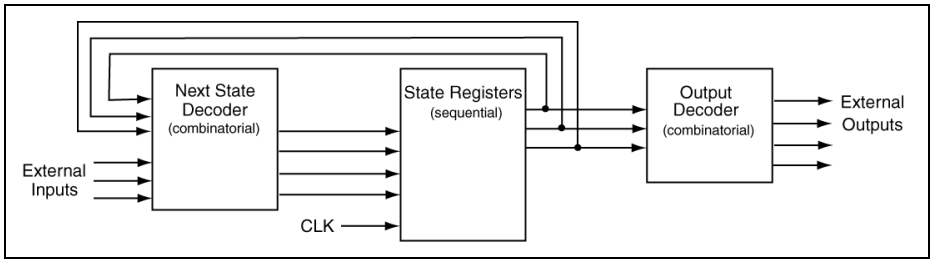


Figure 32.1: A block diagram showing a model for a Moore-type FSM.

The approach we've used to implement FSMs up until now (not including the VHDL behavioral modeling approach) is sometimes referred to as the *classical* approach. Although it is instructive and interesting (and tedious), we need to come up with other techniques to allow us to work with more complex problems.

For the *New* techniques we describe in this chapter, we'll still be outside the realm of VHDL behavioral modeling of FSMs. While these new techniques extend the range of problems that we can implement with direct modeling of FSMs, they are still not as powerful as VHDL behavioral modeling. The bottom line is that if your project could use VHDL behavioral modeling to implement the FSM, that's the technique you would use. On the other hand, if VHDL modeling were not available on a certain project, you would need to resort to one of the direct FSM implementation techniques.

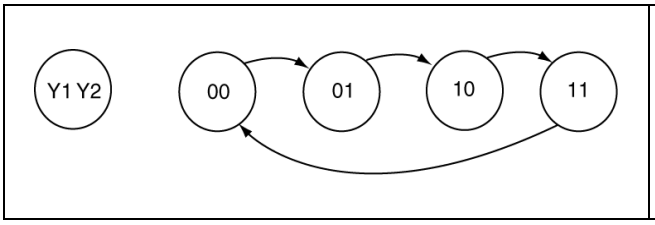
The "new" FSM techniques allow us to move past the grunt limitations presented by the classical FSM approach (PS/NS table and K-map-based). We'll first introduce the motivation behind these techniques and then present a few examples. As you'll see, these new techniques have their basis in the standard D, T, and JK flip-flops (for better or worse).

32.3 Motivation for the New FSM Modeling Techniques

The counter is one of the most basic sequential circuits and should be quite familiar to you. We'll therefore use it as a basis to introduce these new FSM implementation techniques. Consider the simple counter design modeled by the state diagram (including legend) shown in Figure 32.2(a) and the associated PS/NS table shown in Figure 32.2(b). The state diagram and PS/NS table describe a 2-bit counter that counts in a normal binary sequence. Literals Y1 and Y2 arbitrarily represent the state variables. For this example, we'll only be interested in the Y1 variable; in other words, we'll only be interested in the Y1 state transitions, or $Y1 \rightarrow Y1^{+}$.

The reason we're so interested in the Y1 state transitions is because for this counter, every possible transition for a single-bit storage element is represented: ($0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 1$, and $1 \rightarrow 0$). The Y2 variable is included in the following documentation but we don't use it as part of the motivation for this

technique because not all transitions are represented by Y2. For this example, we'll be generating the excitation inputs for D, T, and JK flip-flops.



(a)

Y1	Y2	$Y1^+$	$Y2^+$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

(b)

Figure 32.2: State diagram and PS/NS table for simple counter.

32.3.1 New Technique Motivation: D Flip-flops

Figure 32.3 shows the PS/NS table including the excitation data associated with the use of D flip-flops for the Y1 and Y2 variables. Common sense dictates that you label the D flip-flops D1 and D2 for the Y1 and Y2 variable, respectively. Note that the D1 and D2 columns match the corresponding next-state variables, which are necessarily true for D flip-flops[344]. From this point, using the classical FSM approach, you would then drop the D1 and D2 excitation data into K-maps and generate the associated excitation equations in reduced form. For this example, we'll make an important observation regarding the Y1 transitions which will allow us to bypass the use of K-maps for generating the associated excitation equations.

Recall that for a K-map, we would be interested in grouping the 1's and writing an equation for all the subsequent grouped product terms. Looking at this in a different way, the 1's in the excitation data are rows in the truth table that we're interested in. Looking at the 1's in the D1 column of Figure 32.3 indicates that the 1's of the circuit *occur* under only two specific conditions. Figure 32.3 highlights these conditions, which we refer to as the "Set" and "Hold-1" transitions. Remember, the $Y1 \rightarrow Y1^+$ transitions are important because the Y1 variable represents every possible transition.

Y1	Y2	$Y1^+$	$Y2^+$	D1	D2	
0	0	0	1	0	1	
0	1	1	0	1	0	"Set" transition
1	0	1	1	1	1	
1	1	0	0	0	0	"Hold 1" transition

Figure 32.3: The D FF: the implicants of the D1 function.

At this point, if we simply list the minterms for the Set and Hold-1 conditions (the 1's) for the D1 column, we'll have an equation that represents the excitation logic for the variable in question. In other words, this step involves writing a product term for each row in the table that contains a '1'. One important item to note with this approach is that since we have not utilized a K-map, the resulting equations generated from this technique will not necessarily be in reduced form. Equation 32-1 shows

[344] This is because in the excitation equation for a D flip-flop, $Q^+ = D$, the next-state is not a function of the present-state.

the resulting excitation equations for the D1 and D2 columns. Note that the product terms in the D1 and D2 equations are *standard* product terms and since each term contains one of each of the independent variables. In that the terms in these equations are standard product terms, it should be obvious that these equations are not in reduced form.

$D1 = \overline{Y1} \cdot Y2 + Y1 \cdot \overline{Y2}$	$D2 = \overline{Y1} \cdot Y2 + Y1 \cdot Y2$

Equation 32-1: The excitation equations for the D1 and D2 variables.

As a final comment, also note that the excitation equations for D1 and D2 could have been written by inspection. D1 is an exclusive OR function of Y1 and Y2 while D2 is equivalent to the complemented Y2 state variables. It's always good to apply a smattering of horse-sense when working with design problems such as these.

The observation of the Set and Hold-1 characteristics essentially allows us to go directly from the state diagram to the excitation equation; we used the PS/NS table shown in Figure 32.3 only for motivational considerations. The only possible drawback here is that the resulting excitation equations are not in reduced form. From this point, you could reduce the equations by inspection, by basic Boolean algebra techniques, or find some software that will do the reductions for you.

32.3.2 New Technique Motivation: T Flip-flops

We can also derive a similar technique for T flip-flops using the same approach as we did for the D flip-flop case. We'll reuse the counter described in Figure 32.2 for the T flip-flop motivation. The 1's of the T1 flip-flop are once again what we're interested in representing in Boolean equation form. Recall that the $Y1 \rightarrow Y1^+$ transition represents all of the possible transition cases so this is where we'll put the focus of this discussion as is indicated in PS/NS table shown in Figure 32.4.

The difference between the T the D flip-flop is that 1's are present in both the "Set" and "Clear" conditions as indicated in Figure 32.4. For the D flip-flop, the 1's were present in the "Set" and "Hold-1" conditions. Use of classical FSM techniques would place the T1 column into a K-map and reduce it. We instead once again simply list the standard product terms associated with the "Set" and "Clear" transitions. Equation 32-2 shows the resulting excitation equations written for the Y1 and Y2 flip-flops.

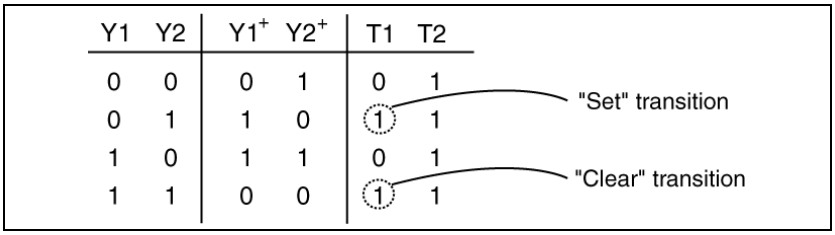


Figure 32.4: The T FF: the implicants of the T1 function.

$T1 = \overline{Y1} \cdot Y2 + Y1 \cdot \overline{Y2}$	$T2 = 1$

Equation 32-2: The excitation equations for the T1 and T2 variables.

32.3.3 New Technique Motivation: JK Flip-flops

Finally, we can derive a similar approach for the JK flip-flop by applying the same type of arguments as we applied to the D and T flip-flops. The approach for the JK flip-flop is probably the trickiest of the three flip-flops. The reason for the tricks is that the JK flip-flops are arguably more versatile than the D and T flip-flops. Figure 32.5 shows the excitation data for the JK flip-flop for the example counter problem. In the classical approach, we use K-maps to simplify the resulting excitation data and we place as many "don't cares" in the table as possible.

Our approach now is to omit as many 1's as possible from the table because we must write a product term for each '1' that appears in the excitation data (the J and K columns of the PS/NS table). For this reason, we fill in the J and K excitation data in such as way as to make the required transition happen, but also minimizing the number of 1's in the corresponding column. For example, for the 0→ 0 transition in the first row of the truth table in Figure 32.5, we could have place a "don't care" in the K1 column (as is found in the excitation table for a JK flip-flop) because both a JK = "00"or a JK = "01" would cause a 0→ 0 transition. However, placing a "00" in the columns reduces the number of 1's in the circuit, which results in saving us time and effort in this new approach.

The same argument can be made for the other three rows of the truth table keeping in mind that the excitation table for a JK flip-flop contained a "don't care" in each of the rows. After applying this approach to including as few 1's as possible in the PS/NS table, we'll see that the only 1's that appear are in the "Set" transition for the J excitation input and the "Clear" transition for the K excitation input. Equation 32-3 lists the resulting equations for the J and K inputs for only one of the two JK flip-flops.

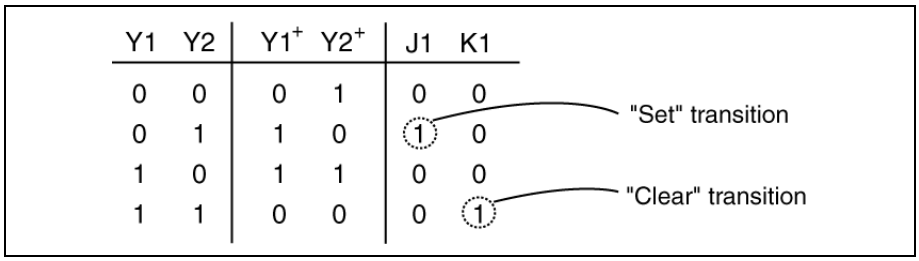


Figure 32.5: The JK FF: the implicants of the J1 and K1 function.

$J1 = \overline{Y1} \cdot Y2 = Y2$	$K1 = Y1 \cdot Y2 = Y2$

Equation 32-3: The equations for the J1 and K1 inputs.

As you may have noticed in Equation 32-3, the resulting equations applied a special type of reduction that is unique to the JK flip-flops. It is not obvious, however, why such a reduction is valid. Understanding why this reduction can be applied is important to understand because if requires that you intuitively understand the characteristics of the JK flip-flops. The logic behind this special JK reduction is below.

- Special J Reduction: because we listed the '1' in the excitation data for the J input based solely on a set transition in the $Y1 \rightarrow Y1^+$ state variable, we know the current state of Y1 is a '0'. If the current state of Y1 is a '0', then the complement of Y1, or $\overline{Y1}$, must be a '1'. Because we know in this case that $\overline{Y1}$ must be in the '1' state, the equation $J1 = \overline{Y1} \cdot Y2$ is written as

$J1 = 1 \cdot Y2$ which we'll reduce to J1 = Y2. Keep in mind that the excitation inputs are based on the present state of the state variables as opposed to the next state values.

- Special K Reduction: because we listed the '1' in the excitation data for the K input based on a clear transition $(1 \rightarrow 0)$ on the $Y1 \rightarrow Y1^+$ state variable, we know that the current state of Y1 is necessarily a '1'. If the current state of Y1 is a '1', then the equation $K1 = Y1 \cdot Y2$ can be written as $K1 = 1 \cdot Y2$ which further reduces to K1 =Y2 by using the Boolean algebra thing.

The punch line to this example is as follows: we've shown that we have method to generate the excitation equations without taking the truth table route as we did in the classical FSM approach. All we need to do is list the conditions and state that cause a '1' to appear in the excitation logic column for the D, T, and JK flip-flops. Table 32.1 shows these conditions in closed form. It should be no surprise that we refer to these three methods as the "Set or Hold-1", the "Set or Clear", or the "Set-Clear" methods, respectively. Although these names are not overly imaginative, they are somewhat instructive.

The motivation example was simple because all of the state transitions in the original state diagram were unconditional. As you would probably expect, the input conditions will also be included in the generated equations for a more complex example. Officially speaking, Table 32.1 shows the closed form equations associated with these new FSM implementation techniques. Note that for our motivations example, our final excitation equations only included the PS (present state). As the equations in Table 32.1 indicate, if there were input conditions controlling the state transitions, they would be ANDed with the PS term and included in the final equation. Note also that the equations of Table 32.1 use the summation symbol; but since we're talking about digital logic here, logical addition is implied (OR function) as opposed to standard mathematical summation.

Equation Type	Closed Form Equation
D excitation equation:	$\sum(PS \cdot \text{input conditons associated with "Set" transitions}) +$ $\sum(PS \cdot \text{input conditons associated with "Hold-1" transitions})$
T excitation equation:	$\sum(PS \cdot \text{input conditons associated with "Set" transitions}) +$ $\sum(PS \cdot \text{input conditons associated with "Clear" transitions})$
J excitation equation:	$\sum(PS \cdot \text{input conditons associated with "Set" transitions})$
K excitation equation:	$\sum(PS \cdot \text{input conditons associated with "Clear" transitions})$

Table 32.1: Closed form representations of excitation equations for the D, T, J, and K inputs.

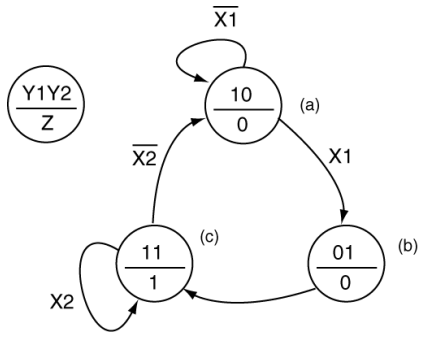
32.3.4 The Clark Method for the New FSM Techniques

Now let's apply everything we've learned up until now in an actual example. The task is to use a state diagram to generate the excitation equations that could implement a FSM using D, T, or JK flip-flops. First thing to remember here is that this example has nothing to do with any other examples we've worked with up to this point. The previous example only served as a motivation to the approach we'll be using in this example. The second thing to note is this structure of the approach used to solve the given example problem was developed by a Carissa Clark, a former Cal Poly student. The cool thing

about this approach is that you'll be able to do these problems and make a lot fewer mistakes in the process. This approach is aptly named the Clark Method.

Example 32-1

Using the state diagram shown in below, generate the excitation equations that can be used to implement the corresponding FSM in hardware using 1) D flip-flops, 2) T flip-flops, and 3) JK flip-flops. Also, include equations representing the external outputs represented by the state diagram.



Solution: The solution is comprised of the following series of steps. Once you solve a few of these problems, you'll probably not follow these steps again as the approach begins to make intuitive sense.

Step 0) Stare at the problem for a while. From this step, you should be able to glean the following information from the state diagram:

- There are three states in the state diagram. This means that you'll need at least two flip-flops to implement the FSM (which is somewhat implied by the legend provided in the diagram).
- There is one external input to the FSM: X1.
- The FSM contains one external output: Z. Because the Z output resides in the state bubbles, it must be a Moore-type output.
- The parenthetical letters serve as a documentation aid for this technique.

Step 1) Develop *State Variable Transition Table (SVTT)*: Table 32.2 shows a typical SVTT. The SVTT should contain one row for each of the state transitions in the state diagram. In other words, each arrow in the state diagram is indicates a state transition; your table should contain one row for each of the arrows in the state diagram. For this step, you list the state transitions in the "transitions" column of your SVTT. Note once again that labels (a, b, and c) were added to enhance your enjoyment of this method. If the state diagram did not provide these labels, you should provide them yourself.

Transitions	Conditions	Y1	Y2
a → a			
a → b			
b → c			
c → c			
c → a			

Table 32.2: The *State Variable Transition Table (SVTT)* with the listed transitions.

Step 2) List the present state (PS) and the associated conditions that cause the transitions. For each row of the SVTT, the present state is always the state from which the arrow emanates from as listed in "transitions" column of the SVTT (or the original state diagram for that matter since they are necessarily the same). As the equations in Table 32.1 indicate, the complete term associated with these rows will be the present state conditions (PS) ANDed with the conditions of the external inputs that allow the particular transition to occur.

Note that transitions from both the (a) state and the (b) state are conditional (they are based on the state of the X1 variable at the time of transition) while the transition from the (b) state is unconditional. The unconditional transition implies that the FSM always transitions out of the (b) state on the next system clock edge. Table 32.3 shows the results of this step; the verbage below details the approach on a single row.

As an example, the product term associated with the "a→a" transition has both a PS and a "conditional" component. The PS component is Y1Y2 = "10" which is listed as $Y1 \cdot \overline{Y2}$. The conditional portion of the product term is essential the external variable controlling the "a→a" transition. From the state diagram in the problem description, you can see that the "a→a" transition occurs when the X1 input is not asserted, or $\overline{X1}$. The complete product term, as listed in Table 32.3 is thus $Y1 \cdot \overline{Y2} \cdot \overline{X1}$.

Transitions	Conditions	Y1	Y2
a → a	$Y1 \cdot \overline{Y2} \cdot \overline{X1}$		
a → b	$Y1 \cdot \overline{Y2} \cdot X1$		
b → c	$\overline{Y1} \cdot Y2$		
c → c	$Y1 \cdot Y2 \cdot X2$		
c → a	$Y1 \cdot Y2 \cdot \overline{X2}$		

Table 32.3: The SVTT: listing the conditions.

Step 3) List the pertinent transition characteristics associated with each state variable for each of the listed transitions. For this step, the important transition characteristics are the "Set" the "Hold-1" and the "Clear" characteristics (and therefore this approach can ignore all other transitions). To perform this step, you must look back at the original state diagram. In particular, you must characterize every transition of the Y1 and Y2 state variables as a "Set", a "Hold-1", a "Clear", or "none of the above". Table 32.4 shows the results of this step (the "none of the above" category uses a '-').

As an example, consider the "a→b" transition. Examining the state variables listed in the problem description shows that the Y1 variable clears (1→0) when transitioning from state (a) to state (b); a

"clear" is subsequently entered into the Y1 column of the row representing the "a→b" transition. The Y2 state variable "sets" (0→1) during this same transition, so we enter a "set" into the Y2 column of the row associated with the "a→b" transition.

Transitions	Conditions	Y1	Y2
a → a	$Y1 \cdot \overline{Y2} \cdot \overline{X1}$	hold-1	-
a → b	$Y1 \cdot \overline{Y2} \cdot X1$	clear	set
b → c	$\overline{Y1} \cdot Y2$	set	hold-1
c → c	$Y1 \cdot Y2 \cdot X2$	hold-1	hold-1
c → a	$Y1 \cdot Y2 \cdot \overline{X2}$	hold-1	clear

Table 32.4: The SVTT: characterizing the transitions in terms of Y1 and Y2.

Step 4) Use the information you've listed in Table 32.4 and the equations listed in Table 32.1 to generate the excitation equations for the D, T, and JK flip-flops. To perform this step, scan down the columns representing the state variables and collect terms associated with the desired flip-flops. For the D flip-flop, you collect the product terms (listed in the "conditions" column) associated with the "set" and "hold-1" characteristics listed in the Y1 and Y2 columns. Keep in mind that since this design requires two flip-flops, you will need two excitation equations: one for both Y1 and Y2. The T flip-flop equations will also require two equations while the JK flip-flop requires four excitation equations. Figure 32.6 shows the results of this step.

For example, we write the D1 excitation equation (the D input to the flip-flop representing the Y1 state variable) by logically summing the product terms corresponding the "sets" and "hold-1's" in the Y1 column. Since there are four "sets" and "hold-1's" there are four product terms in the excitation equation as listed in Figure 32.6.

$D1 = Y1 \cdot \overline{Y2} \cdot \overline{X1} + \overline{Y1} \cdot Y2 + Y1 \cdot Y2 \cdot X2 + Y1 \cdot Y2 \cdot \overline{X2}$ (**set** and **hold-1** transitions for Y1)

$D1 = Y1 \cdot \overline{Y2} \cdot X1 + \overline{Y1} \cdot Y2 + Y1 \cdot Y2 \cdot X2$ (**set** and **hold-1** transitions for Y2)

$T1 = Y1 \cdot \overline{Y2} \cdot X1 + \overline{Y1} \cdot Y2$ (**set** and **clear** transitions for Y1)

$T2 = Y1 \cdot \overline{Y2} \cdot X1 + Y1 \cdot Y2 \cdot \overline{X2}$ (**set** and **clear** transitions for Y2)

$J1 = \overline{Y1} \cdot Y2 = Y2$ (**set** transitions for Y1)

$K1 = Y1 \cdot \overline{Y2} \cdot X1 = \overline{Y2} \cdot X1$ (**clear** transitions for Y1)

$J2 = Y1 \cdot \overline{Y2} \cdot X1 = Y1 \cdot X1$ (**set** transitions for Y2)

$K2 = Y1 \cdot Y2 \cdot \overline{X2} = Y1 \cdot X1$ (**clear** transitions for Y2)

Figure 32.6: The final excitation equations for D, T, and JK flip-flops.

Step 5) Write an equation for each of the output variables. For this example, this step is not overly complicated; but then again, it's not that complicated for any problem. The approach is once again similar to the approach taken with the excitation equation specification. Recall that in general, a K-map wrote terms that included all the 1's in the groupings. Since we still don't want to deal with K-maps in this new technique, we instead write the output equation by inspection of the original state diagram. In other words, you need to list where the output in question is a '1'. You find out where the output is a '1' by examining the state diagram only (there is once again no need to do the truth table thing).

In this example, this step is done by noting that only time the Z output is only a '1' in the Y1Y2 = "11" state. The equation for the Z output is therefore: $Z = Y1 \cdot Y2$. In this example, the output is a Moore-type output, which makes the equation generation slightly more straightforward than if the state diagram contained Mealy-type outputs. To drive this point home, note that the output equation for Z contains no external input variables (it has no X1 terms); the output equations for Mealy-type output necessarily contain X1 terms, as you'll see in later examples.

Make sure you completely understand this step. Keep in mind that there are no tricks or shortcuts available to reduce the equations for the outputs as there were for the JK flip-flop excitation equations. Historically speaking, generating the equations for the output variables generally causes students problems for two reasons:

1) Students become so excited about generating the excitation equations for the associated flip-flops, they forget to generate equations for the output variables. Please don't forget to generate the equations for the output variables. You'll never see a problem that asks for the excitation equations only: we'll always ask you to include output equations.

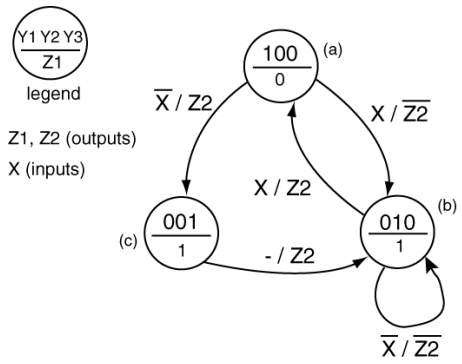
2) Though the reason is not clear, some people's understanding seems to falter when they need to write equations for the output variables. Although they'll list the excitation equations 100% correctly, they'll completely drop the ball on the output equations; please don't be one of these people. The techniques are slightly different; to understand the problem completely, you need to be able to do both. On top of all of this, writing equations for the output is more straightforward than generating the excitation equations.

Step 6) Draw the final circuit (let's skip this step; we've done this many times before). This step is much less exciting than all the other steps.

The final comment: this was not an overly complicated example. The "new" techniques become a little more complicated when there are both Mealy and Moore outputs that you need to represent in the output equations. In a later example, we'll generate outputs for both Mealy and Moore outputs.

Example 32-2

Use the "new" FSM techniques to provide the excitation equations that could implement the state diagram shown below using D, T, and JK flip-flops. Include equations describing the FSM's external outputs.



Solution: There are actually two new things to show you in this example. First, there is yet another special type of state reduction possible when the state variables are one-hot encoded. Second, this example contains both a Mealy and a Moore output. Recall that the previous example only included a Moore-type output. The relationship between the Mealy variables specified in the state diagram and their associated states is often a source of confusion: enlightenment surely follows.

Since we have already implemented a similar FSM in excruciating detail, we'll include less verbage in this example. From here, use the standard approach to writing the excitation equations we used for the "new" FSM methods detailed in the previous example.

Step 0) Stare at the state diagram. This FSM contains three states with 1-hot encoded state variables. Y1, Y2, and Y3, chosen arbitrarily, represent the associated state variables. The FSM uses the variable

X to represent the external input. The FSM contains two external outputs represented by: Z1 (a Moore-type output) and Z2 (a Mealy-type output).

Step 1) List all the conditions associated with the state changes in the SVTT; Table 32.5 shows the Table 32.5 results of this step. Remember, there should be one row in the SVTT for each of the arrows shown in the state diagram of the problem description.

Transitions	Conditions	Y1	Y2	Y3
a → b				
a → c				
b → b				
b → a				
c → b				

Table 32.5: The *State Variable Transition Table (SVTT)* with the listed transitions.

Step 2) List the conditions (both input and present state) that cause the associated state transitions. Figure 32.6 shows this result.

Transitions	Conditions	Y1	Y2	Y3
a → b	$Y1 \cdot \overline{Y2} \cdot \overline{Y3} \cdot X$			
a → c	$Y1 \cdot \overline{Y2} \cdot \overline{Y3} \cdot \overline{X}$			
b → b	$\overline{Y1} \cdot Y2 \cdot \overline{Y3} \cdot \overline{X}$			
b → a	$\overline{Y1} \cdot Y2 \cdot \overline{Y3} \cdot X$			
c → b	$\overline{Y1} \cdot \overline{Y2} \cdot Y3$			

Table 32.6: The SVTT: listing the conditions.

Step 2*) Simplify the expressions for the present states. This is an extra step because you are 1-hot encoding the state variables. This is a special type of simplification can only be applied when you use one-hot encoding. The reduction is possible for the following reason: if one of the one-hot encoded state variables is a '1', then by the definition of the one-hot states, all of the other state variables must necessarily be 0's. Using this valuable information, the condition listed in the first row, $Y1 \cdot \overline{Y2} \cdot \overline{Y3} \cdot X$, reduces to $Y1 \cdot X$.

Another way to look at this is that if Y1 is a '1', then $Y1$ and $Y2$ must be both '0'. This means that $\overline{Y1}$ and $\overline{Y2}$ must both be 1's, so you can reduce the original equation to $Y1 \cdot \overline{Y2} \cdot \overline{Y3} \cdot X = Y1 \cdot 1 \cdot 1 \cdot X = Y1 \cdot X$ as is listed in the first row of Table 32.7. Once again, this type of reduction works only with FSMs where the state variables are one-hot encoded. Table 32.7 shows the final result of this step.

Transitions	Conditions	Y1	Y2	Y3
a → b	$Y1 \cdot X$			
a → c	$Y1 \cdot \overline{X}$			
b → b	$Y2 \cdot \overline{X}$			
b → a	$Y2 \cdot X$			
c → b	$Y3$			

Table 32.7: The SVTT: listing the conditions in the 1-hot reduction form.

Step 3) List the pertinent transition characteristics associated with each state variable for each of the listed transitions. Remember, the transitions we're interested in are Sets, Clears, and Hold-1's. Table 32.8 shows the final results of this step.

Transitions	Conditions	Y1	Y2	Y3
a → b	$Y1 \cdot X$	clear	set	-
a → c	$Y1 \cdot \overline{X}$	clear	-	set
b → b	$Y2 \cdot \overline{X}$	-	hold-1	-
b → a	$Y2 \cdot X$	set	clear	-
c → b	$Y3$	-	set	clear

Table 32.8: The SVTT: characterizing the transitions in terms of Y1, Y2, and Y3.

Step 4) Use the information listed in Table 32.8 and the equations listed in Figure 32.7 to generate the excitation equations for the D, T, and JK flip-flops. Table 32.9 shows these results.

D Flip-flops	T Flip-flops	JK Flip-flops
$D1 = Y2 \cdot X$	$T1 = Y1 \cdot X + Y1 \cdot \overline{X} + Y2 \cdot X$	$J1 = Y2 \cdot X$
$D2 = Y1 \cdot X + Y2 \cdot \overline{X} + Y3$	$T2 = Y1 \cdot X + Y2 \cdot X + Y3$	$K1 = Y1 \cdot X + Y1 \cdot \overline{X} = X + \overline{X} = 1$
$D3 = Y1 \cdot \overline{X}$	$T3 = Y1 \cdot \overline{X} + Y3$	$J2 = Y1 \cdot X + Y3$
		$K2 = Y2 \cdot X = X$
		$J3 = Y1 \cdot \overline{X}$
		$K3 = Y3 = 1$

Table 32.9: The final excitation equations for the D, T, and JK flip-flops.

D excitation equation: $\sum(\text{PS}\cdot\text{input conditons associated with "Set" transitions}) + \sum(\text{PS}\cdot\text{input conditons associated with "Hold-1" transitions})$

T excitation equation: $\sum(\text{PS}\cdot\text{input conditons associated with "Set" transitions}) + \sum(\text{PS}\cdot\text{input conditons associated with "Clear" transitions})$

J excitation equation: $\sum(\text{PS}\cdot\text{input conditons associated with "Set" transitions})$

K excitation equation: $\sum(\text{PS}\cdot\text{input conditons associated with "Clear" transitions})$

Figure 32.7: Closed form representation of excitation equations for the D, T, J, and K inputs.

Step 5) Write an equation for each of the output variables. This is where things get a little strange. This step often causes trouble for some people so don't let this happen to you.

The first thing to remember in this step is that you're still looking for situations where the output is a '1' (remember back to the K-map motivation of this approach). Note that the original state diagram contained two types of outputs: one Mealy and one Moore. Writing equations for the Moore outputs is slightly more obvious (plus we've already did it once in the previous example) so let's do that first. There are two states where Z1, the Moore output, is a '1'. The resulting equation that represents these two conditions (and hence, the Z1 output) is listed in Equation 32-4. Note that the input variable X is not included in this final equation because Z1 is a Moore output. In other words, if the equation for your Moore output were to contain an external input variable, then it by definition can't be a Moore output. Note that Equation 32-4 lists the output equation with the special 1-hot type of reduction.

$$Z1 = \overline{Y1}\cdot\overline{Y2}\cdot Y3 + \overline{Y1}\cdot Y2\cdot\overline{Y3} = Y3 + Y2$$

Equation 32-4: The equation describing the Z1 output.

The Mealy output is more interesting and it delves into the functional purpose of FSMs. What we need to do is to write an equation that describes the Z2 variable. Once again, we want this equation to describe where in the state diagram that the Z2 variable is in the '1' state. Before we start on this task, look closely at the state labeled (a) in the original problem statement. On one arrow leaving the diagram, the Z2 variable is in complemented form and on the other arrow, the Z2 variable is not in complimented form. There are major points associated with these conditions: 1) the state of the Z2 variable depends on the state of the external X input (because it's a Mealy output), and 2) the output listed with the state transitions (which is Z2 in this case) belongs to (is associated with) the state from where the arrow leaves. To sum up these two big points… the Mealy outputs listed with the state transitions are associated with the state from which the arrow is leaving.

Well… that's the big stumbling point. Now that this point stands the chance of being slightly clearer, let's write an equation for Z2. The method used to do this is to locate all the instances of uncomplimented Z2s in the state diagram (the 1's of the circuit), and write a term that describes them. These terms are summed together to form the expression shown in Equation 32-5. Note that Equation 32-5 shows the output equation with the special one-hot style of reduction applied.

$$Z2 = Y1 \cdot \overline{X} + Y3 + Y2 \cdot X$$

Equation 32-5: The boolean expression describing the Z2 output.

Chapter Summary

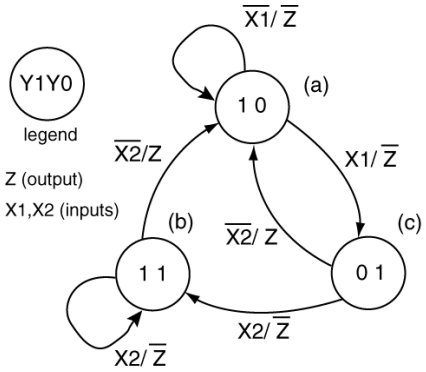
- **"New" FSM implementation techniques:** The "new" FSM techniques were introduced to overcome a basic draw back of "classical" FSM implementation techniques. "Classical" FSM techniques involved the use of K-maps, which essentially limited the number of state variables, and external inputs (these are the inputs to the next state decoder) to no more than four literals. The "new" techniques provided an approach to write the associated excitation equations directly from the state diagram without using K-maps. The only drawback of the "new" approach is that the generated excitation equations are not necessarily in reduced form. The new techniques consist of the "Set or Hold-1" (D flip-flops), the "Set or Clear" (T flip-flops), and the "Set-Clear" (Apply to present state condition for all types of flip-flops and all types of outputs (if the associated FSM is one-hot encoded only).

- **Special JK flip-flop reduction:** The excitation equations can be further reduced when using the "new" techniques with JK flip-flops. This technique cannot be applied to the output equations. The special JK reduction can be applied independently of the type of state variable encoding used for a given FSM.

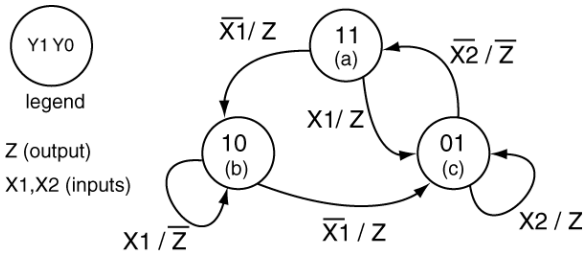
- **Special one-hot state reduction:** The excitation equations can be reduced when one-hot encoding is used. This technique is based on the fact that if we know one state variable is a '1', we all the other state variables are '0' when one-hot encoding us used. This reduction technique can be applied to independent of the type of flip-flop used and is not constrained to "new" FSM implementation techniques.

Chapter Exercises

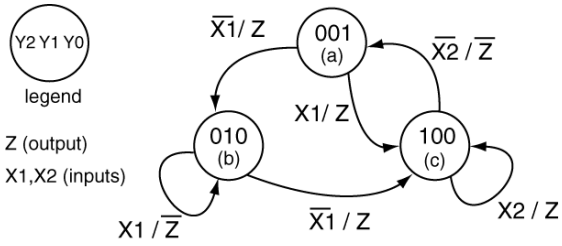
1) Using "new" FSM design techniques, write the excitation equations that would implement the following state diagram. Do the problem using D, T, and JK flip-flops in your implementation and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the circuit.



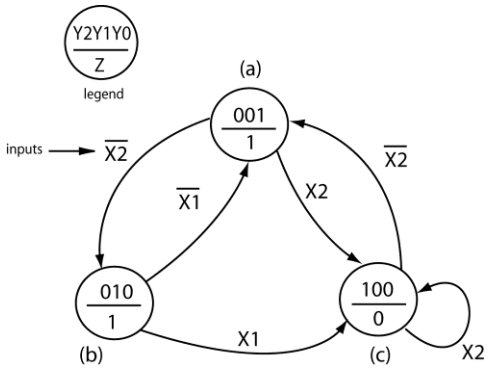
2) Using a "new" FSM design technique, write the excitation equations that would implement the following state diagram. Do the problem using D, T, and JK flip-flops and binary encoding in your implementation and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the circuit – just provide excitation equations.



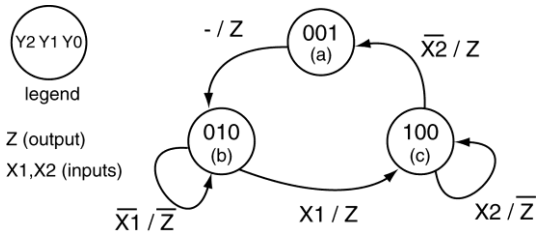
3) Using a "new" FSM design technique, write the excitation equations that would implement the following state diagram. Write the equations for using sets of D, T, and JK flip-flops. Use one-hot encoding in your implementation and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the circuit.



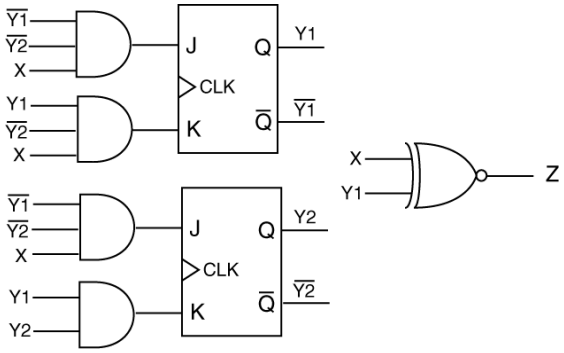
4) Use the Set or Hold 1, the Set or Clear, and the Set-Clear technique to generate the excitation input equations that will implement the following state diagram. Write an equation for the output variable also. Minimize the resulting excitation equations where possible.



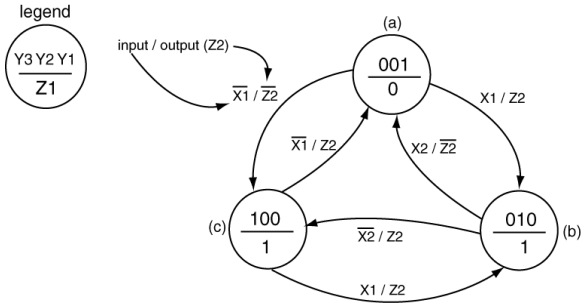
5) Using a "new" FSM design technique, write the excitation equations that would implement the following state diagram. Do the problem using D, T, and JK flip-flops and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the circuit – just provide excitation equations.



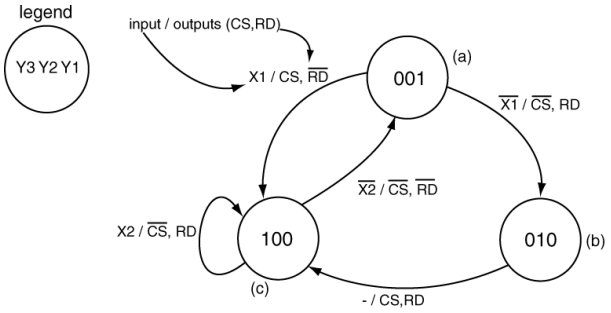
6) The following circuit implements a finite state machine that was specified using the "new" FSM design technique. Re-implement the circuit using two T flip-flops instead of the listed JK flip-flops. Be sure to draw the final circuit.



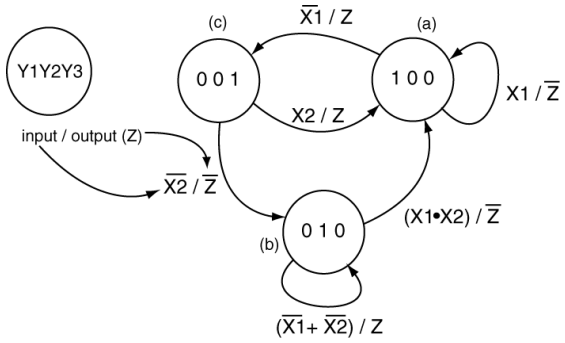
7) Using a "new" FSM design technique, write the excitation equations that could be used to implement the following state diagram. Use D flip-flops in your implementation and reduce the resulting equations using the special one-hot techniques (when appropriate). Write an equation for each of the output variables also. Don't draw the final circuit.



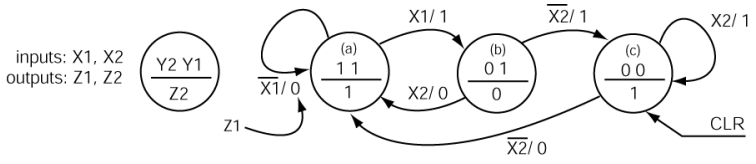
8) Using a "new" FSM design technique, write the excitation equations that could be used to implement the following state diagram. Use JK flip-flops in your implementation and reduce the resulting equations using the special FSM techniques (when appropriate). Write an equation for each of the output variables also. Don't use classical FSM design techniques. Don't draw the final circuit.



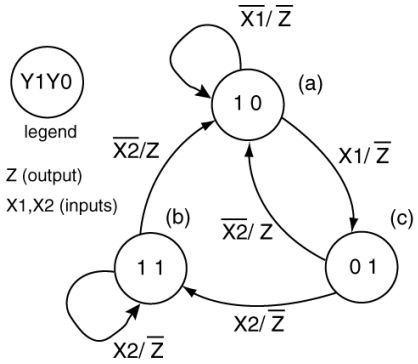
9) Using a "new" FSM design technique, write the excitation equations that could be used to implement the following state diagram. Use T flip-flops in your implementation and reduce the resulting equations using the special techniques (when appropriate). Write an equation for the output variable also. Don't draw the final circuit.



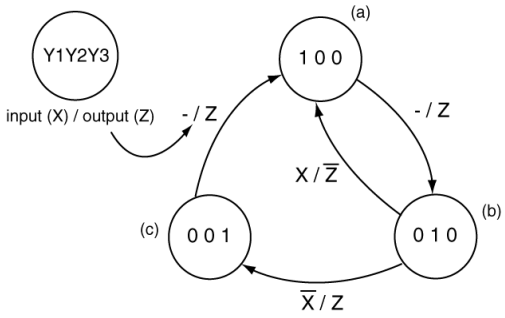
10) Using a "new" FSM design technique, write the excitation equations that could be used to implement the following state diagram. Use T flip-flops in your implementation and reduce the resulting equations using the special techniques we discussed in this chapter (when appropriate). Write an equation for the output variable also. Don't draw the final circuit.



11) Using a "new" FSM design technique, write the excitation equations that would implement the following state diagram. Use JK flip-flops in your implementation and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the circuit.



12) Using a "new" FSM design technique, write the excitation equations that would implement the following state diagram. Use JK flip-flops in your implementation and reduce the resulting equations when possible. Write an equation for the output variable also. Don't draw the final circuit.



33 Chapter Thirty-Three

(Bryan Mealy 2012 ©)

33.1 Chapter Overview

I have no trouble stating that most commonly used circuit in digital design is the "register". We've already used the term quite often in this text, particularly regarding finite state machines (FSMs). Recall that a main component of FSM was the storage associated with the state variables. Although I did my best not to use the word, there were several instances when I used the term "state registers" to refer to the circuit elements storing the state variables.

The concept of registers is not complicated and you've been dealing with the basic register concepts for many chapters at this point. This chapter describes the notion of registers and their many various flavors and incarnations. Most of the description appearing in this chapter is at a higher-level as the low-level details are somewhat cumbersome and not overly useful. All forms of registers are massively useful in digital design.

Main Chapter Topics

- **SIMPLE REGISTERS AND REGISTERS "WITH FEATURES":** This chapter defines and describes basic including registers with extended features that make them more useful in digital circuits.

Why This Chapter is Important

This chapter is important because registers and their variations are extremely useful and thus often found in just about all meaningful digital designs.

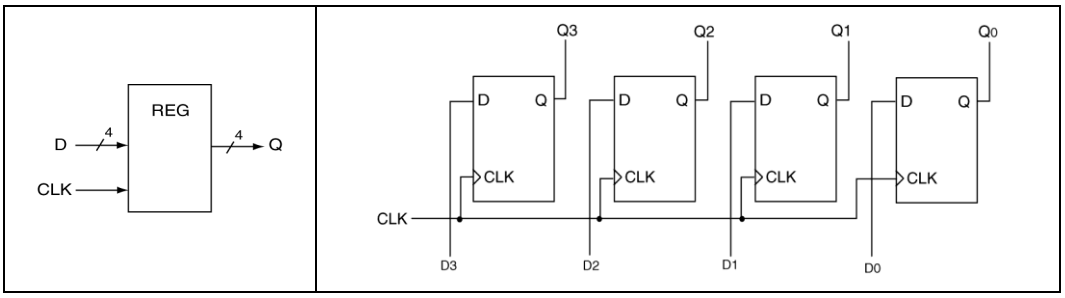
33.2 Registers: The Most Common Digital Circuit Ever?

Stated as simply as possible, a register is nothing more than a multi-bit flip-flop. Flip-flops are single bit storage elements while registers multi-bit storage elements modeled as a given number of flip-flops connected in parallel. The good news is that only D flip-flops are used to model registers, which simplifies their understanding and representation. Moreover, VHDL models of registers are similar to flip-flop models and only differ in the width of the "data" inputs and outputs. In addition, being that register such as these have such simple descriptions, we'll refer to this flavor of registers as "simple registers". For now and evermore, when we say, "register", we typically mean "simple register"; this works well as the more specialized registers have their own names (which we'll present in a later chapter).

Jumping right into it, Figure 33.1 shows four D flip-flops assembled such that they act as a register. In particular, Figure 33.1(a) shows the block diagram for a 4-bit register and Figure 33.1(b) shows the underlying circuit. Here are a few things to note about Figure 33.1:

- The block diagram in Figure 33.1(a) shows a clock signal but also assumes other characteristics. Since the register is modeled with D flip-flops, there must be an active clock edge that Figure 33.1(a) does not show. Unless otherwise stated, registers are generally active on the rising edge of the clock. Figure 33.1(b) shows the rising clock edge, though you would generally not see a circuit such as this when you're working with registers.

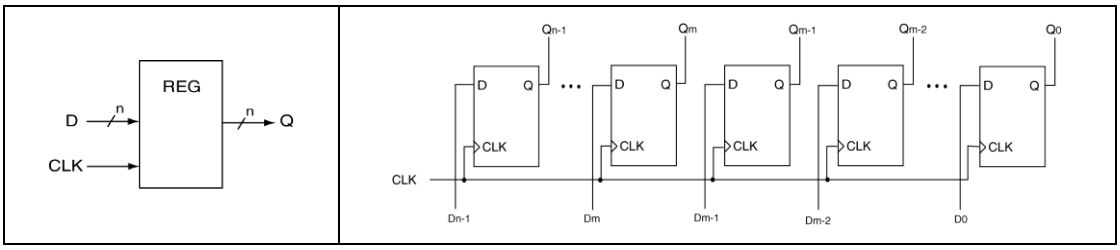
- Figure 33.1(b) shows that each flip-flop in the register shares the same clock. The result is that all the flip-flops latch their data simultaneously. We'll demonstrate this later in a timing diagram.



(a) **(b)**

Figure 33.1: A block diagram for a 4-bit register (a), and the lower-level implementation details of a 4-bit register (b).

Figure 33.2(a) shows the block diagram for a generic n-bit register; Figure 33.2(b) shows the underlying details. The main point behind Figure 33.2 is to show the notion that registers are simple to model and it takes about zero effort to model registers of any width. The only thing about registers that may be somewhat tricky is the notion that an n-bit register is generally modeled using n signals: the least significant bit (LSB) has an index of "0" while the most significant bit (MSB) has an index of "n-1". Get used to it.



(a) **(b)**

Figure 33.2: A general case of an n-bit register; the block diagram (a), and a model for the underlying circuit (b).

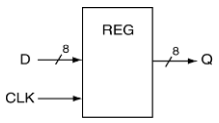
Probably the happiest way to describe the operation of a register is with the associated VHDL model. Figure 33.3 shows a VHDL model for an 8-bit register. From this model, you can see that the register is in fact active on the rising clock edge. There is not a lot to say about the VHDL model shown in Figure 33.3 as this model should appear familiar, as it resembles a simple model for a D flip-flop. In truth, D flip-flops are naturally easy to model in VHDL mainly because the main goal in design the language was to be able to model popular digital circuits such as registers without expending too much effort.

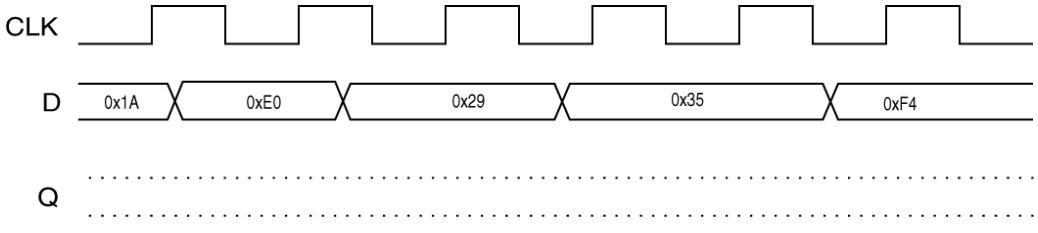
```
-----------------------------------------------------------
-- VHDL model of 8-bit register
-----------------------------------------------------------
entity reg_8b is
    Port (   D : in std_logic_vector(7 downto 0);
           CLK : in std_logic;
             Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin

   process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process;

end my_reg_8b;
```

Figure 33.3: The VHDL model for a simple 8-bit register.

Example 33-1

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered; ignore all propagation delay issues.





Solution: From the problem description, we know the block diagram represents an 8-bit register that is active on the rising clock edge. This means that we need to examine only the portions of the timing diagram aligned to the rising edge of the clock. At these times, the data on the input of the register

transfers to the output of the register. Figure 33.4 shows the final result for this example. The final solution is straightforward but does have a few items worth noting.

- The solution added dotted vertical lines on the rising clock edges. This is something you should always consider doing when working with timing diagrams. Timing diagrams can quickly become quite complexicated so drawing dotted lines such as these as a first step in solving these problems may save your arse.

- The problem did not provide an initial value for the contents of register. Because of this, the first time-slot on the "Q' line contains question marks. In other instances, the problem may either state an initial value or provide some type of signal that places the register into a known state. We'll see such a "reset" signal in a later example problem.

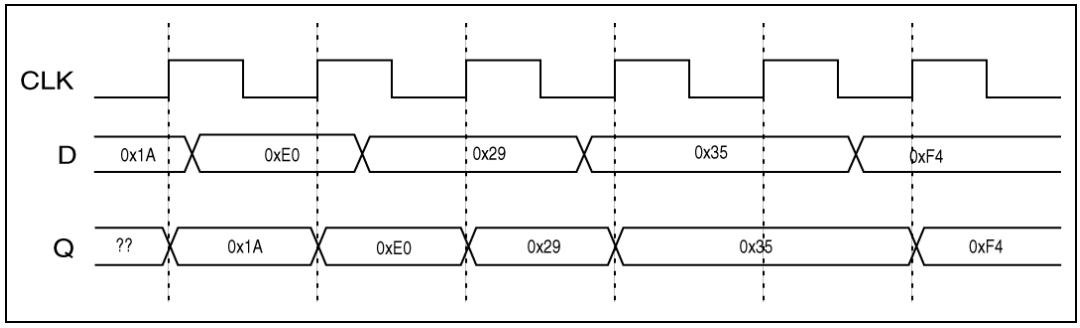
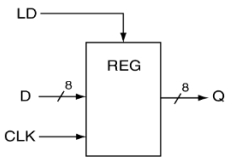


Figure 33.4: The solution for Example 4-7.

In real digital circuits, you rarely see registers as simple as the register described in Figure 33.3. These registers don't have enough "control" to make them ultimately useful. The issue is that at every active clock edge, the register latches the input data. Real registers generally contain some sort of control signals that direct when the register latches data.

Without too much explanation, Figure 33.5 shows a register containing a signal that controls when the register latches the input data. Control signals for such registers are typically associated with the word "load"; registers typically "load" the input data into the register. As a result, we use "LD" as a signal name for the control signal in the register shown in Figure 33.5. This signal allows each of the single-bit storage elements in the register to latch their associated bits. In particular, Figure 33.5(a) shows a block diagram for the register with a control signal while Figure 33.5(b) shows the associated VHDL model. The following example shows the operation of this register.



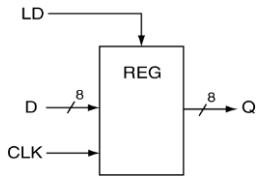

```
-------------------------------------------------------------
-- VHDL model of 8-bit register with parallel load
-------------------------------------------------------------
entity reg_8b is
    Port (      D : in std_logic_vector(7 downto 0);
           CLK,LD : in std_logic;
                Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin

   process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         if (LD = '1') then
            Q <= D;
         end if;
      end if;
   end process;

end my_reg_8b;
```

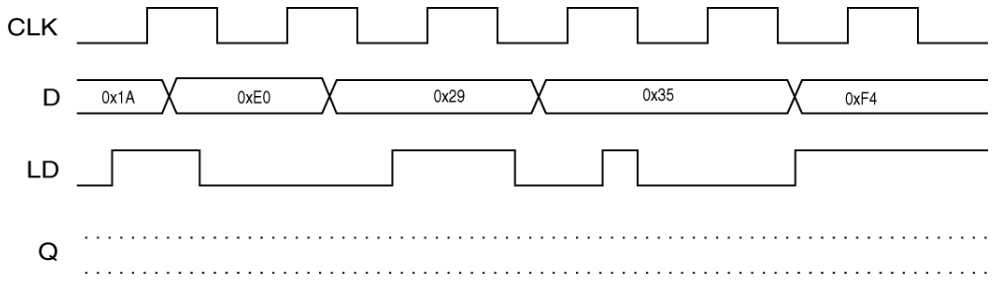
(a) **(b)**

Figure 33.5: An 8-bit register with a parallel load signal (a), and a VHDL model for the underlying circuit (b).

Example 33-2

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered; ignore all propagation delay issues.





Solution: This problem is similar to the previous problem but now we need to keep track of the "LD" signal. In the previous problem, we only needed to examine the times when the rising edges occurred. In this problem, we need to examine the times where the both the rising edge occurs and where the LD signal is asserted. Note that because the LD signal on the register does not have a bubble, the load

signal is active high. Figure 33.6 shows the solution for this example; some interesting things to note surely follow as well.

- The rising edges are explicitly marked with vertical dotted lines in order to avoid confusing ourselves.

- The problem statement does not provide an initial value for the register so we must mark it as unknown. The question marks work well for this dilemma.

- The LD signal is "level sensitive" which essentially means that it is not edge sensitive. This means that we are only concerned with the register loading when the LD signal is asserted and not only on the rising edge associated with the signal. Note that in Figure 33.6, at the time marked with the circled "1", the LD signal asserts and then de-asserts shortly thereafter. This small pulse has no effect on the register because there was not rising clock edge present when the LD signal was asserted.

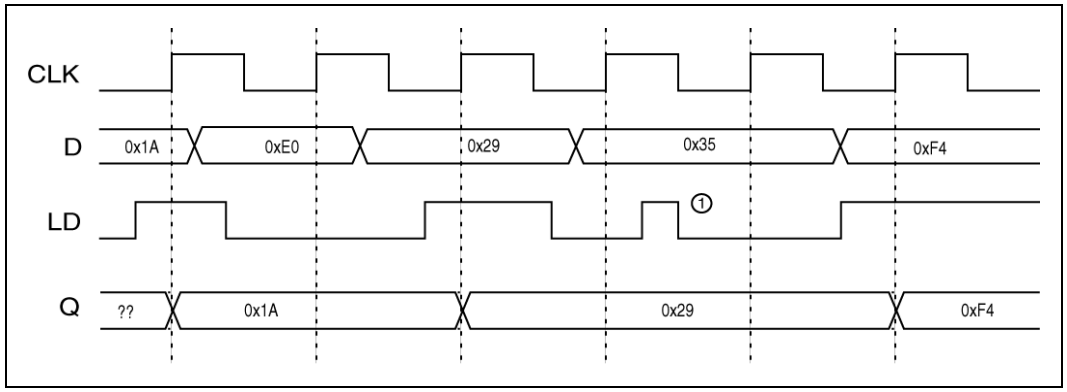
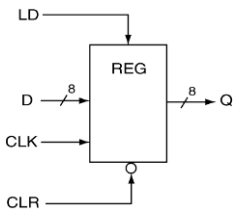


Figure 33.6: The solution for Example 33-2.

Registers can have other control options also. The notion here is that if you need to use a register in your circuit, you choose the one with the smallest feature set but still allows you to get your job done. Extra features in circuits require extra hardware; extra hardware takes up space and requires extra power to operate. If you're designing your own registers, such as in a VHDL application, you have the ability to design just about any feature into the device as required by your circuit. The next example we'll look at has one more added feature; after that, we'll stop talking about registers.

Figure 33.7(a) shows a register that has both a load and a clear input. Because this is a register, everyone generally assumes that the load signal is synchronous. The clear signal is usually asynchronous, but not always. The moral of this circuit is that you should make sure you know everything there is to know about the circuit; making assumptions is usually problematic. What saves us on Figure 33.7(a) is that Figure 33.7(b) provides the VHDL model for the diagram. If you read the associated VHDL model, you can see that the LD signal is synchronous while the CLR signal is asynchronous. Yet another example problem shows how this circuit operates.



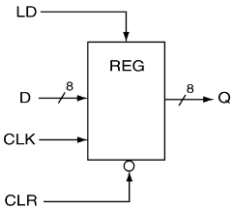

```
-------------------------------------------------------------
-- VHDL model of 8-bit register with load and clear
-------------------------------------------------------------
entity reg_8b is
    Port (          D : in std_logic_vector(7 downto 0);
           CLK,LD,CLR : in std_logic;
                    Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin
   process (D,CLK)
   begin
      if (CLR = '0') then   -- asynchronous clear signal
         Q <= (others => '0');
     elsif (rising_edge(CLK)) (then
         if (LD = '1') then
            Q <= D;
         end if;
      end if;
   end process;

end my_reg_8b;
```

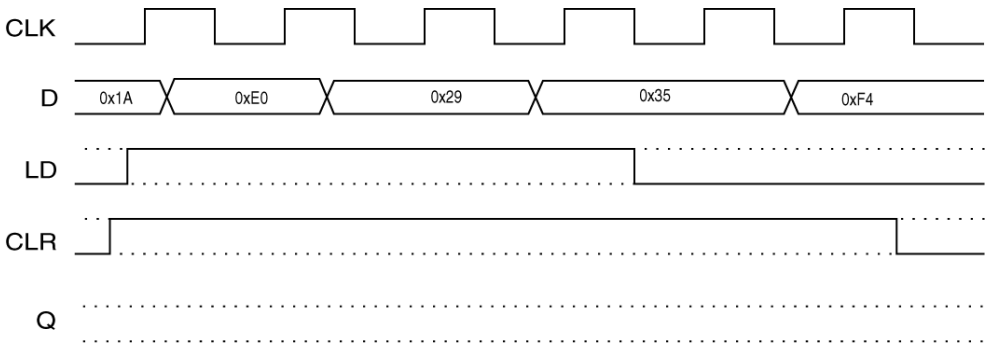
(a) **(b)**

Figure 33.7: An 8-bit register with a synchronous parallel load input and an asynchronous clear input (a), and a VHDL model for the underlying circuit (b).

Example 33-3

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues. The LD input is a synchronous parallel load input while the CLR signal is an asynchronous active low signal that clears the register when asserted.





Solution: Although this solution is rather straightforward, it provides a few new tidbit of information regarding the operation of registers.

- Unlike the previous examples, the asserted CLR signal at the beginning of the timing diagram makes the value stored in register a known value. Because of the asserted CLR, the register clears all the internal storage elements as indicated by the timing diagram.

- Though you can't tell from the first instance of the asserted clear signal, the second instance shows that the CLR signal is actually asynchronous. We know this because the clearing of the output register occurs shortly after the CLR signals asserts near the end of the timing diagram.

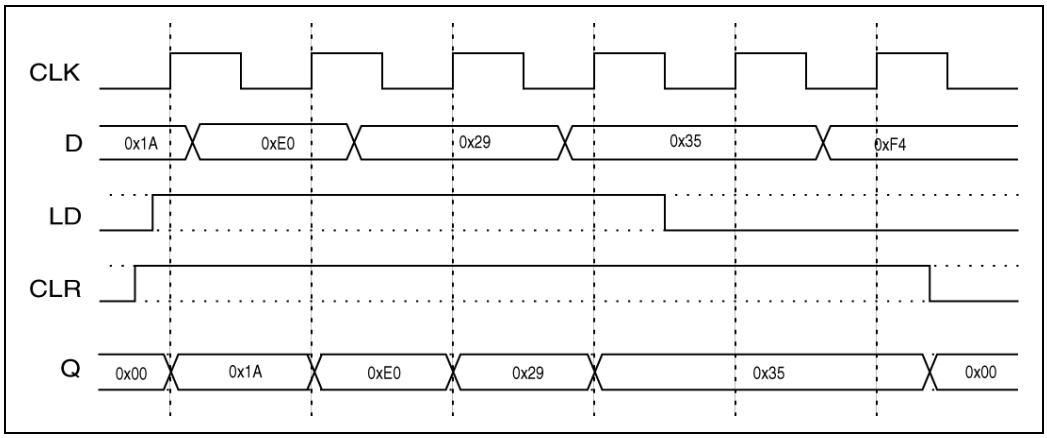


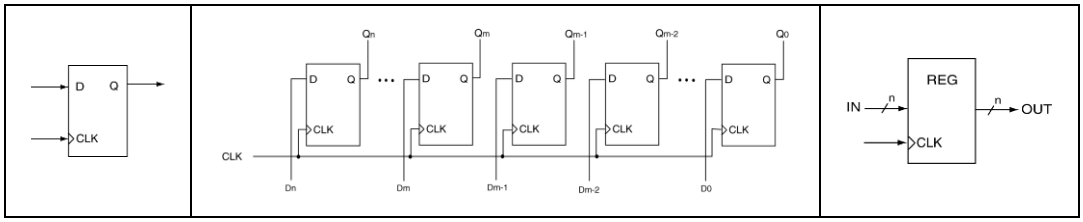
Figure 33.8: A block diagram for a FSM (Moore-machine).

A few final words on registers as a reminder… Registers come in many different flavors; this section only presented a few of them. As you'll find out in later chapters, there are a few more types of common devices out there that are essentially highly specialized registers. When someone mentions a "register", they typically are referring to the register described in this section. The other types of common registers have their own special names ("shift registers" and "counters"); the next chapter describes these registers.

33.3 Registers: The Final Comments

A register is nothing more than a set of bit storage elements that share a single clock signal. In other words, registers are a parallel configuration of signal bit storage elements; what makes them parallel is the fact that the individual storage element operations synchronize themselves to some event (usually a clock edge). D flip-flop easily model single bit storage elements; if you line up a bunch of D flip-flops together and synchronized their actions with a clock edge, you have a register.

Once you abstract all of these matters to a higher level, you'll forever more speak about *n-bit registers*. Figure 33.9 shows the progression of this abstraction. One thing to note here is that the black box diagram of a register shown in Figure 33.9 (c) includes a clock signal. The level of abstraction here sometimes continues to the point of not including the synchronizing signal (in this case, the clock) in the block diagram. In these cases, we assume the register to have a clock signal and it is interpreted accordingly. Additionally, you can safely assume that all registers are edge-triggered unless told otherwise.



(a) (b) (c)

Figure 33.9: The progression from D flip-flop to register block diagram for n-bit register.

Chapter Summary

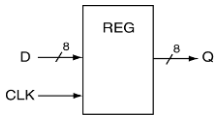
- **Registers:** A register is a sequential circuit that can be considered nothing more than a parallel combination of single-bit storage elements. These storage elements are modeled as a given number of D flip-flops that share a common clock signal and possibly other control signals typically associated with D flip-flops (such pre-set and clear signals). The register is typically used to "latch" (and thus remember) an n-bit wide set of data on the active clock edge of the device.

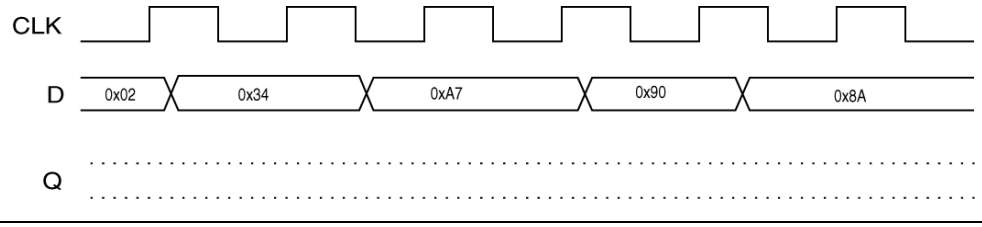
Chapter Exercises

1) Why are simple registers typically associated with D flip-flops? Would it be possible to construct a register using something such as a T or JK flip-flop? Briefly explain.

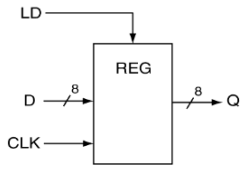
2) In what cases would there be an advantage to constructing a register (any type) using something other than a D flip-flops? Briefly explain.

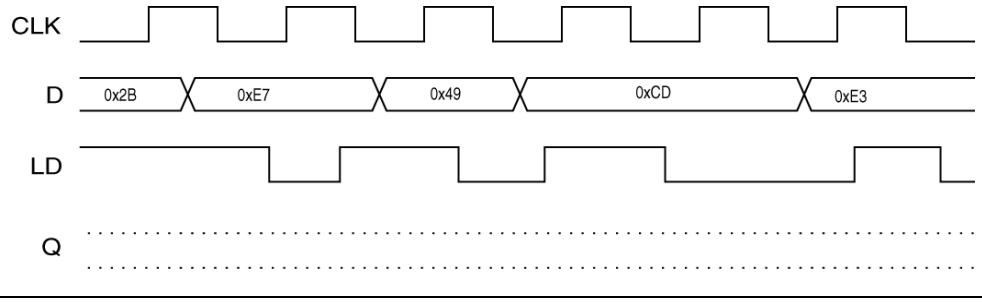
3) Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



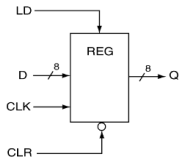


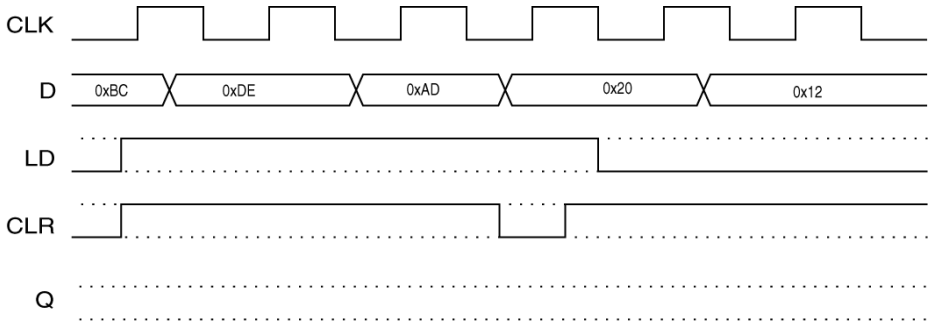
4) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



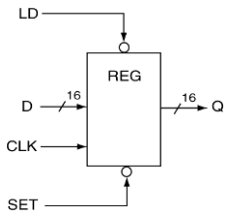


5) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. The CLR input is an asynchronous input that clears the register when asserted and has a higher precedence than the LD input. Consider the register to be rising-edge triggered and ignore all propagation delay issues.

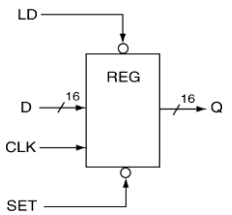




6) Provide a VHDL model that supports the black box diagram of a register on the right. The SET input is asynchronous and sets all bit storage elements in the register. The LD input loads the register on the active clock edge (falling edge triggered) and has a lower priority than the SET input.

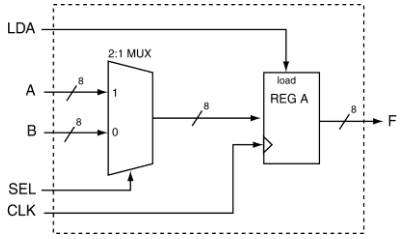


7) Using the block diagram on the right, provide a schematic diagram detailing how you would use this device to create a 32-bit register with all the same features listed on the 8-bit device.



8) Write a VHDL architecture that implements the following circuit. It is not necessary to use VHDL structural modeling for your architecture; it can be done quite nicely using a combination of dataflow and behavioral modeling.

```
entity my_ckt is
   port ( CLK, SEL, LDA : in std_logic;
          A,B : in std_logic_vector (7 downto 0);
            F : out std_logic_vector (7 downto 0));
end my_ckt;
```




```
architecture ckt1 of my_ckt is -- (you fill in the rest)
```

34 Chapter Thirty-Four

(Bryan Mealy 2012 ©)

34.1 Chapter Overview

The previous chapter dealt with the notion of registers. This chapter was relatively short and not too groundbreaking, as registers are nothing more than a bunch of D flip-flops connected in parallel. The basic simplicity of registers hides the fact that they are massively useful in all forms of meaningful digital design. If you have not seen this yet, you'll for sure see it in this chapter.

Registers come in many forms: this chapter deal with two of the most common forms. As the name implies, shifter registers are nothing more than registers with special (and useful) functionality. Counters are yet another major type of specialized registers. This chapter introduces both shift registers and counters in the context of VHDL modeling. You'll surely see that even the "registers with features" are not that much more involved than the simple register of the previous chapter.

Main Chapter Topics

- **SHIFT REGISTERS:** This chapter describes various flavors of shift registers and their basic implementations. This chapter also describes one of the main flavors of shift registers: the extremely useful barrel shifter.
- **BASIC COUNTER AND COUNTERS "WITH FEATURES":** This chapter describes various approaches to high-level VHDL behavioral modeling of counters. Previous chapters described low-level counter implementations; this chapter omits the low-level details in favor of a higher-level and thus more efficient approach.

Why This Chapter is Important

This chapter is important because shift registers and counters are extremely useful in many areas of digital design, particularly in applications requiring fast arithmetic operations. These devices are basically extended feature simple registers.

34.2 Shift Registers: the Most Useful Digital Circuit?

A shift register is another type of register that is surprisingly similar to simple registers. This section describes shift registers in the general case at a low level and then proceeds to describe other common types of shift registers at a higher level of abstraction. As you'll see, shift registers, and their various flavors, are massively useful devices because of their ability to perform a small but useful subset of mathematical operations in a quick and simple manner.

34.2.1 Basic Shift Registers

It turns out that one of the more simple circuits out there in digital-land is also one of the most useful: the friendly shift register. Shift registers, and their variants[345], are extremely useful in many digital applications primarily because the things they do can be done relatively fast. Shift registers don't really do that much[346], but they do a few things really well. Basic shift register circuitry is not complicated and thus helps the noob understand the basic functioning of useful sequential circuits.

Shift register operation is based on a simple concept. In other words, we can decompose a shift register down to its most basic component, which is referred to as a shift register cell. As you would guess, this cell is nothing more than a simple storage element and is once again modeled as a D flip-flop. Figure 34.1 shows a schematic diagram of a generic shift register. Your initial inspection of Figure 34.1 should reveal that there is not that much to the circuit. Upon further inspection, you should discern the following:

- The shift register is a sequential circuit that is similar to the simple register. First, the n-bit shift register can be modeled as a set of "n" specially connected D flip-flops. Second, all the D flip-flops in the shift register share the same clock signal which indicates all the storage elements are acting in parallel.

- The only different between simple register and shift registers is the notion that the individual storage elements in the flip-flops connect to each other in a special way. While simple registers had D flip-flops that had inputs and outputs connected to the outside world only, the shift register has inputs and outputs that interconnect between individual storage elements. Figure 34.1 shows that the output of one flip-flop becomes the input to the adjacent flip-flop in the shift register. This special type of connection is what makes it shift register.

- The number of bit storage elements in a shift register generally defines shift registers. The shift register in Figure 34.1 represents a generic model of a shift register including the magic ellipsis' placed in strategic locations. Common descriptions of shift registers include "a 4-bit shift register" or "an 8-bit shift register", etc. Thus, Figure 34.1 shows an "n-bit shift register" in its most generic form.

- Often times the letters "SR" are used to refer to a shift register (not to be confused with a SR latch).

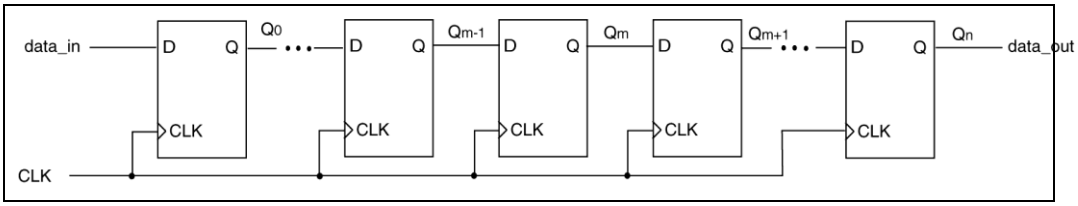
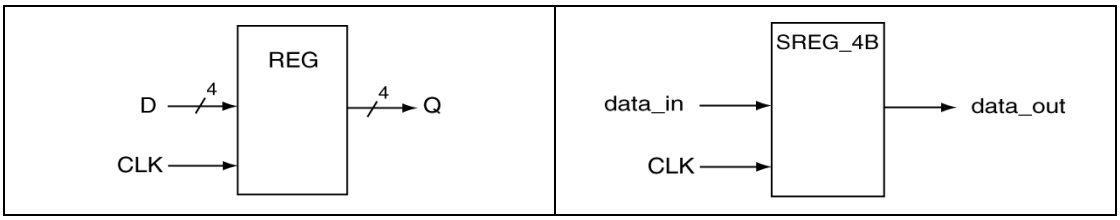


Figure 34.1: A typical n element shift register.

[345] Such as universal shift registers, barrel shifters, cyclic redundancy checking, bowling ball polishers…

[346] Back in the days before microcontrollers, you often saw shifters used as FSMs that acted as controllers.

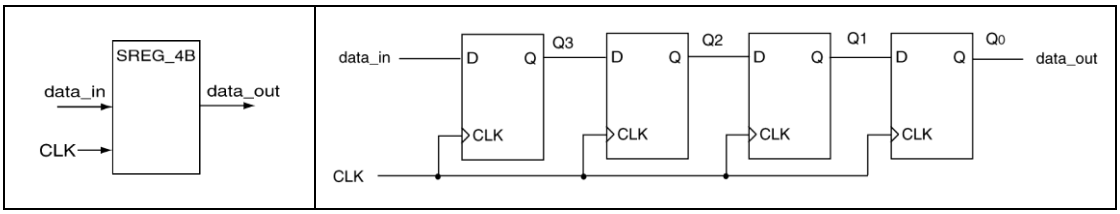
Figure 34.2 shows a comparison of block diagrams for a simple 4-bit register and a basic 4-bit shift register[347]. The important thing to notice from these diagrams is that the simple 4-bit register generally deals with "parallel" data while the basic shift register generally deals with "serial" data. What you'll find later in this chapter is that the definition of these devices starts to overlap as more features are added to the devices.



(a) **(b)**

Figure 34.2: A block diagram for a 4-bit simple register (a), and a basic 4-bit shift register (b).

The operation of a shift register is simple but can be somewhat tricky when you first encounter it. Figure 34.3(a) once again shows a schematic diagram of a 4-bit shift register while Figure 34.3(b) shows a model of the underlying circuitry. There is not really a lot to say about Figure 34.3 as the fun stuff begins when you examine a timing diagram associated with this circuit. Figure 34.4 shows an example timing diagram associated with the 4-bit shift register shown in Figure 34.3(b). Figure 34.4 contains some annotations to help with the following description of the shift register.



(a) **(b)**

Figure 34.3: A block diagram for a 4-bit simple register (a), and a model of the underlying circuitry of a 4-bit shift register (b).

[347] Keep in mind that the block diagrams show only the very basic devices for comparison purposes, which hopefully is somewhat instructive.

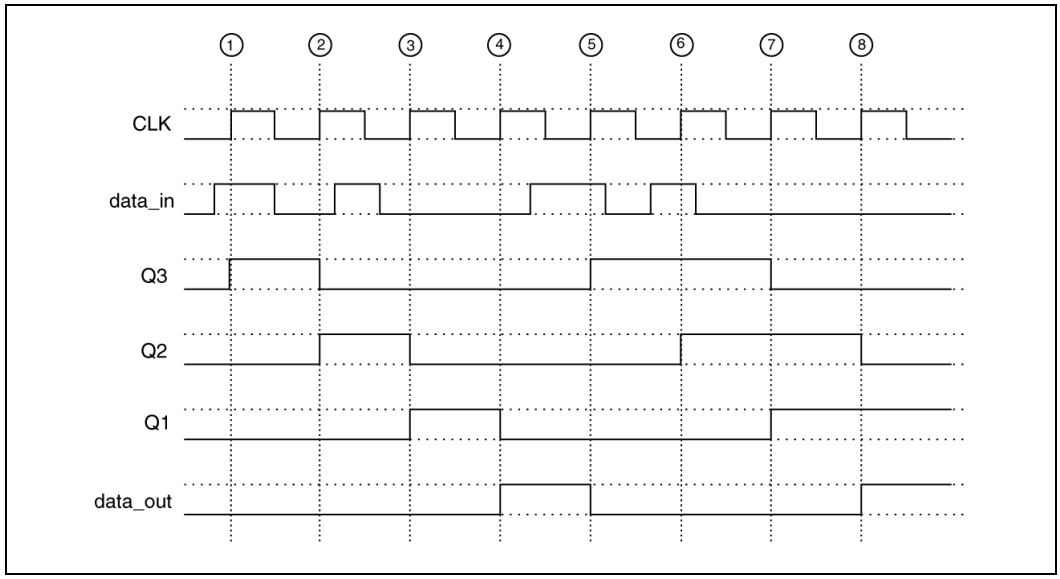


Figure 34.4: An arbitrary timing diagram associated with the shift register of Figure 34.3(b).

The items listed below describe some of the fun things to note about Figure 34.3(b) and Figure 34.4.

- This is a 4-bit shift register, meaning that the shift register circuitry contains four storage elements. Figure 34.3(a) makes a feeble attempt to indicate this is a 4-bit shift register with the label attached to the schematic symbol.

- The schematic in Figure 34.3(b) labels each of the internal shift register signals. These labels primarily serve to help describe the operation of the basic shift register as they relate to Figure 34.4. The notation of "Qx" is typical of any circuit having flip-flops as storage elements.

- The "Qx" notation used in these figures indicates the bit positions of the storage elements in the shift register. For this example, Q3 is considered the higher order bit while Q0 (or data_out) is the lowest order bit[348]. In order to simplify this explanation, the signal "data_out" and "Q0" are the same signal.

- Shift registers are considered to "shift" in either direction; that is, they shift to the left ("shift left") or shift to the right ("shift right"). The way the circuit of Figure 34.3(b) is drawn makes this a right-shifting shift register; this circuit cannot shift left based on the way its internal connections.

The notion of this circuit actually shifting something can be initially confusing. Please realize that the notion of "shifting" is primarily a term of convenience and not altogether accurate for the actual operation of the circuit (but we'll keep using it). The "thing" being shifted by the shift register of Figure 34.3(b) is the "data". Another way to look at this is that the circuit is passing in 1's and 0's from the left side of the circuit and passing them through to the right side. Figure 34.4 makes a feeble attempt at showing this shifting by way of a timing diagram. Here are some fun things to note about the timing diagram in Figure 34.4.

[348] Keep in mind that SRs are often used for mathematical operations; numbers generally have weights associated with the bit positions (unless you're a cave-person).

- Since this is a sequential circuit, the storage elements have a state associated with them. For the timing diagram of Figure 34.4, the initial state of each storage element is '0', which is completely arbitrary.

- Since the storage elements are D flip-flops, they can only change state on the active clock edge.

- On the clock edge labeled '1', all of the flip-flops transfer the value on their inputs to their outputs. In other words, on the active clock edge, the left-most flip-flop latches "data_in"; Q3 latches into the second to the left-most flip-flop, etc. In yet another way of looking at this, on each active clock edge, each of the four D flip-flops can change state or hold their current state based on the value of the D input.

- The "data_in" input is changing at various times; the only time the input has an affect is on the active clock edge.

- Overall, if you stand back a few paces, you can see the so-called shifting action of the shift register. The individual signals are shifted versions of each other; specifically, Q3 is a shifted version of "data_in", Q2 is a shifted version of Q3, etc. Another way to look at this is that the "data_out" signal is a delayed version of the "data_in" signal. In this case, Q0 is a delayed version of Q3; the delay is three clock cycles because the pulse appearing on Q0 is the same pulse that appeared on Q3 three clock cycles earlier.

Yes, shift registers are massively powerful. Keep in mind that the right-shift operation (one shift in the right direction) is the same thing as a divide-by-two operation with truncation[349]. Also, keep in mind that the circuit shown in Figure 34.3(b) is overly simple. In most shift register circuits there are other features such as most of the ones listed below. We'll discuss these in upcoming sections.

- parallel clearing or parallel setting of storage elements

- parallel loading of values to the storage elements

- Shifting left or shift right operations

- Multiple shifts on one clock edge

- Automatic bowling score feature

A few final comments regarding basic shift registers… shift register are amazingly straightforward to model in VHDL. Even shift registers that have all the bells and whistles take about zero-time to model if you understand the basics of how VHDL models sequential circuits. Figure 34.5 shows a simple no-feature VHDL model of the shift register shown in Figure 34.3(b). As you can see from Figure 34.5, the VDHL model for this simple shift register is very similar to the VHDL model for a D flip-flop (although there is one interesting trick in this model). Keep in mind that there are many different ways

[349] Truncation means the lowest order bit is lost; a similar operation is "round-up" where the value of the lowest order bit is "taken into account" and your weeds are killed at the same time.

to model shift registers in VHDL; the model shown in Figure 34.5 is not necessarily the best way[350]. The VHDL model of the basic shift register has a few interesting features worth noting.

- The single-bit storage elements associated with the basic shift register are not overly apparent. It turns out that the storage elements are associated with the "s_D" signal declaration, which is 4-bit signal. The VHDL model induces memory for this signal by the fact that the "if" statement in the process does not contain an associated "else". You've seen this before and you'll see it again.

- The VHDL model accesses only three-bits of the "s_D" signal by using the "downto" operator. This is typical VHDL vernacular and you'll for sure see this again.

- The "s_D" signal essentially "retains its value" across various executions of the process statement. Relative to computer science, signals are similar to static variables (variables stored memory as opposed to being stored on the stack[351]). In a later chapter, we'll discuss VHDL constructs that are similar to local variables of computer programming fame (variables stored on the stack).

```
----------------------------------------------------------------
-- massively generic 4-bit right-shifting shift register
----------------------------------------------------------------
entity my_sr is
    port (   DATA_IN : in std_logic;
            DATA_OUT : out std_logic;
                 CLK : in std_logic);
end my_sr;

architecture my_sr of my_sr is
   signal s_D : std_logic_vector(3 downto 0);
begin
   process (CLK,DATA_IN)
   begin
      if (rising_edge(CLK)) then
         s_D <= DATA_IN & s_D(3 downto 1);
      end if;
   end process;

   DATA_OUT <= s_D(0);

end my_sr;
```

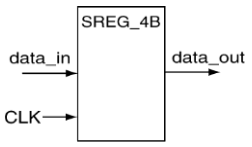
Figure 34.5: VHDL code for a basic 4-bit shift register.

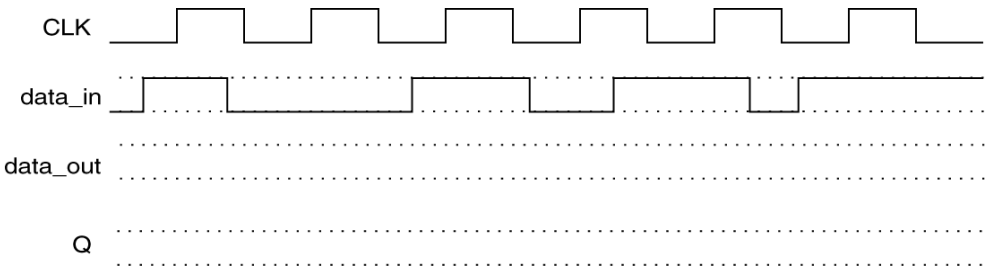
Another issue that usually surrounds shift registers is the notion of cascadeabilitly. If you're actually unfortunate enough to be forced to use shift registers on discrete ICs, you may have to use a bunch of them to actually obtain the data width that you need. For example, if you need a 64-bit SR, and all you have to work with are ICs containing 8-bit shift registers, you'll need to cascade[352] eight 8-bit shift registers in order to create a 64-bit SR. Sad as it seems, things really used to be done this way.

[350] But I certainly don't care because I sleep well at night knowing that the VHDL synthesizer is going to take care of the details for me.
[351] Don't worry if this does not make sense; it's computer science stuff and is designed to be confusing.
[352] In this context "cascade" is a fancy way of saying "connect up the part properly".

Example 34-1

Using the block diagram on the right to complete the timing diagram provided below. Consider the circuit to be a 4-bit shift register (shifts from left to right) that is active on the rising-edge triggered of the clock signal. Consider the line labeled "Q" to represent the 4-bit value stored by the shift register. Assume the "data_out" signal is the LSB of Q. Assume the initial value stored by the shift register is 0x8. Ignore all propagation delay issues with this circuit.





Solution: The interesting thing about this example is that the problem statement provides the initial value stored in the shift register. The other interesting thing is that the problem asks for what is being stored in the shift register despite the fact that only one-bit of shift registers contents appears as an output (the "data_out" signal is the output of the LSB).

Figure 34.6 shows the solution to this example. I'm omitting lots of written description as this is a great problem for you to work through yourself. Keep in mind that this is a right-shifting shift register, which means the "data_in" signal is the MSB of the shift register while the "data_out" signal is the LSB. Another happy thing to note about this problem is the fact that the shift register is dividing the current shift register contents by a factor of two (with truncation) when the "data_in" signal is a '0'. Convince yourself of this because this is a massively important and useful feature of shift registers.

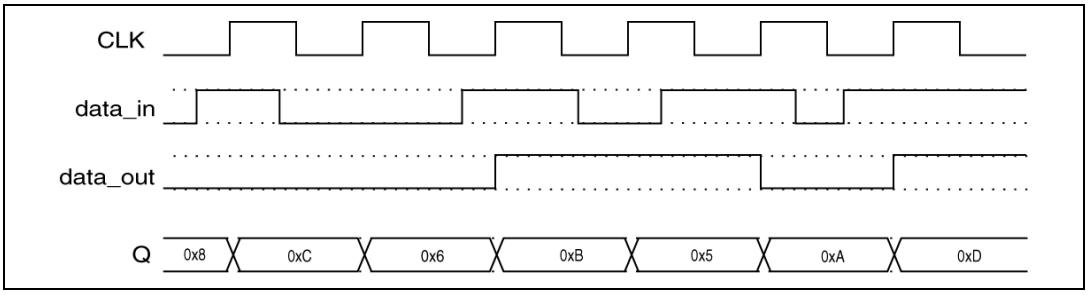


Figure 34.6: The solution to Example 34-2.

34.2.2 Universal Shift Registers

Shift registers that only shift in one direction are not that useful out there in digital-land. Most shift registers do many more operations such as shift left, shift right, parallel load, parallel clear, hold (don't change state), pick up the spare, etc. The term in digital-land for shift registers containing features such as these is "universal shift register". There is no one definition for universal shift registers; the only thing the term means is that you're dealing with some sort of shift register that does more than shift in one direction. From that point, you need to consult the datasheet or designer as to what exactly the device does.

In terms of digital design, you have the ability to easily design just about anything using VHDL. Because VHDL is so powerful, you can thus design devices at just about any level of abstraction. The following example problem picks a certain level of abstraction in its implementation of a universal shift register. There may be better ways to implement universal shift register, but the following approach is relatively efficient and somewhat instructive. Once again, the power of VHDL behavioral modeling is highlighted by the fact that no matter what features your universal shift register requires, it's rather straightforward to model in VHDL. The following quick example hopefully[353] shows that.

Example 34-2

Provide a VHDL model for an 8-bit universal shift register that supports the following operational characteristics. For this problem, assume the parallel load signal is synchronous and that all shift register operations are synchronized to the rising clock edge. The shift register's output should be only an 8-bit bundle that indicates the current state of the shift register.

- Hold
- Shift right
- Shift left
- Parallel load

Solution: The first step in this problem is to understand all of the features requested by the problem. The following list describes these features, in case you were actually wondering.

- Hold: This operation means that the shift register's contents do not change state on active clock edge.
- Shift Right: This is a typical shift right operation; this typically means that there needs to be an input into the shift register from the left side.
- Shift Left: This is typical shift left operation; this typically means that there needs to be an input into the shift register from the right side[354].
- Parallel Load: This implies that there needs to be an 8-bit bundle input that simultaneously loads all the shift register elements.

[353] We can only hope.

[354] You could also use the same signal for inputting signals for either shift-left or shift-right operations. The problem did not state how to do this so we have arbitrarily decided to have an input for both "sides".

From the above clarifications, we now know two types of information: the number and widths of the inputs and outputs required to complete this problem. Specifically, know the following; from this list of happy stuff, we can generate the block diagram shown in Figure 34.7. By the way, drawing a block diagram is still a great place to start problems such as they help you understand the problem and help you generate the VHDL entity. The following lists some other fun stuff.

1. The shift register has four unique operations: hold, shift-right, shift-left, and parallel load. This means we somehow need to control which operation will actually occur. We do this by adding a control signal that "selects" the desired operation. This signal is an input to the shift register and allows the shift register to be controlled by some external circuit. Since the shift register has four operations, we need a two-bit control signal that selects the desired operation.

2. We know all the inputs and outputs to the shift register. The problem states the outputs; they comprise of only the state of the shift register storage elements. The inputs include a 2-bit operation select signal, a 1-bit input for shift-left operations, a 1-bit input for shift-right operations, an 8-bit bundle for parallel loads, and a lively clock signal.

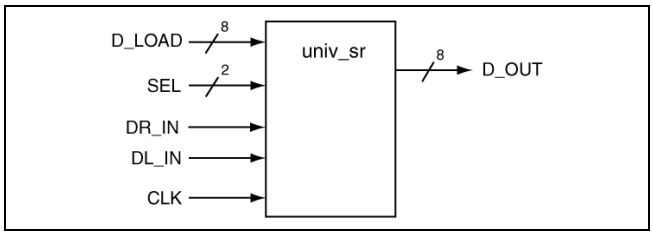


Figure 34.7: A black box diagram of the universal shift register.

Allow me to blather on about this before we get to the VHDL model. Figure 34.8 repeats Figure 34.1 for your viewing convenience; this diagram once again shows a generic schematic for a simple right-shifting shift register. The way you should think about the hardware-based solution to this problem is to imagine that each shift register storage element is now going to have to decide upon what value is going to be loaded on the next active clock edge.

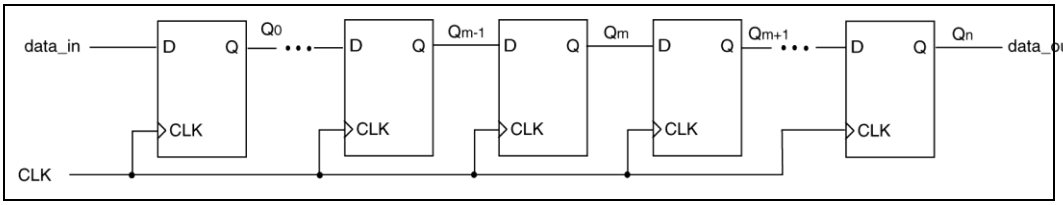


Figure 34.8: A typical n element shift register.

When you hear the word "decision" in digital design-land, you should think "MUX". If you think about it in this manner, it sure seems as if each storage element is now going to have its own MUX to decide which value is going to be loaded to the storage element. The notion here is each shift register storage element needs to decide which signal will be used to load the element. Figure 34.9 shows the schematic for the single shift register storage element that you're probably imagining.

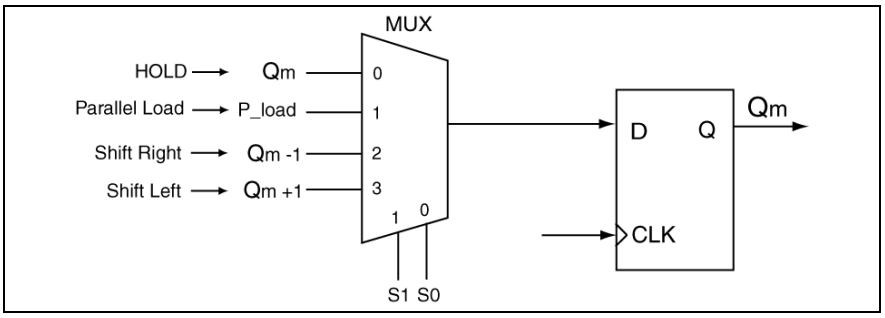


Figure 34.9: A shift register element with an attached MUX for data selection.

Figure 34.9 shows a 4:1 MUX with two control signals. The control signal selects between four different signals to load into the storage element in order to satisfy the problem. The MUX data signals shown in Figure 34.9 represent the following; the choice of MUX input index was arbitrary.

- **Qm** (0): The input to the D flip-flop is chosen to be the current output of the D flip-flop in question. Qm is an internal signal and ensures that the storage element does not change state by "reloading" its current value. In other words, the present state of the D flip-flop becomes the next state.

- **P_load** (1): The input to the D flip-flop is the appropriate signal from the parallel loading bundle input.

- **Qm-1** (2): The input is part of a shift right operation, which indicates the input to a storage element is the first storage element to the left of this storage element (this is confusing; check out Figure 34.8 for the details for the subscripted numbers).

- **Qm+1** (3): The input is part of a shift left operation, which indicates the input to a storage element is the first storage element to the right of this storage element (check out Figure 34.8 for clarification).

Table 34.1 summarizes the information in the previous list. So, all we need to do from here is to put a bunch of these in a row and call it a universal shift register? We could proceed with at this level, but let's instead bump up a level of abstraction in order to complete this problem. We need to use VHDL, so we may as well use it to make solving this problem as easy as possible.

S1	S0	D	Comment
0	0	Q_m	hold
0	1	P_load	parallel load
1	0	Q_{m-1}	shift right
1	1	Q_{m+1}	shift left

Table 34.1: Summary of the SR element functionality.

The design of a universal shift register using VHDL is straightforward and instructive. It's only slightly more complicated than just a simple D flip-flop (or simple register). To put it another way, if you understand how a D flip-flop is generated using VHDL, you'll easily understand the VHDL implementation of a universal shift register. Figure 34.10 once again shows the block diagram we'll use while Figure 34.11 shows the associated VHDL model.

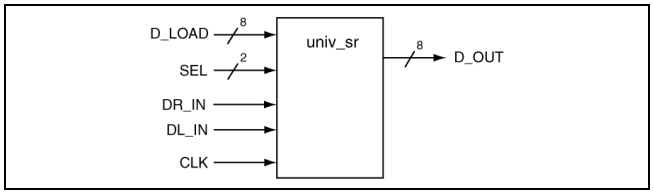


Figure 34.10: A black box diagram of the universal shift register.

```
-----------------------------------------------------------------
-- Model for a universal shift register
-----------------------------------------------------------------
entity univ_sr is
    port (    SEL : in std_logic_vector(1 downto 0);
            P_LOAD : in std_logic_vector(7 downto 0);
             D_OUT : out std_logic_vector(7 downto 0);
               CLK : in std_logic;
             DR_IN : in std_logic;  -- input for shift left
             DL_IN : in std_logic); -- input for shift right
end univ_sr;

architecture my_sr of univ_sr is
   signal s_D : std_logic_vector(7 downto 0);
begin

   process (CLK,SEL,DR_IN,DL_IN,P_LOAD)
   begin
      if (rising_edge(CLK)) then

         case SEL is
            -- do nothing (don't change state) --------------
            when "00" =>  s_D <= s_D;

            -- parallel load --------------------------------
            when "01" =>  s_D <= P_LOAD;

            -- shift right ----------------------------------
            when "10" =>  s_D <= DL_IN & s_D(7 downto 1);

            -- shift left -----------------------------------
            when "11" =>  s_D <= s_D(6 downto 0) & DR_IN;

            -- default case ---------------------------------
            when others => s_D <= "00000000";
         end case;

      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

Figure 34.11: VHDL code for the universal shift register.

The VHDL model in Figure 34.11 shows some cool stuff, as the list below happily mentions:

- The approach taken by this model is to “synthesize” the correct eight bits based on the value of the SEL input; these changes to the shift register storage elements are synchronized with the rising edge of the clock.

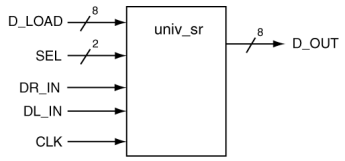
- There are a couple of instances of the "&" operator, which is the concatenation operation in VHDL. The operator concatenates two signals (or parts of two signals) together in order to synthesize the correct output based on the selected operation.

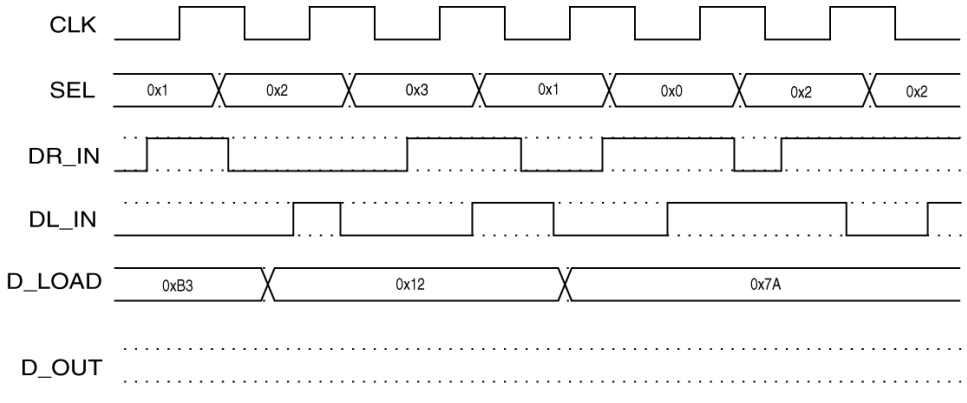
- The model takes the approach of assigning a temporary signal inside the process; once the process terminates, an assignment to the "s_D" signal causes this signal to be assigned to "D_OUT", which is the parallel output of the shift register.

Example 34-3

The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:

- SEL = "00": hold
- SEL = "01": parallel load of D_LOAD data
- SEL = "10": right shift; DL_IN input on left
- SEL = "11": left shift: DR_IN input on right
- All operations are synchronized to the rising edge of the CLK signal.
- Propagation delays are negligent.
- Initial D_OUT value is 0x45





Solution: The first step in any problem that contains a sequential circuit is to establish the initial state of the storage elements. This problem states that the initial value of D_OUT value is 0x45; this value is the initial state of the shift register.

From there, a good approach to problems such as these is to list what actions the SEL signal is selecting throughout the timing diagrams. Figure 34.12 shows a partially annotated timing diagram highlighting the operations selected by the SEL signal. Note that all of these annotations are synchronized with the rising clock edge.

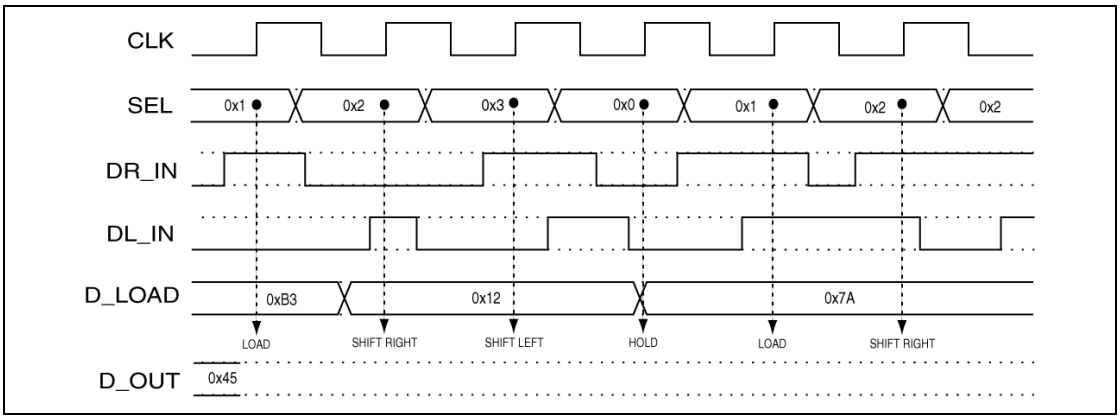


Figure 34.12: A black box diagram of the universal shift register.

Figure 34.13 shows the final timing diagram. As you can see, most of the changes in the DR_IN, DL_IN, and D_LOAD signals have no affect on the final output. The important thing to do for this problem is to verify for yourself that each of the values in the D_OUT are in fact correct.

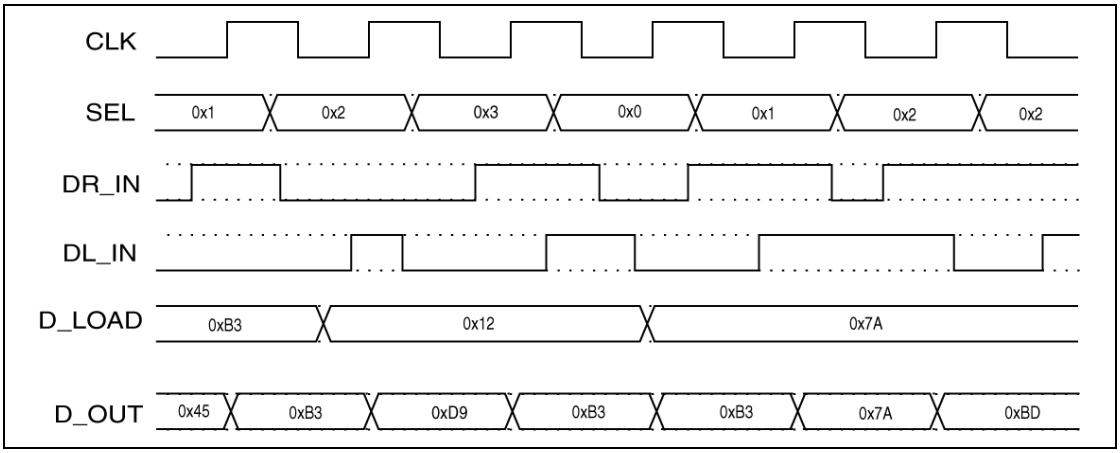


Figure 34.13: A black box diagram of the universal shift register.

34.2.3 Barrel Shifters

Once you get past the notion that shift registers do some actions that appear to be similar to “shifts”, you’ll find that there are other circuits out there that do similar shifting operations. Once of the common operations out there is a “barrel shift”. The operation of barrel shifters is straightforward as it’s simply an extension of simple shifting operations. While simple shift registers only performed one shift per clock cycle, barrel shifters are capable of performing more than one shift per clock cycle. As you would imagine, barrel shifters can shift either left or right.

The key to understanding barrel shifters is realizing the main reason they exist. Keep in mind that shift registers contain “bits” which generally represent binary numbers. The notion of shifting left and right are associated with multiplying (left shift) or dividing (right shift) by two. Thus, barrel shifters are then associated with multiplying and dividing by “powers of two” (such as 4, 8, 16, 32, etc.). What these

operations provide are super fast (namely, one clock cycle) multiply and divide operations. As you continue in digital stuff and/or computer programming, you'll find that multiplying and dividing binary numbers is extremely time consuming relative to other computer operations. Barrel shifters provide a cheap and fast, although somewhat limited alternative.

Barrel shifters are commonly used in arithmetic applications where 100% accuracy of results is not required. For example, there is always a big push to have your circuit perform "integer-based math" because working with integers is much less "computationally expensive" than working with other options such as "floating point numbers". A good example of this is with non-professional cameras such as the ones included with cell phones. Because cameras on these devices are partially judged by their speed of operation, they use integer math. Using integer math causes you to lose some precision, but your eyes will never know the difference. All you know is that your tiny hand-held device is able to take high definition movies and do so with out delay. Big wup.

Table 34.2 shows two example barrel shifting operations. Both of these examples use an 8-bit value; the top example is the value before the active clock edge while the bottom value is the value after the active clock edge. The examples show both a starting and ending point for the barrel shifting operation described by the particular row in the table. The (a) row shows a 2x right barrel shift that arbitrarily inputs 0's on the left side of the register. The (b) row shows a 2x left barrel shift that arbitrarily inputs 1's from the right side of the register. The operation in the (a) row represents a divide by two; the operation in the bottom row is one the many open mysteries in this world.

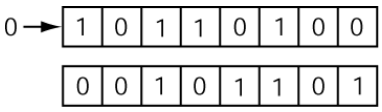
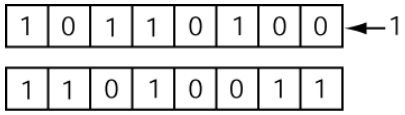
	Description	**Example**
(a)	barrel shift right 2x; stuff in a two 0's from the left side.	
(b)	barrel shift left 2x; stuff in a two 1's from the right side.	

Table 34.2: Examples of possible barrel shifting operations.

34.2.4 Other Shift Register-Type Features

But wait… it gets better: there are even more common shifting operations out there in digital land. Two more of the common shifting operations are rotates and arithmetic shifts. These operations are also simple in their basic states[355]. Rotate operations can be useful in many applications, though there is not one slam-dunk great example I can think of. Arithmetic shift operations are similar to simple shift operations but specifically work with signed binary numbers.

Rotate operations include rotate left or a rotate right with the actual shifting occurring on the active clock edge. The notion with rotate-type shifts is that no bits from the original register values are lost by "shifting them out" of the register as was the case with simple shift registers. Specially, for a rotate right

[355] The truth is that is can get really ugly out there. You many need to combine operations with as "barrel rotates" or "barrel arithmetic shift", or some type of shift to enhance your bowling skills. We won't go there in this chapter.

operation, the LSB of the register becomes the new MSB while all other bits are shifted one position to the right. For a rotate left operation, the MSB of the register becomes the new LSB while all other bits in the register are shifted one position to the left.

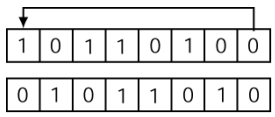
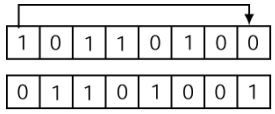
	Description	Example
(a)	rotate right; the LSB is transferred to the MSB;	
(b)	rotate left; the MSB is transferred to the LSB.	

Table 34.3: Examples of rotate-type shifts.

Arithmetic shifts are similar to simple shifts in their ability to perform mathematical operations[356]. The key different is that arithmetic shifts work with signed binary number and preserved the "signedness" of the value they operate on. For an arithmetic shift left operation, the value of the sign bit does not change as a result of the shift. Thus, the left shift operation retains the sign of the number as well as the ability to perform fast multiplication with the left shift operation. For an arithmetic shift right operation, the sign bit is both retained as a sign bit and propagated to the right with each shift. This sounds somewhat strange, but it truly both retains the sign of the value in the register as well as performing a fast division operation. I suggest working through a few examples on your own.

[356] When you read this paragraph, recall that we represent signed binary numbers using 2's complement notation, AKA, "diminished radix complement" notation.

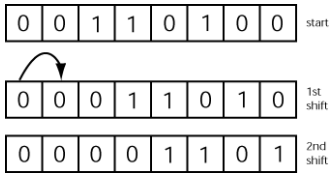
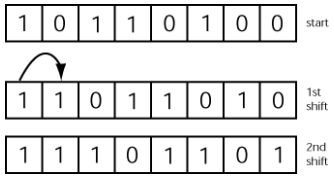
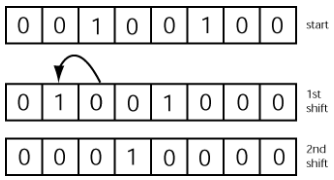
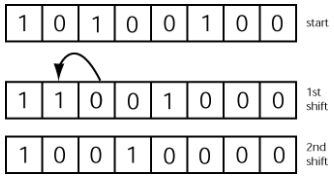
	Description	Example
(a)	An arithmetic shift right of a positive number in 2's complement form; the sign bit is copied from sign-bit position to the next bit on the right with each shift. This is a divide by two on a signed number (positive).	
(b)	An arithmetic shift right of a negative number in 2's complement form; the sign bit is copied from sign-bit position to the next bit on the right with each shift (the sign bit remains unchanged). This is a divide by two on a signed number (negative).	
(c)	An arithmetic shift left on a positive value in 2's complement form. The left shift does not alter the sign; all other bits shift left and a '0' is arbitrarily stuffed into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land[357]. This is a multiply by two on a signed number (positive).	
(d)	An arithmetic shift left on a negative value in 2's complement form. The left shift does not alter the sign bit; all other bits shift left and a '0' is arbitrarily stuffed into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a signed number (positive).	

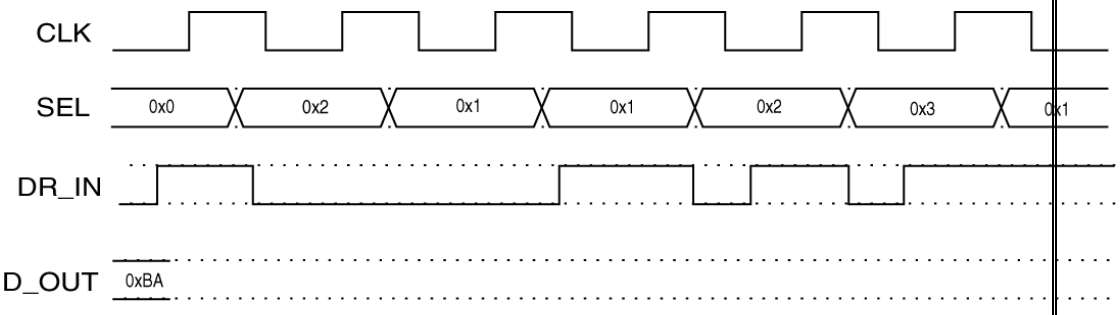
Table 34.4: Examples of many flavors of arithmetic shifts.

[357] A place where all academic administrators were born; a place where we all wish they would go back to as soon as possible.

Example 34-4

Using the following timing diagram, provide a VHDL model of an 8-bit shift register that performs the operations listed below. Make sure you also complete the timing diagram and provide a block diagram of the final circuit. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need to make in order to get past yet another poorly worded example problem. Assume the DR_IN signal is the bit that is an input on the right for shift left operations while shift right operations utilize the sign bit for an input. Assume D_OUT represents the 8-bit value stored by the shift register.

- SEL = "00": arithmetic shift right
- SEL = "01": arithmetic shift left
- SEL = "10": rotate right
- SEL = "11": rotate left



Solution: The first step in any problem that does not provide a black box diagram is to generate the black box diagram. From the problem statement we can see that the circuit's input are a clock signal (CLK), a selection signal (SEL), and a bit input signal (DR_IN). The only output of the circuit is the D_OUT signal, which represents the contents of the shift register. Figure 34.14 shows the final block diagram for this example problem.

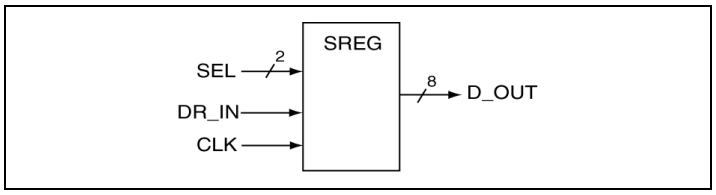


Figure 34.14: A black box diagram of the universal shift register of Example 34-4.

The next step is to annotate the provided timing diagram to explicitly show (in English) the operations selected by the SEL signal. This step is not necessary, but it ensures the mistakes you make are of the intelligent type rather than dumbtarted type. Figure 34.15 shows this intermediate helper step.

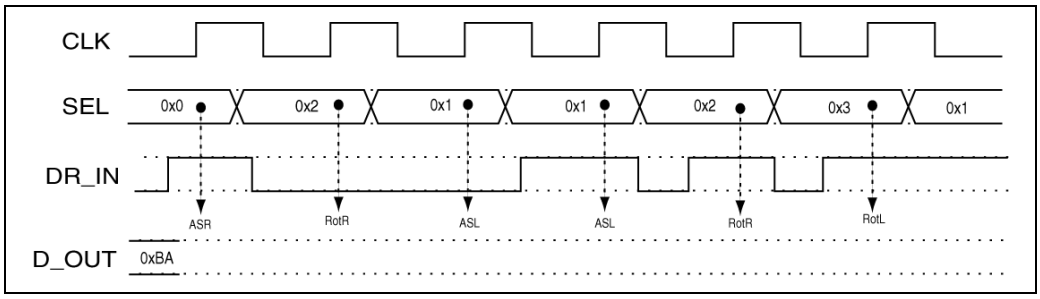


Figure 34.15: A black box diagram of the universal shift register of Example 34-4.

Without too much verbage, Figure 34.16 shows the final timing diagram solution to Example 34-4. One thing to note about this problem is that the circuit only uses the DR_IN input for arithmetic shift left operations.

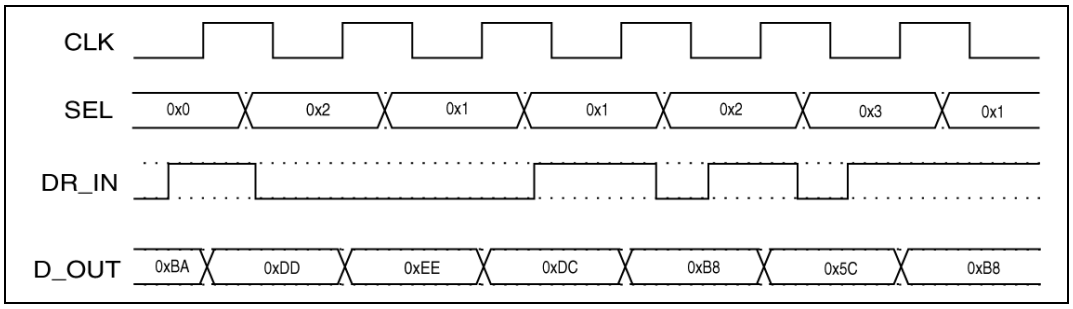


Figure 34.16: A black box diagram of the universal shift register of Example 34-4.

This example problem also states that we need a VHDL model also. This solution uses Figure 34.14 and the problem description as an aid in generating the VHDL model; Figure 34.17 shows the final VHDL model. The only worthy comment to make about this model is that the VHDL code models the "next values" of the shift register using a combination of the previous state of the shift register, the sign bit, and the DR_IN input. Liberal use of the concatenation operator also helps this model.

```
-----------------------------------------------------------------
-- Yet another shift register model
-----------------------------------------------------------------
entity sreg is
    port (    SEL : in std_logic_vector(1 downto 0);
            D_OUT : out std_logic_vector(7 downto 0);
              CLK : in std_logic;
            DR_IN : in std_logic); -- input for shift left
end sreg;

architecture my_sr of sreg is
   signal s_D : std_logic_vector(7 downto 0);
begin
   process (CLK,SEL,DR_IN)
   begin
      if (rising_edge(CLK)) then
         case SEL is
            -- arithmetic shift right
            when "00" =>  s_D <= s_D(7) & s_D(7 downto 1);

            -- arithmetic shift left
            when "01" =>  s_D <= s_D(7) & s_D(5 downto 0) & DR_IN;

            -- rotate right
            when "10" =>  s_D <= s_D(0) & s_D(7 downto 1);

            -- rotate left
            when "11" =>  s_D <= s_D(6 downto 0) & s_D(7);

            when others => s_D <= (others -> '0');
         end case;
      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

Figure 34.17: The final VHDL model for Example 34-4.

34.3 Counters: Yet Another Register Flavor?

Registers… it's really hard to underscore their popularity in digital design. Another massively common register is the counter. In its simplest form, a counter is a register that "counts", but also retains all the characteristics of a register (such as operations synchronized with a clock signal). While this sounds straightforward, the reality is that people make many assumptions when they use the term "counter". The list below describes these assumptions.

- Unless otherwise stated, a counter is actually a binary counter, meaning that it counts in binary. This is an important distinction because there is also the notion of a decade counter (we'll talk about that later) which does not count using a binary sequence. You can design a counter to count in any sequence, but a normal binary sequence is generally assumed unless stated otherwise[358].

- Related to the last issue is the notion that a counter only counts "up" unless otherwise stated. There are also counters that count down also (more on this later).

[358] Keep in mind that we previously designed counters using FSMs; many of these counters had wacky and pointless counting sequences.

- Once again related to the last issues is the notion that counters count up by a value of '1' on each clock cycle, unless stated otherwise. This means that a '1' is added (arithmetic addition) to the current count value on each clock edge. The notion of counting up by '1' as it relates to a counter is referred to as an increment. You can easily design counters that count up (or down) by any value.

When counters are the topic of discussion, you may hear many new and unusual words. The idea of counters is straightforward, meaning, I can't think of any new and amazing things to say about them that was not already been said. In addition, we've worked with counter in the context of FSMs many chapters ago. The approach I'll take here is to define and describe every word and/or term I've ever heard used in the context of counters and then do a few example problems. In truth, counters used to be a big deal back when you had to design them yourself using discrete logic. Now, discrete ICs have many flavors of counters, and more importantly, VHDL makes the modeling of counters almost trivial. I'll save all the verbage and remain at a high level of abstraction in regards to counter design.

When you say the word counter, it has a few standard connotations that you can assume are true unless told otherwise. The following list describes even more assumptions made when dealing with counters.

- Counters always refer to a sequential circuit. There are combinatorial counters out there, but they are somewhat rare and painful to think about.

- An active clock edge synchronizes a counter's traversing of the count sequence. Thus, there is one count value, or code-word from the count sequence at each clock cycle.

- A counter's output represents a specific and repeatable sequence of a given number of bits. This means that the sequence the counter "counts" in will not change; the bit-width of the counter won't magically change either.

- When a counter completes a traversal through its count sequence (either in the up or down direction), the counter automatically starts counting over.

Wow, those facts and definitions were so fun that we'll follow them up with a listing of vernacular and definitions typically associated with counters (and similar devices):

- Binary Counter: A counter that counts in a binary sequence. This means a 4-bit binary count sequence goes from 0-15, or 0x0 to 0xF (up direction).

- Decade Counter: A counter that counts in a binary coded decimal (BCD) sequence. This means a 4-bit decade counter will count from 0-9 (up direction).

- n-bit Counter: A counter that uses n-bits to represent each of the values in its count sequence.

- Up Counter: A counter that counts up (increasing count values in count sequence).

- Down Counter: A counter that counts only down (decreasing count values in count sequence).

- Up/Down Counter: A counter that can counter either up or down according to a selection input on the device.

- Increment: An operation associated with counters where '1' is added to the current value of counter.

- Decrement: An operation associated with counters where '1' is subtracted from the current value of counter.

- Counter Overflow: The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its largest representable value to its smallest value.

- Counter Underflow: The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its largest representable value to its smallest value.

- Cascadeable: A characteristic of many digital devices such as counters and shift registers that allow you to effectively increase the overall bit-width of devices providing inputs and outputs such that you can easily interface the devices. One such output is the "ripple carry out".

- Count Enable: A signal on counters that enables the counting operation of the counter when asserted and disables the counting when not asserted.

- Ripple Carry Out (RCO): A signal typically found on counters that indicate when the counter has reached its maximum count value (for an up counter). This signal often aids in cascading multiple counter devices. The RCO is often used to indicate when the counter has reached its minimum count value (for down counters).

- Parallel Load: A characteristic of a counter or shift register indicating that all the storage elements in the device can simultaneously latch external values.

34.3.1 A Modern Approach to Counter Design

There are many ways to design counters; the most efficient way is to model their behavior using VHDL. Let's skip over some of the older methods as they are primarily academic exercises and are not overly useful in modern digital design. Many old digital design textbooks list these older methods; it you need to know of them, get a copy of these older texts.

When I need a counter, I go right to the generic VHDL counter model I keep around for that purpose. Figure 34.18 shows my starting point when my digital designs require counters. This model contains the basic structure of a counter in addition to many of the features associated with counters. I typically start with this model because it has everything, and then remove the parts that I don't need. As you'll see from examining Figure 34.18, there are some worthy things to note:

- The counter looks a lot like all the other register models we've been discussing.

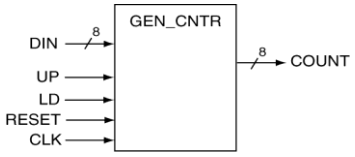
- The counter overflows when it increments at its maximum value (0xFF → 0x00) and underflows when it decrements at its minimum value (0x00 → 0xFF). These operations occur automatically so there is no need to design them into the counter.

- The counter has an asynchronous reset signal. This could easily be changed to a synchronous reset if my design so desired.

- The parallel load signal is synchronous and takes precedence over the other basic counter operations listed in the model.

- The counter has a signal dedicated to the counter direction. The model indicates that if UP is asserted (a positive logic signal), the counter counts up. If the UP signal is not asserted, the counter counts down.

- The counter uses the "+" operator for incrementing and the "-" operator for subtraction. This is somewhat of an advanced concept in VHDL modeling that you may or may not have encountered by now. In short, VHDL can model mathematical operations if the proper library files are included.

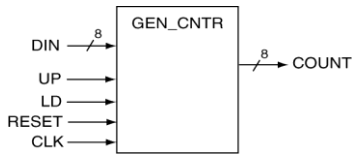


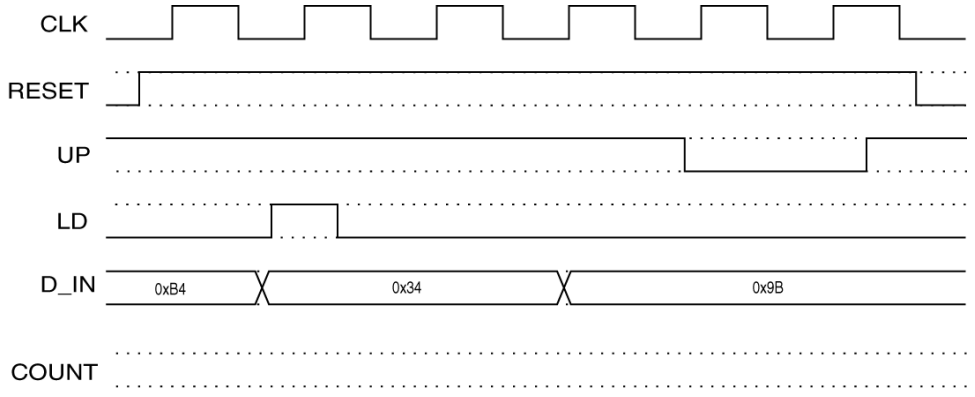

```
--------------------------------------------------------------------
-- Counter: synchronous up/down counter with asynchronous
-- reset and synchronous parallel load.
--------------------------------------------------------------------
entity GEN_CNTR is
   port ( RESET,CLK,LD,UP : in std_logic;
                      DIN : in std_logic_vector (7 downto 0);
                    COUNT : out std_logic_vector (7 downto 0));
end GEN_CNTR;

architecture my_count of GEN_CNTR is
   signal  s_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, RESET, LD, UP)
   begin
      if (RESET = '1') then
         s_cnt <= (others => '0');  -- asynchronous clear
      elsif (rising_edge(CLK)) then
         if (LD = '1') then
            s_cnt <= DIN;  -- load
         else
            if (UP = '1') then
               s_cnt <= s_cnt + 1; -- increment
            else
               s_cnt <= s_cnt - 1; -- decrement
            end if;
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

Figure 34.18: Generic VHDL model of a counter that does everything.

Example 34-5

The block diagram on the right shows a model of an 8-bit counter. The VHDL model associated with this block diagram appears in Figure 34.18. Use the block diagram and VHDL model to complete the following timing diagram. Assume propagation delays are negligent.





Solution: This problem attempts to show you everything interesting and useful with counters. Figure 34.20 shows the final solution to this example; the following verbage describes some of the more interesting things about the solution. In this case, the interesting things are when the output changes and what causes those changes.

1) The circuit was initially in a reset condition. On this active clock edge, the counter output is incremented due to the assertation of the UP signal.

2) The UP signal is still asserted, but due to the way the LD signal is modeled, it takes precedence over the LD signal. Thus, the output loads the value on the D_IN input into the counter.

3) This is an increment operation due to the assertation of the UP signal.

4) This is another increment operation due to the assertation of the UP signal.

5) This is a decrement operation since the UP signal is no longer asserted.

6) This is another decrement operation since the UP signal remains unasserted.

7) This is a register clear operation due to the assertion of the RESET signal.

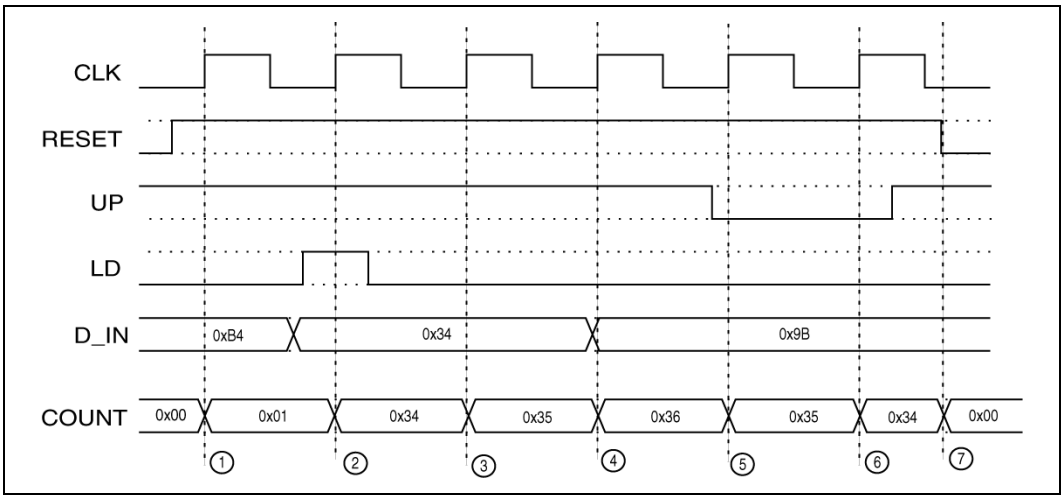


Figure 34.19: The solution (with annotiations) to Example 34-5.

34.3.2 Up-Down Counters

One popular term out there in digital design-land is the notion of an "up-down counter". This counter is nothing more than a counter that has a control signal the enables the counter to count either up or down. I've included a special section for this counter simply because the term is so popular. The generic counter model in Figure 34.18 contains the code that makes it into an official up/down counter. Let's do an example problem for this counter and then move on.

Example 34-6

Provide a block diagram and a VHDL model for a synchronous 8-bit up/down counter. The output of the counter should the 8-bit count only.

Solution: The first step in this problem is to find out what the problem is looking for and then generate a black box diagram. This problem wants an up/down counter, but, it does not state that it needs parallel loading capabilities or any type of asynchronous presets or clears. Figure 34.20 shows the block diagram we'll work from.

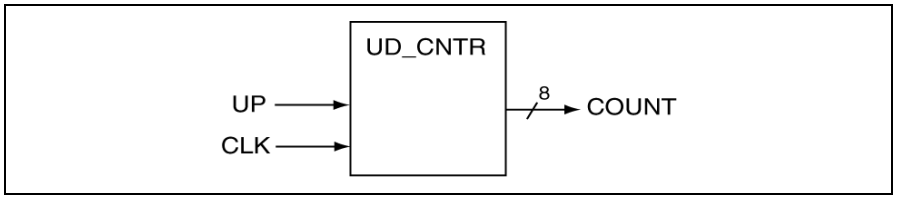


Figure 34.20: A black box diagram of the universal shift register of Example 34-4.

After generating the block diagram, we need to grab our generic counter model and modify it in order to satisfy the problem description. Figure 34.21 shows the final solution to this example problem.

```
-----------------------------------------------------------------
-- No Frills Synchronous 8-bit Up/Down Counter
-----------------------------------------------------------------
entity UD_CNTR is
   port ( CLK,UP : in std_logic;
          COUNT : out std_logic_vector (7 downto 0));
end UD_CNTR;

architecture my_count of UD_CNTR is
   signal  s_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, UP)
   begin
      if (rising_edge(CLK)) then
         if (UP = '1') then
            s_cnt <= s_cnt + 1; -- increment
         else
            s_cnt <= s_cnt - 1; -- decrement
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

Figure 34.21: VHDL model for a simple 8-bit up/down counter.

34.3.3 Decade Counters?

Another common counter you occasionally hear about is the decade counter. While counters are generally considered to be binary counters (unless specified otherwise), non-binary counters count in some sequence other than binary. A decade counter counts in a decimal sequence much like a binary coded decimal number. That is, the output of the counter shows a 4-bit binary zero through nine ("0000" → "1001") rather than a binary zero through fifteen ("0000" → "1111"). The counters can be quite useful as the non-computer portion of the world is still decimal[359]. It sounds like we need to do an example problem .

Example 34-7

Provide a block diagram and a VHDL model for a synchronous 8-bit decade counter. The output of the counter should the 8-bit count only.

[359] Excluding the academic administrative portion of the world which is still using stone age binary

Solution: The problem describes a two-digital decimal counter; each of the digits is represented in binary using binary coded decimal numbers. This problem only has a clock input as no other features were requested. Figure 34.20 shows the block diagram we'll work from.

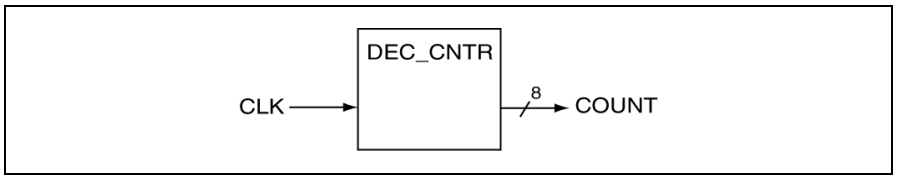


Figure 34.22: A black box diagram of the universal shift register of Example 34-4.

Figure 34.23 shows the final VHDL model for this example. This solution is by no means unique; it was the first thing I thought about when solving this problem. This problem highlights the strength of behavioral modeling in VHDL. The only problem I see with this solution is the notion that there are three levels of nesting in the "if" statements. This makes me nervous; so I will flag that in my brain and make this the first module I look at if my circuit is not working properly.

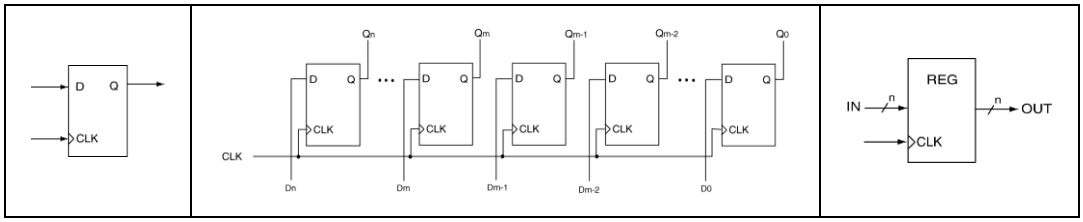
```
-----------------------------------------------------------------------
-- No Frills Synchronous 2-digit Decade Counter
-----------------------------------------------------------------------
entity DEC_CNTR is
   port (    CLK : in std_logic;
            COUNT : out std_logic_vector (7 downto 0));
end DEC_CNTR;

architecture my_count of DEC_CNTR is
   signal  s_cnt_tens : std_logic_vector(3 downto 0);
   signal  s_cnt_ones : std_logic_vector(3 downto 0);
begin
   s_cnt_tens <=
   process (CLK)
   begin
      if (rising_edge(CLK)) then
         if (s_cnt_ones = "1001") then
            s_cnt_ones <= "0000";
            if (s_cnt_tens = "1001") then
               s_cnt_tens <= "0000";
            else
               s_cnt_tens <= s_cnt_tens + 1; -- increment tens digit
            end if;
         else
            s_cnt_ones <= s_cnt_ones + 1;    -- increment ones digit
         end if;
      end if;
   end process;

   COUNT <= s_cnt_tens & s_cnt_ones;

end my_count;
```

Figure 34.23: VHDL model for a simple decade counter.

34.4 Registers: The Final Comments

A register is nothing more than a set of bit storage elements that share a single clock signal. In other words, registers are a parallel configuration of signal bit storage elements; what makes them parallel is the fact that the individual storage element operations are generally synchronized to some event (usually a clock edge). A single bit storage element is easily modeled as a D flip-flop; if you line up a bunch of D flip-flops together and synchronized their actions with a clock edge, you have a register.

Once you abstract all of these matters to a higher level, you'll forever more speak about *n-bit registers*. Figure 33.9 shows the progression of this abstraction. One thing to note here is that the black box diagram of a register shown in Figure 33.9 (c) includes a clock signal. The level of abstraction here sometimes continues to the point of not including the synchronizing signal (in this case, the clock) in the block diagram. In these cases, the register is assumed to have a clock signal and is interpreted accordingly. Additionally, you can safely assume that all registers are edge-triggered unless told otherwise.



(a) (b) (c)

Figure 34.24: The progression from D flip-flop to register block diagram for n-bit register.

The definition of the register provided in Figure 33.9 is general enough to encapsulate everything you know about registers up to this point. The registers we've previously looked at included several common sequential circuits such as shift registers and counters. The main difference between the many types of register is their feature set. In an attempt to show all the possibilities in one spot, Table 34.5 shows a possible breakdown of the register types and their relation to each other. Keep in mind that many of the features listed in Table 34.5 can be either synchronous or asynchronous.

Register Type	Sub-Types	Features
simple register		not much
better register		parallel load, preset, clear, load enable, cascadeability
shift register	Universal Shift Registers, Barrel Shifters	parallel load, preset, clear, load enable, shift left/right, arithmetic shift left/right, hold, rotate left/right, cascadeability
counters	Up/Down Counters, Decade Counters	parallel load, preset, clear, load enable, increment, decrement, cascadeability

Table 34.5: The feature progression of the device referred to as a register.

Chapter Summary

- **Shift Registers:** Shift registers are in many ways similar to simple register; their primary different is with the inputs to the individual shift register storage elements. Shift registers are designed such that the data output from one shift register element becomes the data input to a contiguous element. IN this way, data is said to be "shifted through" the shift register. In general, there is one "shift" per clock cycle. Shift register operations are often used to implement fast but limited mathematical operations with single left shift being a divide-by-two and a single right shift being a multiply by two.

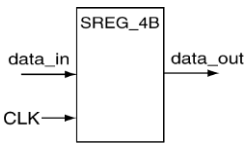
- **Universal Shift Register:** A type of shift register that performs more operations than a simple shift register. These operations can typically include both a shift left and a shift right, a parallel load, a preset and/or clear. Somewhere in here could also be arithmetic shift operations and various forms of rotate operations.

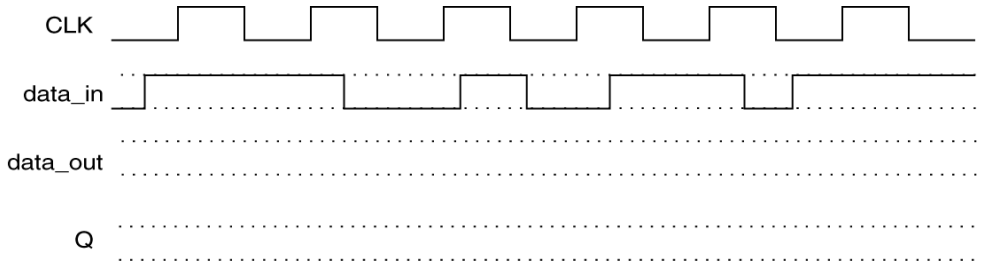
- **Barrel Shifters:** A type of shift register that performs multiple shifts on a single clock edge. In reality, barrel shifters are wired such that they can shift multiple bit locations in one clock cycle, and probably do not perform multiple shifts. Barrel shifters are useful for mathematical operations including multiplication and division by powers of two.

- **Counters:** A generic term for a device that traverses a set sequence on a given clock edge. There are many ways to design counters, the most efficient and modern approach is to use VHDL modeling. There are many types of counters out there including binary counters, up/down counters, decade counters, ring counters, Johnson counter, bowling counters, etc.

Chapter Exercises

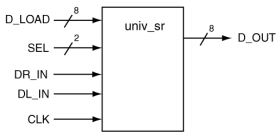
1) Use the block diagram on the right to complete the timing diagram below. Consider the circuit to be a 4-bit shift register (shifts from right-to-left) that is active on the rising-edge triggered of the clock signal. Consider the line labeled "Q" to represent the 4-bit value stored by the shift register and the "data_out" output to represent the value of the highest order bit stored by the shift register. Assume the initial value stored by the shift register is 0xC. Ignore all propagation delay issues with this circuit

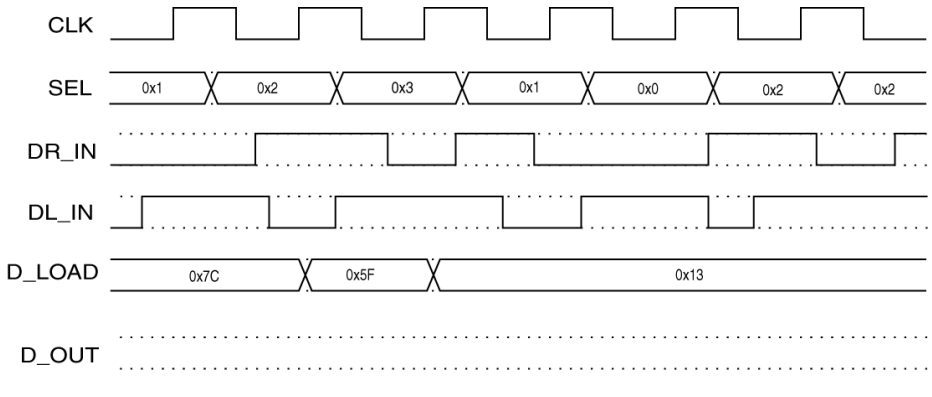




2) The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:

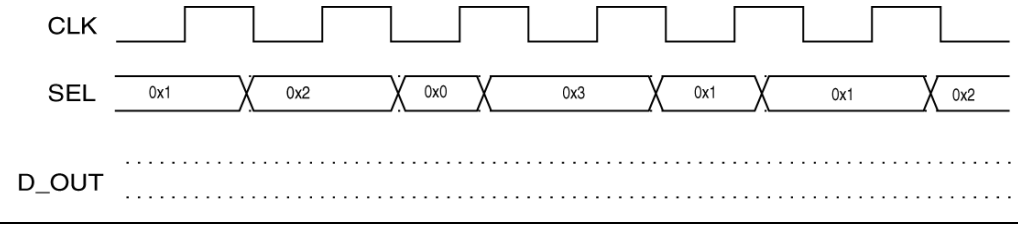
- SEL = "00": hold
- SEL = "01": parallel load of D_LOAD data
- SEL = "10": right shift; DL_IN input on left
- SEL = "11": left shift: DR_IN input on right
- The rising edge of the CLK signal synchronizes all shift register operations
- Propagation delays are negligent.
- Initial D_OUT value is 0xAB





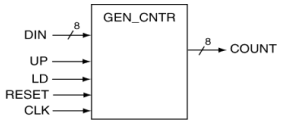
3) Using the following timing diagram, provide a VHDL model of an 8-bit shift register that performs the operations listed below. Make sure you also complete the timing diagram and provide a block diagram of the final circuit. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need to make in order to complete this problem. Assume the 0x39 is the initial value stored by the shift register. Assume "D_OUT" is an 8-bit output representing the value stored by the shift register.

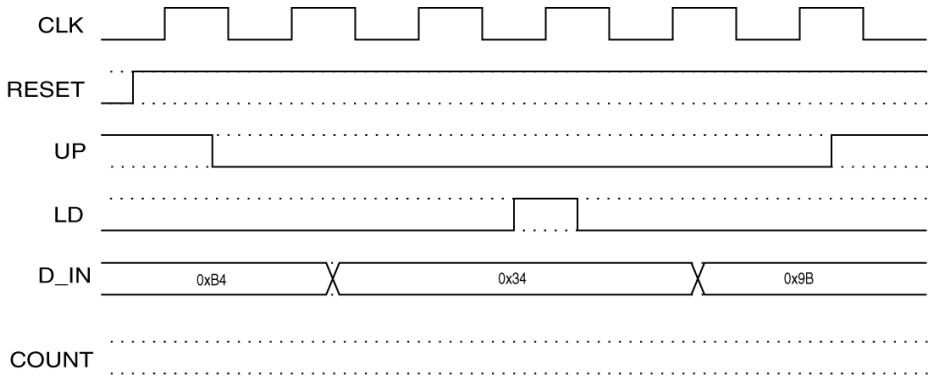
- SEL = "00": rotate right
- SEL = "01": rotate left
- SEL = "10": divide by 8 (bit stuff 0's)
- SEL = "11": multiply by 8 (bit stuff 0's)



4) The block diagram on the right shows a model of an 8-bit counter. Use the following assumptions in order to complete the following timing diagram. Assume propagation delays are negligent.

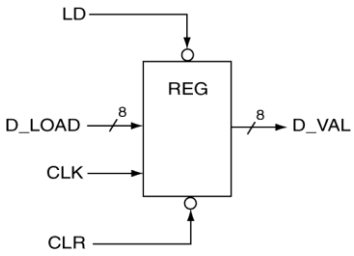
- The LD input enables the loading of the DIN input to the counter
- The RESET input is an asynchronous and active low used to reset the counter
- The COUNT output shows the current value stored by the counter
- The counter counts up when the UP input is asserted (active high) or down otherwise. All count operations are synchronous.



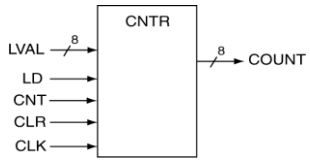


5) Provide a VHDL model that supports the black box diagram of the register on the right. Make the following assumptions for this problem.

- LD synchronously loads the value of D_LOAD into the register.
- D_VAL is the 8-bit value stored in the register.
- CLR is an asynchronous input that resets the register when asserted. This input takes precedence over the LD input.

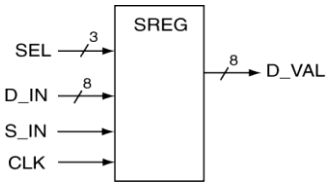


6) Provide a VHDL model that supports the black box diagram of the counter on the right. Make the following assumptions for this problem.



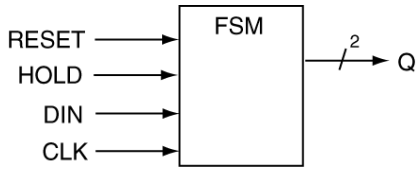
- LVAL is loaded to the counter synchronously when LD is asserted
- COUNT is the value stored in the counter.
- CLR synchronously resets the counter and takes precedence over all other inputs.
- CNT is the input of lowest precedence and instructs the counter to count up by '1' if asserted and '2' otherwise

7) Provide a VHDL model that supports the black box diagram of the shift register. Make the following assumptions for this problem.

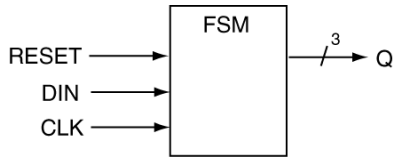


- The S_IN input is used for all single-bit shift left and shift right operations where the input value is not stated.
- D_IN is the 8-bit input value used for parallel load operations.
- The SEL input synchronously chooses the following operations
  - SEL = "000": shift left (stuff '0' on right)
  - SEL = "001": shift left
  - SEL = "010": shift right (stuff '1' on left)
  - SEL = "011": shift right
  - SEL = "100": rotate left
  - SEL = "101": divide by 4
  - SEL = "110": divide by 16
  - SEL = "111": multiply 8

8) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 2-bit shift register. Consider the Q output to be a 2-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs. When the HOLD input is asserted, the Q output does not change.



9) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 3-bit shift register. Consider the Q output to be a 3-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs.



10) The following diagram shows a circuit that is used to perform a serial-to-parallel conversion on the OP_A and OP_B input and then perform a mathematical operation. In other words, two four-bit numbers will be provided serially (LSB first) on the OP_A and OP_B inputs. The two tables below describe the MUXes and the Universal Shift Register (USR).

- Provide a state diagram that could be used to control the circuit such that it performs ***A - B*** and registers the result in REG_ACC (A & B are the parallelized versions of the OP_A & OP_B serial data). The serial to parallel conversion will initiate when the signal GO (not shown) is asserted. Minimize the number of states in your design. State any other assumptions you deem necessary.

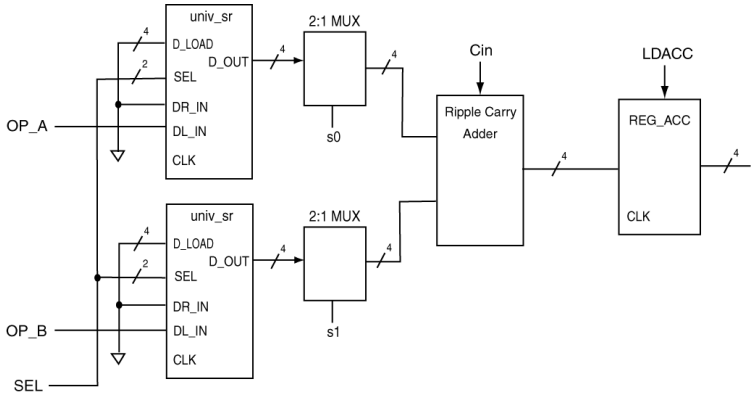
MUX description

```
if (sx = 0) then
   out <= in;
else
   out <= not in;
end if;
```

Assumptions:

- LSB is first to arrive in serial bit stream
- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All Shift register operations are synchronous

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



11) The following diagram shows a circuit that can perform a mathematical operation. The two tables below describe the MUXes and the Universal Shift Register (USR). The registers have a synchronous load input (LD). Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.*

- If a GO signal is received (GO is not shown in diagram), the following operation is generated and the result appears on the output: **OP_OUT = (*OP_B* - *OP_A*) ÷ 16**

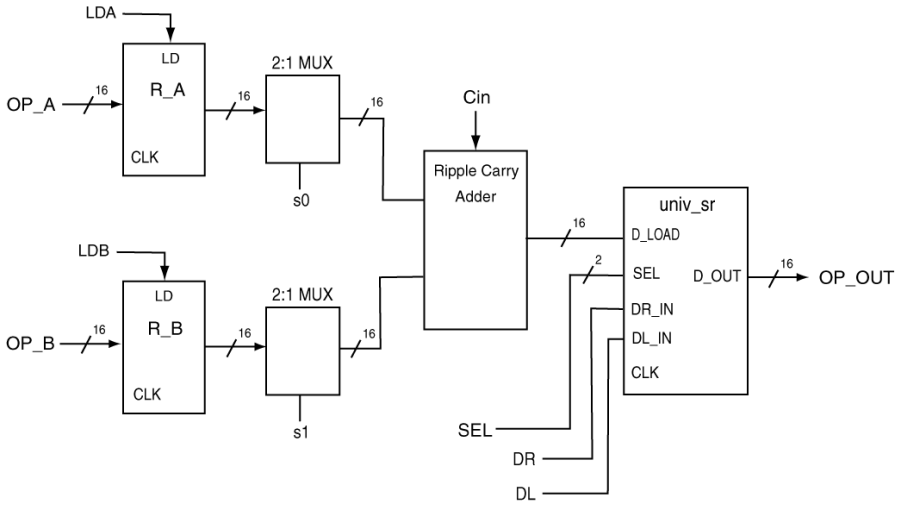
MUX description

```
if (sx = 0) then
   out <= in;
else
   out <= not in;
end if;
```

Assumptions:

- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All Shift register operations are synchronous
- Registers (non-USR) have synchronous load inputs (LD)

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



35 Chapter Thirty-Five

(Bryan Mealy 2012 ©)

35.1 Chapter Overview

No matter how you look at it, computers represent a major portion of the digital design experience. First, modern digital design uses personal computers as a major design tool. Secondly, we can argue that an underlying objective of learning digital design is to understand the notion of "computers" at many different levels. You are quickly gathering digital design skills; you're really not that far away from designing a circuit that is officially a "computer". This does not mean that you'll soon be designing a "PC", but a PC is not the only form of computer out there in digital-land.

This chapter presents a brief and high-level view of computers and then moves on to the describing one aspect of a computer: the Arithmetic Logic Unit, or ALU. This is another one of those subjects that people write books about and get PhDs for, so we'll not attempt to present the end-all of computer and/or ALU descriptions. What we will present is an overview of ALUs constructed first with low-level hardware, and then modeled at a high level using VHDL.

Main Chapter Topics

- **COMPUTER ARCHITECTURE OVERVIEW:** This chapter presents a brief high-level view of computer architecture as a motivation to introduce ALUs.
- **LOW-LEVEL ALU DESIGN:** This chapter describes a low-level yet limited approach to ALU design. This approach is primarily a review of modules we've already worked with. .
- **HIGH-LEVEL ALU DESIGN:** This chapter describes ALU design using some of the high-level modeling techniques available in VHDL.

Why This Chapter is Important

This chapter is important because it describes several approaches to designing ALUs. This description includes an introduction to the use of variables in VHDL.

35.2 Computer Architecture Overview

Because this chapter is primarily concerned with ALU design, we'll first develop the proper context for ALUs. The ALU is a major component in computer design, so our approach in this section is to describe computers at a high-level. This will hopefully order provide an understanding of the purposes and possibilities for ALUs.

The term architecture appears often in digital-land, so often that you'll find it to have many different meanings depending on the context it is used in. The best definition for "architecture" in a hardware context is that the architecture of circuit describes the individual modules of a circuit and the connection between the modules. Based on this definition, we can substitute the word "architecture" any time we've used the term "block diagram"[1].

The notion of an "arithmetic logic unit" has no solid definition. Though it sounds like a circuit that contains both arithmetic and logic units, you'll find that is not always true. The modern use of the term ALU is attachable to any digital circuit since you can certainly argue that any digital circuit necessarily performs logic operations. Probably the most useful definition of an ALU is a circuit that has inputs for one or more operands, inputs for one or more controls, and output for the results. The ALU tweaks the operands as directed by the control signals in order to generate a required result. Thus, the ALU is a box that tweaks data and generates a result; by no means is this said tweaking limited to arithmetic and logic operations.

35.2.1 Computer Architecture in a Few Paragraphs

A computer is a digital system, which means that it is comprised of a bunch of gates and things that are connected in some intelligent manners. From a higher level, a computer can be viewed as nothing more than a special connection of all the standard digital circuits you've learned about up until now, plus others that you'll learn soon[2]. A computer is no different than any of the other digital systems you've worked with except that it is generally more complex. But then again, the complexity comes from the sheer amount of simple elements in the circuit and not the elements themselves[3].

What is a computer? The definition we'll work with in this discussion: A computer is any electronic device that reads instructions from memory and carries out those instructions on data. Somewhere in this definition, we need to include the notion that the computer is able to interface with the outside world, so our computer must be able to handle various input and output needs. The instructions essentially tell the computer what operations need to take place on the associated data[4].

Figure 35.1 shows the basic model of a computer that we attempted to describe in the previous paragraph. As you can see, a computer is comprised of three main components: the central processing unit (CPU), memory, and input/output (I/O). The memory block stores the "instructions" that the computer executes while the I/O block allows the outside world to interface with the computer. The item we're slowly working our way to is the block labeled "CPU".

The CPU is an acronym that stands for "central processing unit" Once again, there is a lot we can say about the CPU, but we'll keep it to a comfortable minimum in this discussion. As you can see from Figure 35.1, the CPU is comprised of two main blocks: the "datapath" and the "control unit". The control unit interfaces with the computer instructions read from memory and tells the datapath what to do with data.

The notion of the "central" processing unit came from days where hardware was massively expensive, both on the discrete level and on the silicon level. Things are slightly different these days though. As the underlying IC fabrication techniques become better and allow for smaller digital circuitry, the required processing in computers can less typically be described as central. Having more processing

[1] And often times that is what people do.
[2] You can survive this discussion without knowing these items.
[3] Once again, the key to understanding complex issues such as computers is to divide the associated digital circuitry into more manageable blocks. This form of abstraction is absolutely required because even the simplest computer is arguably complex. Lucky for us that VHDL structural modeling fully support this flavor of abstraction.
[4] Instructions do other things that we're not mentioning here in order to keep things simple.

units increases the overall throughput of the computer, which allows more advanced features on the devices in which these ICs appear. Where would the world be now without a device that allows you to call your buddy and watch a stream of video data all on the same device and at the same time? Now that's progress!

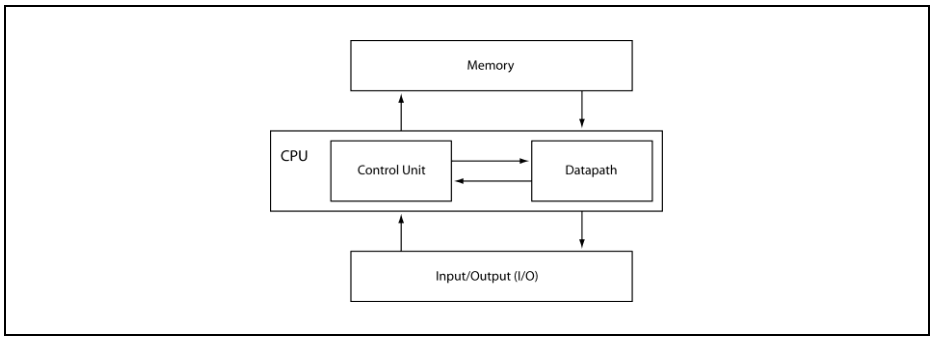


Figure 35.1: A block diagram for a basic computer architecture.

Figure 35.2 shows a more detailed diagram of the CPU. From this diagram, you can see that data is passed into the datapath and then passes out of the datapath. During this datapath traversal, the data is tweaked according to the instructions that were read from memory. Note that the control unit controls the datapath; this control includes receiving status from the datapath.

The datapath is the bit-crunching heart of the central processing unit (CPU). As was mentioned earlier, the datapath is a giant circuit that is filled with such a great number of simple devices that it becomes somewhat complex to study if your examination is at too low of a level. These simple devices include many types of digital circuitry, which is interconnected in some intelligent and organized fashion that produces the desired result.

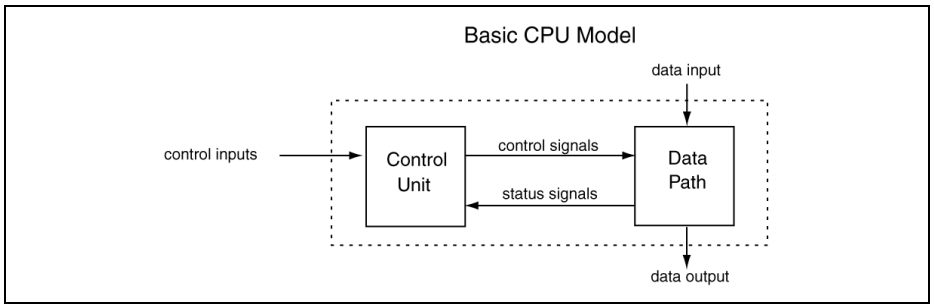


Figure 35.2: A block diagram for a basic computer architecture.

The point we're trying to arrive at with this discussion that the ALU forms the bulk of the datapath block and is therefore considered to be one of the basic building blocks of the CPU. Datapaths contain a lot of useful and interesting circuitry; they only thing we'll consider here is the ALU portion of the CPU. In summary, the ALU is part of the datapath which is part of the CPU, which is one of the three major functional blocks of a computer.

Figure 35.3 shows a lower-level diagram of a simple ALU. This is roughly the model we'll be working with in this chapter, but once again, this is by no means the only approach to ALUs out in digital-land.

There are really no guidelines on how to model an ALU, which becomes more true once we start using VHDL to model ALUs.

In accordance to the acronym "ALU", Figure 35.3 shows that this particular model of an ALU contains two sub-blocks including the "arithmetic unit" and the "logic unit". In theory, all the arithmetic functions go into the arithmetic block while all of the logic functions go into the logic block. This particular ALU contains two operands: A and B[5]. The width of the operands is arbitrary as indicated by the "n" width of the operands. The S1 signal is a bundle that instructs the arithmetic unit and logic unit as to what operation to perform[6]. The width of the S1 signal depends upon what level of control is required by the two units; the more operations performed by these units, the higher value for the number "m". This model indicates that the arithmetic and logic units share the control signal; as a result, arithmetic and logic operations from these two blocks occur simultaneously. The S2 signal is another control signal that chooses between either the arithmetic or logic result to exit the ALU.

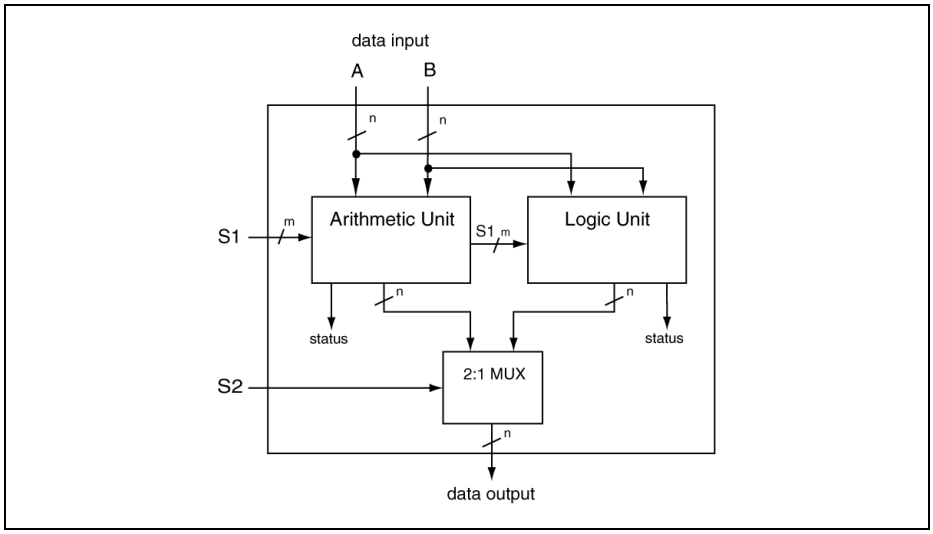


Figure 35.3: A block diagram for a basic ALU.

35.3 Low-Level ALU Design

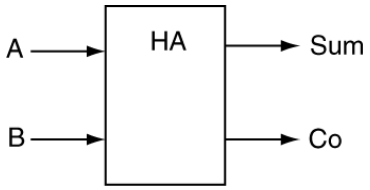
You can choose one of many different ways to design ALUs. The approach we take in this section is a low-level approach that represents a review of our arithmetic-type standard digital design modules. As you will see, this low-level approach is somewhat limited, particularly when you compare it to using VHDL to model ALUs on a behavioral level. Behavioral modeling of ALUs is the topic of an upcoming section, and I bet you can hardly wait.

35.3.1 The Arithmetic Unit

The half adder (HA) was the first circuit we designed. The HA was simple, useful, and also represented the first arithmetic circuit we designed; it gave us a view of the vast usefulness and endless possibilities

[5] ALUs can have as many operands as you feel like designing into them, though they usually have one to three operands.
[6] The associated computer instruction is decides what operation needs to be executed.

of digital design. The humble HA added two 1-bit numbers; the results included a sum and carry output. Figure 35.4 shows the block diagram and the associated truth table defining the operation of the HA.

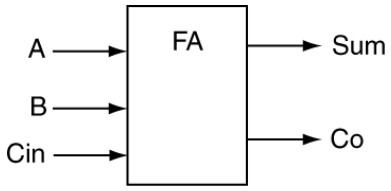


A	B	Sum	Co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) (b)

Figure 35.4: The block diagram (a) and truth table (b) for the half adder.

The HA circuit was an effective learning tool but it was not overly applicable in a real circuit. The problem with the HA was that its simplicity limited its usefulness; we could not use the circuit in a modular manner to obtain a circuit that was capable of adding more than one bit at a time. The particular limitation with the HA was the fact that it could not handle the needed input from other devices. In order to make this device useful in a modular manner, it needs to have an input that accepts and processes a carry out (Co) from another similar device. The solution to this problem leads to the notion of a full adder (FA). The FA circuit contains both a carry in (Cin) as well as a carry out (Co). Figure 35.5 shows the black box diagram and associated truth table describing the FA.



A	B	Cin	Sum	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

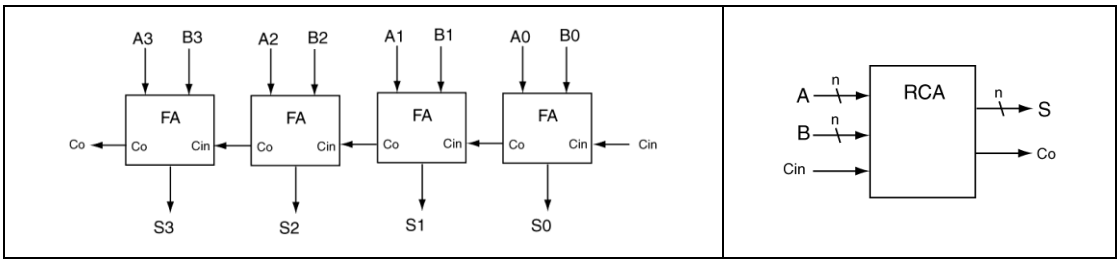
(a) (b)

Figure 35.5: The full adder block diagram (a) and the associated truth table (b).

Although the FA is only capable of adding one bit at a time, it has the capability of being configured to add n-bits. Making the move from a one-bit adder to an n-bit adder is done by applying the iterative modular design (IMD) approach. The next step towards developing an arithmetic unit is to configure a set of *n* adders in such as way as to form an *n-bit* adder. This configuration requires the parallel placement of *n* FAs with the Co output of each adder driving the Cin input of the adder. In this way, the Co output of the lesser significant FA is used to drive the Cin input of the next more significant FA in the parallel configuration.

Figure 35.6 shows schematics and diagrams of a 4-bit ripple carry adder (RCA). This name comes form the notion that the carry from the less significant bits may need to transition towards the more

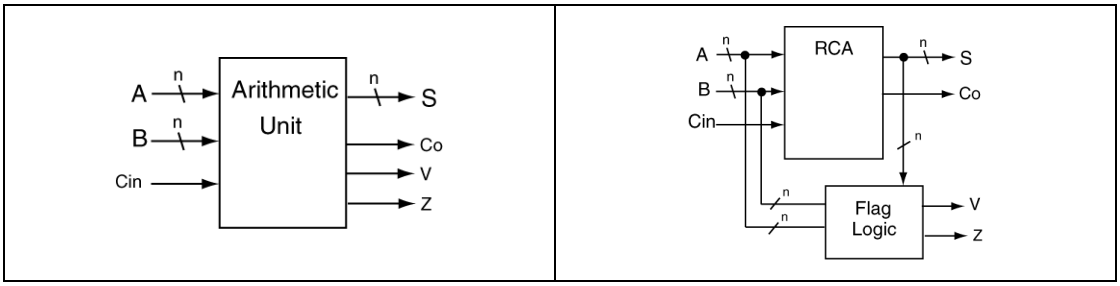
significant bits before the output becomes valid. This feature delays the final sum output until the carry ripples through the individual FA elements in the RCA. Delayed results are undesirable in any digital circuit. However, what the RCA lacks in speed, it makes up for in simplicity.



(a) (b)

Figure 35.6: A diagram of a 4-bit adder (a) and the related n-bit ripple carry adder black box (b).

The next step in our design of an arithmetic unit is to include a few circuit additions that will be useful to us later in our discussion. Figure 35.7 shows the circuit we want to design. Note there are two extra outputs included in this circuit that were not included in the RCA circuit: V and Z. In reference to Figure 35.3, the V output indicates that an overflow condition[7] exists as a result of the arithmetic operation. The Z output indicates when the result of the arithmetic operation is zero. Specifically, if all the bits in the sum are zero, the Z output is set to '1'; otherwise the Z output is '0'. Figure 35.7(b) shows the approach we'll take to design the logic for the V and Z outputs; the "flag logic" will massage the augend, addend, and Sum in order to generate the V output; the Z output is a function of the S output only.



(a) (b)

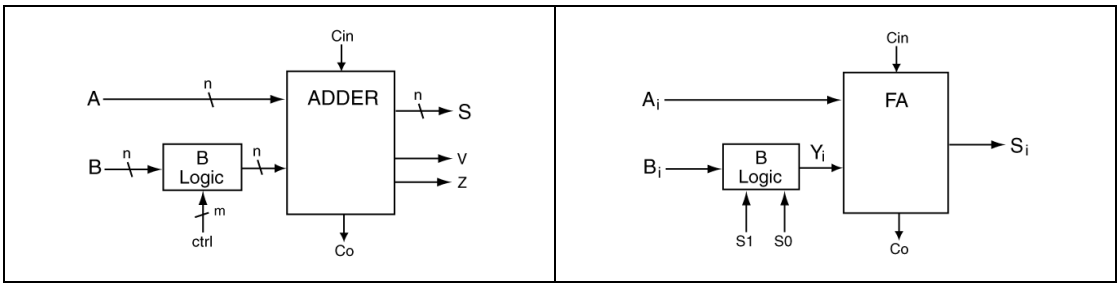
Figure 35.7: The full arithmetic unit we'll design (a) and the circuit we'll use to design it (b).

Figure 35.8 shows the circuit diagrams we'll use to generate our arithmetic unit module. The approach we'll take is to massage the logic to the B input of the "adder". Note that we're referring to the box as an "adder" as it no longer modeled as a RCA because of the V and Z outputs. As you'll see, if we can tweak some of the input values, we can generate several useful mathematical operations beyond the basic adding operations advertised by the RCA module.

Keeping with our digital design theme, we'll be designing the circuit shown in Figure 35.8(a) using a modular approach. Figure 35.8(b) shows the main element in this approach: the friendly FA module.

[7] In this context, an overflow condition is where the sign of the two numbers being added is the same, the sign bit of the sum is different.

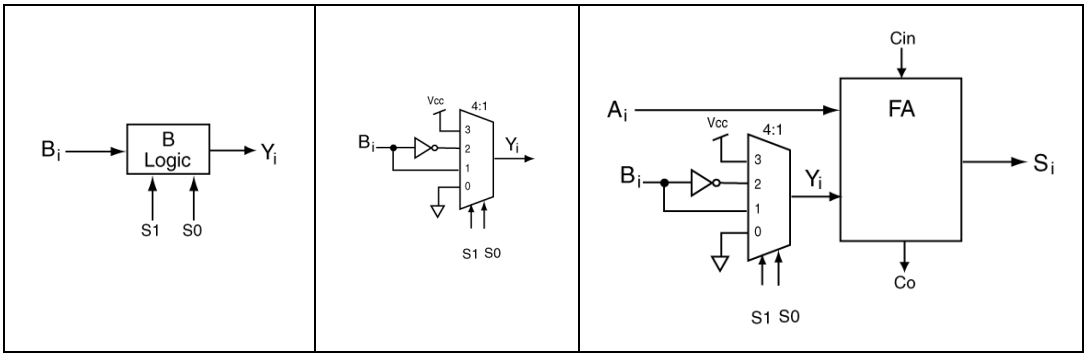
Note that the FA shown in Figure 35.8(b) is drawn at the bit-level (using the subscripted i notation). We'll be designing the logic in the "B Logic" box of Figure 35.8(b). Note that there are two inputs to this box, which implies there are four different outputs for any given B input.



(a) (b)

Figure 35.8: A block diagram of the arithmetic unit (a) and the bit-level internal element (b).

Figure 35.9(a) shows the B Logic block on the bit level. Note that this block contains three inputs (one data input and two control inputs) and one output. Yi is the arbitrary name given to the output. The two control inputs to this block "select" between one of four possible tweaks to the B input. Since the B input is only a one-bit value, the four possible tweaks to this value are 1) set the value, 2) toggle the value, 3) do nothing to the value, and 4) clear the value. Figure 35.9(b) shows the resultant MUX circuit. Placing the circuit of Figure 35.9(b) into the block diagram of Figure 35.8(b) results in Figure 35.9(c).



(a) (b) (c)

Figure 35.9: The B Logic block (a), the internal circuitry (b), and the final circuit (c).

The circuit shown in Figure 35.9(c) is a one-bit element of our final arithmetic unit. If we place "n" of these units in parallel, we would have an n-bit arithmetic circuit, which is what we've set out to do. The trick here is that by tweaking the B input, we'll be able to do more operations than the addition that is the main function of the FA element. The next task is then to see what operations our unit is capable of by listing all the possibilities in truth table format. Table 35.1 shows these possibilities.

The values in the final two columns of Table 35.1 are generated by keeping in mind that the interior of the arithmetic unit is an RCA and performs the operation shown in Equation 35.1, The output of the RCA (Sum or S) is the Sum of the inputs (A & B) added to the carry in (Cin). The mathematical operation shown in Equation 35.1 generates the information shown in the two right-most columns of

Table 35.1. The thing to remember here is that the augend and addend can be considered signed binary numbers; Table 35.1 uses is 2's compliment notation.

$$\text{Sum} = S = A + B + Cin$$

Equation 35.1: Equation for of the output of the arithmetic unit.

S1	S0	Y	Cin = 0	Cin = 1
0	0	all 0's	S = A + 0 + 0 = A (transfer)	S = A + 0 + 1 = A + 1 (increment)
0	1	B	S = A + B + 0 = A + B (addition)	S = A + B + 1 (??)
1	0	$\overline{B}$	$S = A + \overline{B} + 0 = A + \overline{B} = (??)$ (??)	$S = A + \overline{B} + 1 = (A - B)$ (subtraction)
1	1	all 1's	S = A - 1 + 0 = A - 1 (decrement)	S = A - 1 + 1 = A (transfer)

Table 35.1: The table showing possible arithmetic operations under S1 and S0 control.

The moral so far in this arithmetic design effort is that by including a chunk of logic that massages one of the operands to the RCA, the arithmetic unit went from a single operation (adding), to five valid and useful operations: addition, subtraction, increment, decrement, and a transfer[8]. Wow! You gotta love digital logic design.

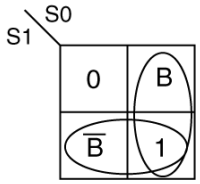
As in interesting point, we could reduce the circuitry required by the B Logic and retain the same functionality. In other words, it's possible to implement the B Logic with less circuitry than the MUX as is shown in Figure 35.9(c). The approach to this is to once again go back to your digital roots and start with a truth table as is shown in Figure 35.10(a). The Y variable represents the output of the B Logic circuit and is placed into the compressed Karnaugh-map shown in Figure 35.10(b). Note that this K-map lists B as a mapped entered variable (MEV). Equation 35.2 shows the resulting Y logic; Figure 35.10(c) shows the final bit-level circuitry.

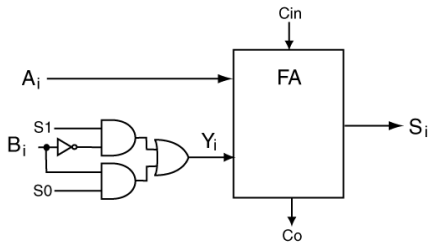
$$Y_i = S1 \cdot \overline{B_i} + S0 \cdot B_i$$

Equation 35.2: Yi output equation.

[8] You won't see it anytime soon, but the transfer action is quite useful.

S1	S0	B	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1





(a) (b) (c)

Figure 35.10: The truth table (a), compressed K-map with MEV (b), and resulting B Logic circuitry (c).

This concludes what we need to do for the arithmetic portion of this circuit. We now need to provide some logic that generates the V (overflow) and Z (zero) outputs. The best approach is to use the fact that the V output will be set when the sign of the augend and addend are equal to each other but different from the sign of the result. Figure 35.11(a) shows that you can model this statement in truth table form. Note that in this truth table, the (n-1) bit position is the left-most bit (zero referenced) and hence, the sign bit for an n-bit signed binary number. Figure 35.11(b) shows the resulting equation.

$A_{(n-1)}$	$B_{(n-1)}$	$S_{(n-1)}$	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

(a)

$$V = \overline{A_{(n-1)}} \cdot \overline{B_{(n-1)}} \cdot S_{(n-1)} + A_{(n-1)} \cdot B_{(n-1)} \cdot \overline{S_{(n-1)}}$$

(b)

Figure 35.11: The truth table (a) and resulting equation for V (b).

The logic for the Z signal is more straightforward than the development of the overflow indication signal. The Z output is a '1' whenever the result of the logic operation is zero. In other words, if all the bits in the Sum are 0's, the Z output will be set; otherwise, it will be zero. This is results in a NOR operation; Figure 35.12 shows this amazing circuit.

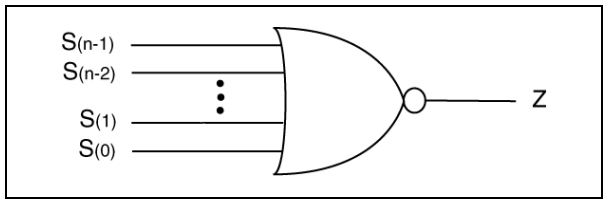
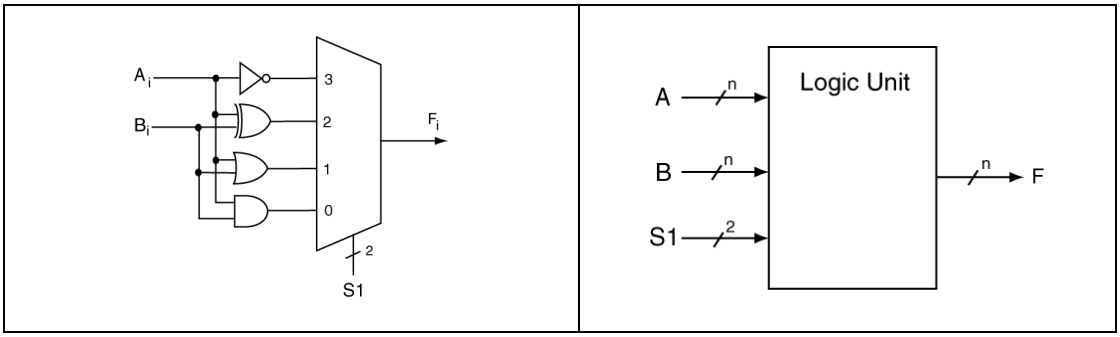


Figure 35.12: The logic for the Z output.

35.3.2 The Logic Unit

Designing the logic unit is so straightforward that we'll not need to spend much time here. What logic you choose for your ALU is up to you. For this discussion, we'll arbitrarily give our logic unit the ability to invert, XOR, OR, or AND. Specifically, the logic unit can apply these operations as follows: 1) compliment a single operand, 2) XOR two operands, 3) OR two operands, or 4) AND two operands. The logic unit will choose one of these operations based on the logic's units two control inputs. We've chosen four operations because we are reusing the control inputs that we used in the arithmetic unit (the ones that controlled the B Logic block). In our original diagram of Figure 35.3, recall that both the arithmetic and logic units share the same control inputs.

As with the arithmetic unit, we'll first show that we can model the logic unit at the bit-level. Figure 35.13(a) shows a 4:1 MUX; we consider each bit associated with the A & B operands to be input to one of these MUXes as part of the logic unit. The module of Figure 35.13(b) results from assembly "n" of the circuits shown in Figure 35.13(a); we refer to this as an "n-bit logic unit".



(a) (b)

Figure 35.13: The circuitry (a) and black box diagram (b) for our logic unit.

Consider for a moment that we now want to show a logic diagram for an 8-bit logic unit. We could draw eight of the circuits shown in Figure 35.13(a), but that would be a massive waste of time. A better approach would simply be model the logic unit using VHDL. Figure 35.14 shows a VHDL model for our logic unit. VHDL sure makes things much simpler.

```
entity logic_unit is
   Port (  A,B : in std_logic_vector(7 downto 0);
            S1 : in std_logic;
             F : out std_logic_vector(7 downto 0));
end logic_unit;

architecture lu of logic_unit is
begin
   with S1 select
      F <= (not A)   when "11",
           (A XOR B) when "10",
           (A OR B)  when "01",
           (A AND B) when "00",
            X"00"    when others; -- hex notation
end lu;
```

Figure 35.14: The VHDL model for the logic unit.

To complete this approach to ALU design, Figure 35.15 shows the final ALU block diagram. Note that this diagram includes three status outputs form the arithmetic unit. Any number of status inputs could have also been included from the logic unit, but we opted not to include any. One standard approach to including status outputs from both the arithmetic and logic units is to input a given status signal from each unit into a MUX. In this way, the signals selecting the individual operations on the units would also select which status output exits the ALU module.

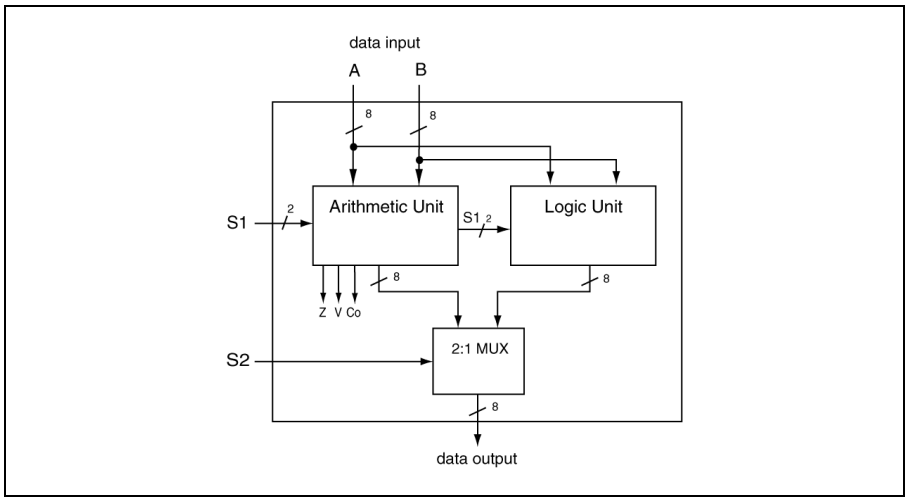


Figure 35.15: The entire ALU (8-bit form) for this section.

In the end, we now have an ALU that does quite a few operations. Table 35.2 provides a summary of the operations our ALU design can do. Here are a few things worth noting:

S1 is a 2-bit bundle, which means there are four control inputs to this ALU design. Note that we consider Cin to be a control input because it allows the ALU to generate several operations in the arithmetic unit.

- With four control inputs, we could maximally choose between 16 different operations with this ALU. Because the logic unit does not use the Cin input, the logic unit loses only has two control signals and thus only performs four operations (a loss of four operations for the ALU).

Additionally, two control input options provide us nothing in the arithmetic side. The ALU thus performs ten operations. In truth, we have listed the "transfer A" option twice, so our ALU only performs nine unique operations.

S2	S1	Cin	Operation
0	00	-	Compliment A
0	01	-	A XOR B
0	10	-	A OR B
0	11	-	A AND B
1	00	0	Transfer A
1	00	1	A + 1 (increment A)
1	01	0	A + B (addition)
1	01	1	-
1	10	0	-
1	10	1	A – B (subtraction)
1	11	0	A – 1 (decrement A)
1	11	1	Transfer A

Table 35.2: A summary of out ALU operations.

35.4 VHDL Modeling: Signals vs. Variables

After reading through the previous section, you may find yourself hoping for a better way to design ALUs. As you may guess from the logic unit design, modeling ALUs using VHDL is straightforward. What makes this modeling straightforward is the use of a new type of VHDL object: the variable. You've been extensively using signals up to this point and hopefully found that those were no big deal. However, as you start designing more complex circuits, you'll find that using strictly signals in your designs can be very limiting. The use of variables frees you from those constraints; they also simplify the modeling of ALUs using VHDL.

Variables usage in VHDL is similar to signal usage; there are only two minor differences. We'll soon describe the two major differences as well. The minor differences lie in the notion of variable declaration and assignment. We'll describe both variable definition and assignment in the context their similarities to signals.

35.4.1 Signal vs. Variables: The Similarities

Figure 35.16 shows the similarities and differences between signal and variable declaration. Note that signals are declared as "signal" types and variables are declared as "variable" types; aside from that, the declarations are similar. Note that the initialization of signals and variables is the same; we can initialize them if we need to, but initialization is optional. Also, note that variables can be of subtype "std_logic" as we're used to using in the context of signals.

```
signal s_sig1 : std_logic;
signal s_sig2 : std_logic := '1';
signal s_vec1 : std_logic_vector(0 to 3);
```

(a)

```
variable v_sig1 : std_logic;
variable v_sig2 : std_logic := '1';
variable v_vec1 : std_logic_vector(0 to 3);
```

(b)

Figure 35.16: The circuitry (a) and black box diagram (b) for our logic unit.

Figure 35.17 shows the differences between signal and variable assignment. The major difference here is that assignment to signals use the signal assignment operator ("<=") while assignment to variables use the variable assignment operator (":="). Also note that variables can be assigned literal values as we've done extensively with signals. Finally, not that signals can be assigned to variables and variables can be assigned to signals. The notion here is that the VHDL language allows the overloading of these two assignment operators[9].

```
s_sig1 <= s_sig2;  ;
s_sig2 <= '1';
s_vec1 <= X"E";
s_sig1 <= v_sig1;
```

(a)

```
v_sig1 := v_sig2;
v_sig2 := '1';
v_vec1 := X"E";
v_sig1 := s_sig1;
```

(b)

Figure 35.17: The circuitry (a) and black box diagram (b) for our logic unit.

35.4.2 Signal vs. Variables: The Differences

There are three major differences between signal and variable usage. The first major difference is that variables can only be defined in the declarative region of process while signals can only be declared in the declarative region of architectures[10]. Conversely, variables can't be declared in the declarative regions of architectures while signals can't be declared in the declarative regions of processes.

Now that we've made these statements, there are a few more issues to consider. An unmentioned similarity between signals and variables is the fact that after they are assigned, they retain their values. This is an easy statement to make, but to make this more memorable, we'll soon use it in an example.

There second major difference between signals and variables is their visibility in the VHDL model. Since signals are declared in the declarative region of the architecture, a signal can be referenced anywhere in that architecture. However, since a variable declaration is part of a process statement,

[9] Many VHDL operators are overloaded; check the VHDL specification for full details. I actually don't know what they are and I have to check them myself when the issues arises.

[10] There is actually more to the story than this. This text does not currently cover the notion of "functions" and "procedures" in VHDL. These two VHDL constructs have yet more rules for signals and variables that we'll not mention here.

variables are only visible and thus usable in the process in which they are declared. This is similar to the notion of "scope" in higher-level computer programming languages[11].

The final main difference between signals and variables relates to how the VHDL synthesizer interprets them. The best way to show and described this difference is with an example. For the following example, we'll model the well-known ripple carry adder (RCA) using variables. Recall that up until now that we have only modeled RCAs using VHDL structural models. Our explanation of the differences between signals and variables should be obvious after this example.

Example 35-1

Model an 8-bit ripple carry adder (RCA) using VHDL variables.

Solution: Figure 35.18 shows a solution to this example. The main point of this example is that it highlights the difference between variables and signals. Honestly, if this problem could be done without the use of variables, than I admit I don't know how. The issue here is that problems such as these are easily modeled with variables; it therefore makes no sense to find a better way to solve the problem.

The final main difference between signals and variable is that in process, the results of an assignment to a variable is available to immediately use in the process while assignments made to signals are "scheduled" to occur once the process suspends[12]. There fact has two main ramifications. First, you can use the result of a variable assignment within the process. Second, you can assign signals in processes are many time as you need to in the process; the only assignment that actually occurs is the last one seen in the process. Similarly, one way to think about signals and variables is that they store intermediate results of calculations. With variables, you can use this result immediately; with signals, the true results are not available until the process suspends.

Here are a few more worthy items to note regarding the solution shown in Figure 35.18. The number in the list below corresponds to the comments in the code.

(1) This is the variable declaration; the variable is of a std_logic_vector type of nine bits. The reason for declaring a 9-bit type with an 8-bit adder will soon become evident.

(2) This is the main addition operation. Because the variable is nine bits, we need to append an extra bit to the A & B operands; this is done with an ampersand, the operator VHDL uses for concatenation. The Cin input is also added to the "v_res" variable. Note that this statement advertises the notion that the addition operator (+) in VHDL is overloaded[13]. This is because we're adding a std_logic type to a std_logic_vector types and assigning the result to a variable. Because this is a variable assignment (note the ":=" operator), the new value is assigned to v_res immediately.

[11] Once again, the notion of VHDL functions and procedures have their own rules of visibility for signals and variables. Check a VHDL reference for details.

[12] There are two ways a process can suspends. The only way you may know now is that execution suspends once the end of the process is reached. The other way, which is covered in the chapter on testbenches, is when a VHDL "wait" statement is encounter. See the chapter on testbenches for more details.

[13] Be aware that this overloading and the liberal use of the "+" operator is based on the notion that we have included the proper libraries with our VHDL model. We always omit the library uses clauses in this text in an effort to save paper and space.

(3) At this point in the execution of the process, the v_res variable now contains the result. The SUM signal is an 8-bit signal while the v_res variable is a 9-bit signal, so we are only interested in the lower eight bits of v_res. This is signal assignment operation, so the actual assignment to the SUM signal does not occur until the process suspends. In VHDL terms, this assignment "is scheduled" to occur.

(4) As it turns out, the most significant bit of the v_res variable is the carry-out we're looking for. The MSB of the v_res result is assigned to the Co signal. Because this is signal assignment, the assignment is scheduled; the actual assignment occurs when then process suspends.

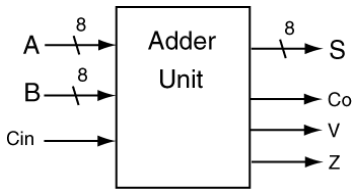
```
entity RCA_8bit is
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
end RCA_8bit;

architecture RCA_8bit of RCA_8bit is
begin
   process(A,B,Cin)
      ------------------------------------------------------(1)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      ------------------------------------------------------(2)
      v_res := ('0' & A) + ('0' & B) + Cin;

      ------------------------------------------------------(3)
      SUM <= v_res(7 downto 0);

      ------------------------------------------------------(4)
      Co <= v_res(8);

   end process;
end RCA_8bit;
```

Figure 35.18: The solution to Example 35-1.

There is one final item to note about the solution to Example 35-1. VHDL is versatile and powerful, but it lacks intelligence. As a result, you the designer need to account for many issues, particularly when arithmetic operations are involved. In the solution to this problem, the VHDL designer needs to understand that the addition of two eight-bit operands generates a nine-bit result. This is why we declared the v_res variable as nine bits. If we had not done this, we would not be able to know the value of the carry-out generated from the operation. This is true with other operations, namely multiplication.

Example 35-2

Model the following 8-bit adder unit using VHDL. Consider the S output to be a sum, the Co output to be a carryout, the Z output indicates when the S values is zero, and the V output indicates overflow from the addition of signed numbers. Consider the Cin input as a carry-in.



Solution: The first thing notice about this problem is that is it is similar to Example 35-1, which means we can use most of that solution for this problem. The main difference with this problem is that we now need to deal with an overflow (V) and zero (Z) indicator signals. The power of VHDL makes this somewhat trivial. Trivial things are good.

Here are a few more worthy items to note regarding the solution shown in Figure 35.18. The number in the list below corresponds to the comments in the code.

(1) This portion of code handles the case of the zero indicator. When the result is zero, we need the Z to be '1'; otherwise it is a '0' (can you say "positive logic"). In digital terms, we use a comparator and compare the bits forming the SUM to 0x00. As you may recall, this is easily modeled using VHDL; note that this section of code includes both an if and an else statement, which is good for reasons we'll get into in a later chapter.

(2) This piece of code handles the case of the overflow indicator. The notion here is that there is an overflow is defined to be when the sign of the two operands are equivalent, but different from the sign of the result. This piece of code implements what the idea stated in the previous sentence. There are two items of interest here[14]. First, the V output is initially assigned to '0' before this piece of code is encountered. The thought here is that the code may then re-assign the signal as part of the if statement. While this may seem strange, you always want to make sure all of the signals you're using are assigned somewhere[15] at least once. The other thing to note here is that we use the VHDL "not equals" operator: "/=", not to be confused with a similar operator in the C programming language.

[14] Please realize that there are many approaches to each section of this code; I made the subjective call that this is the clearest approach.

[15] This is for a reason that we'll discuss in a later chapter. In case you want to know now, the reason is that we're making sure something is always assigned to V in a valiant effort to avoid generating a latch.

```
entity adder_unit is
   port  (  CIN : in std_logic;
             A,B : in std_logic_vector(7 downto 0);
          CO,V,Z : out std_logic;
               S : out std_logic_vector(7 downto 0));
end adder_unit;

architecture my_adder of adder_unit is
begin
   process (A,B,Cin)
      variable v_sum : std_logic_vector(8 downto 0);
   begin

      v_sum := ('0' & A) + ('0' & B) + CIN;

      CO <= v_sum(8);

      SUM <= v_sum(7 downto 0);

      ----------------------------------------------------(1)
      if (v_sum(7 downto 0) = X"00") then
         Z <= '1';
      else
         Z <= '0';
      end if;

      ----------------------------------------------------(2)
      V <= '0';
      if (A(7) = B(7)) then
         if (v_sum(7) /= A(7)) then
            V <= '1';
         end if;
      else
      end if;

   end process;
end my_adder;
```

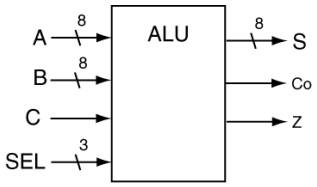
Figure 35.19: The entire ALU (8-bit form) for this section.

35.5 ALU Design using VHDL Modeling

After dragging you through this chapter, we ready to crank out some ALU examples. The approach we'll take utilizes the full power of VHDL; we'll thus be designing ALUs on the highest level possible. While we could design both arithmetic and logic units, we'll opt to focus our designs at as high of a level of abstraction as possible.

Example 35-3

Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out, and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



SEL	S	Co	Comment
"000"	A + B	valid	Addition
"001"	A-B	valid	Subtraction
"010"	A + B + C	valid	Addition with C
"011"	B + B	valid	2X multiplication
"100"	A + A	valid	2X multiplication
"101"	A XNOR B	-	Logic: XNOR
"110"	A NAND B	-	Logic: NAND
"111"	A AND B	-	Logic: AND

Solution: There is not much more to say about this solution that has not already been said. As you'll see from looking at the solution in Figure 35.20, we've previously described most of the VHDL usage in this model. There are a few things to note, and here they are:

(1) the Z and Co outputs are treated as signals. We give these values and overwrite the values later in the process when necessary.

(2) This is subtraction in VHDL. For this operation, we do not include the C input which would have been interpreted used for extended the operation greater than 8-bits. The Co in this case is considered a "borrow" and has all the attributes associated with a subtraction operation.

(3) This is an addition with the C, which is considered a carry-in from another calculation. This style of coding is typically used to extend the addition to something greater than 8-bits.

(4) This is the assignment of the Z indicator. Note that is was previously assigned earlier in the process; if the 8-bit result of the ALU operation is zero, this code reassigns the Z signal.

(5) This assigns the lowest eight significant bits of the intermediate result of the v_res variable to the output.

```
entity alu1 is
   port (   A,B : in std_logic_vector(7 downto 0);
              C : in std_logic;
            SEL : in std_logic_vector(2 downto 0);
          Z,Co : out std_logic;
              S : out std_logic_vector(7 downto 0));
end alu1;

architecture my_alu1 of alu1 is
begin
   process(A,B,C,SEL)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      ------------------------------------------------(1)
      Z <= '0'; Co <= '0';

      case sel is
         when "000" =>
            v_res := ('0' & A) + ('0' & B);
            Co <= v_res(8);
         when "001" =>
            ------------------------------------------(2)
            v_res := ('0' & A) - ('0' & B);
            Co <= v_res(8);
         when "010" =>
            ------------------------------------------(3)
            v_res := ('0' & A) + ('0' & B) + C;
            Co <= v_res(8);
         when "011" =>
            v_res := ('0' & A) + ('0' & A);
            Co <= v_res(8);
         when "100" =>
            v_res := ('0' & B) + ('0' & B);
            Co <= v_res(8);
         when "101" =>
            v_res := A XNOR B;
         when "110" =>
            v_res := A NAND B;
         when "111" =>
            v_res := A AND B;
         when others => v_res := (others => '1');
      end case;

      ------------------------------------------------(4)
      if (v_res(7 downto 0) = X"00") then
          Z <= '1';
      end if;

      ------------------------------------------------(5)
      S <= v_res(7 downto 0);

   end process;

end my_alu1;
```

Figure 35.20: The VHDL model solving Example 35-3.

Chapter Summary

- A computer is a device that reads instructions from memory and executes those instructions on associated data. The three main components of a computer are the memory, input/output, the central processing unit (CPU). The CPU is comprised of a control unit and a datapath; the main component of the datapath is the arithmetic logic unit (ALU).

- The term ALU can mean just about anything; ALUs are not constrained to performing only arithmetic and logic operations. ALUs can be designed on many different levels of abstraction.

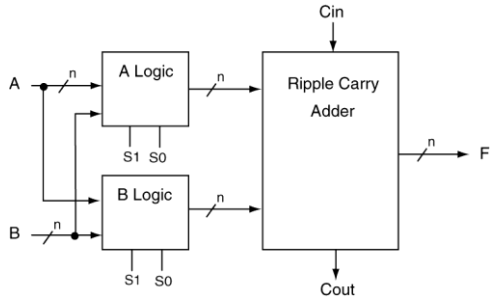
- VHDL support the use of both “signals” and “variables”. Signals are declared only in the declarative regions of architectures while variables are declared only in the declarative regions of processes. Signals can be seen in all processes of an architecture while variables are only visible in the process in which they are declared. Results from signal assignment in processes are not available until the process suspends while the results from variable assignments are available immediately.

- VHDL has many overloaded operators. High-level ALU design typically utilizes the overloading of mathematical operators and the extensive use of variables in modeling of ALUs.

Chapter Exercises

1) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

- Fill in the empty entries in the table below (fill in both equations and names of operations).
- Draw circuits that can be used to implement the "A Logic" and "B Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.

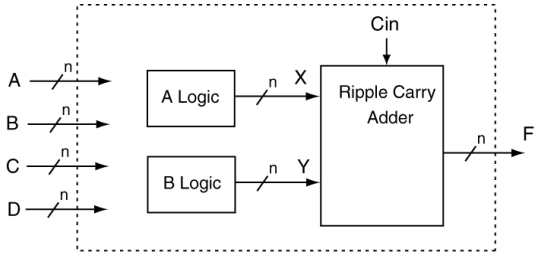


S1	S2	Cin = 0	Cin = 1
		equation	**equation**
		operation	**operation**
0	0	2A	2A + 1
			(??)
0	1	B + $\overline{B}$	
			clear (all 0's)
1	0		
		complement A	negate A
1	1	$\overline{B}$	
			negate B

2) The circuit below is a partially completed block diagram of an arithmetic circuit. The table below lists the operations required by the circuit. For this problem do the following:

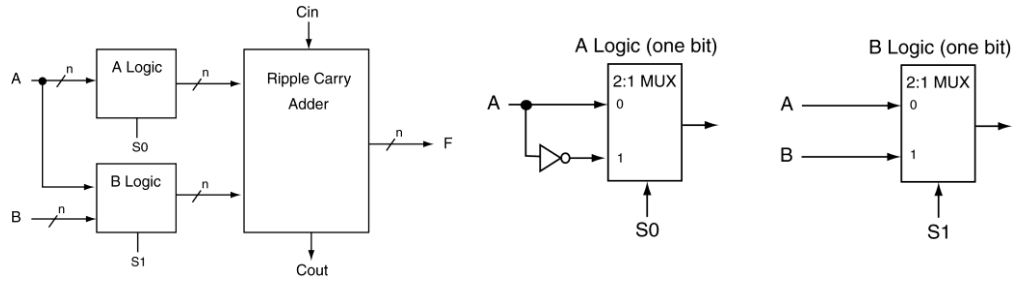
- Using only MUXes, design a bit-level implementation of the **A Logic** and **B Logic** that will implement the mathematical operations listed below. Your entire design should use no more than ***four*** control signals in addition to the **Cin** signal. Be sure to completely label your MUX control signals.

- Complete the table listed below by providing the control signals that your design uses to provide the listed operations.



Operation:	Required Control Signal Values
A - B	
A + B	
C - D	
C + D	
increment A	
decrement B	

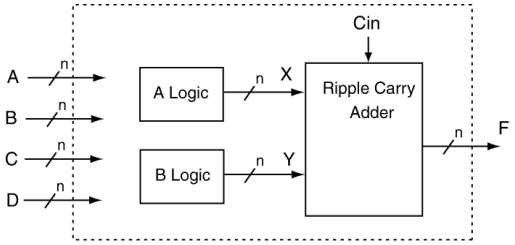
3) The figure on the left represents a block diagram of an arithmetic circuit. The diagrams on the right show the single bit versions of the "A logic" and "B logic" blocks from the diagram on the left. List all possible operations that this circuit can perform. Provide names to the arithmetic operations that make sense (such as addition, subtraction, set, clear, increment, etc.).



4) The circuit below is a partially completed block diagram of an arithmetic circuit. The table below lists the operations required by the circuit. For this problem do the following:

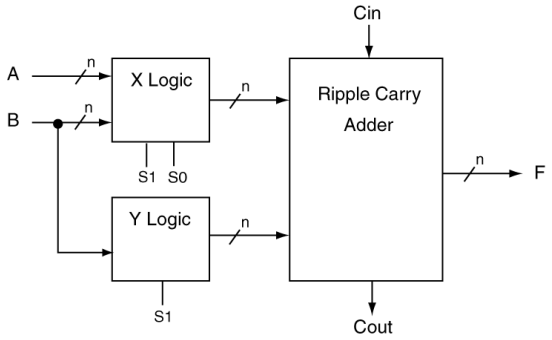
- Using only MUXes, design a bit-level implementation of the **A Logic** and **B Logic** that will implement the mathematical operations listed below. Your entire design should use no more than ***four*** control signals in addition to the **Cin** signal. Be sure to completely label your MUX control signals.

- Complete the table listed below by providing the control signals that your design uses to provide the listed operations.



Operation:	Required Control Signal Values
A - B	
A + B	
C - D	
C + D	
increment A	
decrement B	
transfer A	

5) The figure on the left represents a block diagram of an arithmetic circuit. The table on the right lists the operations that can be implemented using the arithmetic circuit. Show the logic equations or circuit implementations for a single bit for the "X Logic" and "Y Logic" blocks that implements the functionality described by the table on the right.

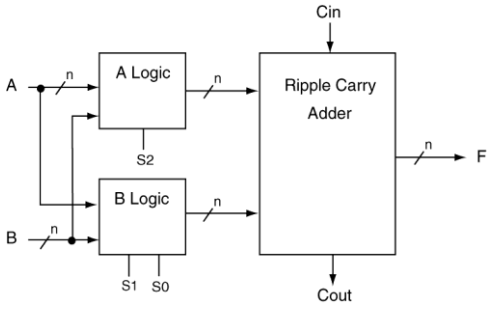


S1	S0	Cin = 0	Cin = 1
0	0	$\overline{B}+B$	0
0	1	$\overline{B}+A$	A - B
1	0	$B+A$	$B+A+1$
1	1	B - 1	B

6) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

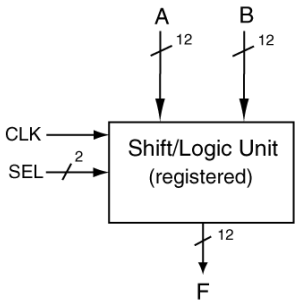
- Fill in the empty entries in the table below (fill in both equations and names of operations). *HINT: complete this bullet first; also, the two right-most columns differ by only the Cin value*.

- Draw circuits that can be used to implement the "A Logic" and "B Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.



S2	S1	S0	Cin = 0	Cin = 1
			equation	**equation**
			operation	**operation**
0	0	0	A + $\overline{B}$	
			(??)	
0	0	1	A + 0	
0	1	0		
			(decrement A)	
0	1	1		A + A + 1
				(??)
1	0	0	B + $\overline{B}$	
1	0	1		
				(increment B)
1	1	0	B - 1	
1	1	1		B + A + 1
				(??)

7) Provide a VHDL description (the architecture) of the Shift/Logic unit described below. For this problem, ignore the CLK signal. Don't use mathematical operators in your VHDL model.

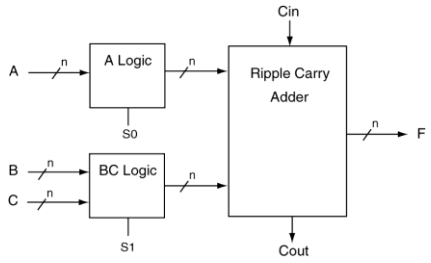


Shift/Logic Unit Specification

SEL(1)	SEL(0)	Out	Comment
0	0	$\overline{A}$	compliment A
0	1	A NOR B	bitwise NOR operation
1	0	A brl 5x	**A** barrel rotate left (5 bits)
1	1	B ÷ 16	**B** divided by 16

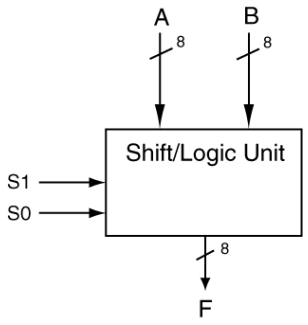
8) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

- Fill in the empty entries in the table below (fill in both equations and names of operations).
- Draw circuits that can be used to implement the "A Logic" and "BC Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.



S1	S0	Cin = 0	Cin = 1
		equation	**equation**
		operation	**operation**
0	0		
0	1		
		(decrement B)	
1	0		A + C + 1
1	1		
		(decrement C)	

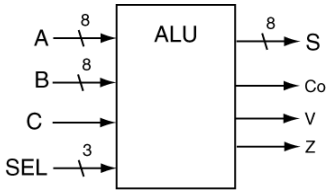
9) Provide a VHDL description of the Shift/Logic unit described below.



Shift/Logic Unit Specification

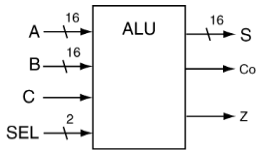
S1	S0	Out	Comment
0	0	A AND B	bitwise AND
0	1	A XOR B	bitwise exclusive OR
1	0	A asl (r-0)	A input arithmetic shift left (0 in right)
1	1	B bsr3x (l-0)	B input barrel shift right 3x (0 in left)

10) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out, the V is an overflow indicator, and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



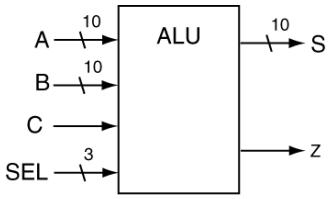
SEL	S	Co	V	Comment
"000"	A + B + C	valid	valid	Add with C
"001"	A-B-C	valid	valid	Subtract with C
"010"	B + B	valid	valid	2x B
"011"	A+ 1	valid	-	Increment A
"100"	B + 1	valid	-	Increment B
"101"	Zero	-	-	All zeros
"110"	-1	1	1	2's comp -1
"111"	A NAND B	-	-	Logic: NAND

11) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



SEL	S	Co	Comment
"00"	|A| + 1	valid	Increment the absolute value of A
"01"	|B| -1	valid	Decrement the absolute value of B
"10"	|A + B|	valid	Absolute value of (A + B)
"11"	A AND B	-	Logic: A AND B

12) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as necessary. The S output contains the result of the selected operation. The Z is a zero indicator and is active for all operations.



SEL	S	Comment
"000"	-A	Negate A
"001"	-B	Negate B
"010"	|A|	Absolute value of (A)
"011"	|B|	Absolute value of (B)
"100"	0	zero
"101"	"CCCCCCCCCC"	Propagate C input
"110"	A/2	Half A (truncate)
"111"	A OR B	Logic: AND

99 Chapter Ninety-Nine

(Bryan Mealy 2012 ©)

99.1 Chapter Overview

If we were all perfect, there would be no need for this chapter. However, since we all generally make mistakes, we definitely need a mechanism to find and correct those mistakes. Out there in digital design-land, the approach we take to creating digital circuits is 1) design them, 2) simulate them, and 3) implement them. The ordering of these steps is massively important despite the fact that most people do step 3) before doing step 2)[1], if they even do step 2) at all.

You should simulate every circuit you design in order to increase your confidence level that the circuit is working properly. Life is easy with simple circuits; you can probably survive without testing them. More complex digital circuits inherently contain nuances so that you simply can't assume they will work without somehow properly verifying that fact. The main goal of this chapter is to get you started simulating your own circuits. This chapter presents the basic tools and theory of writing "testbenches", which is the term VHDL uses to describe a VHDL-based simulation mechanism for your circuit. This chapter is not an exhaustive approach to writing testbenches; it only aims to get you started. Once you get started, your circuit simulation techniques will quickly go far beyond the drivel in this chapter.

We opted to use a high chapter number for this chapter because there seemed to be no optimal location to present this material. If you don't want or need to test your VHDL models, you'll have no need for this chapter. But, any good digital designer knows that testing is an important part of designing digital circuits[2]. Additionally, you don't need to read this entire chapter; use what you need when you need it.

Main Chapter Topics

- **VHDL TESTBENCHES:** VHDL uses the notion of testbenches to verify circuit operation. This chapter presents an overview and introduction to VHDL testbenches.
- **TESTBENCH TEST VECTORS:** This chapter describes the options VHDL testbenches can use to generate and/or access data used by testbenches.
- **VHDL ASSERT STATEMENT:** This chapter describes how testbenches use assert statements to help verify proper circuit operation..
- **VHDL PROCESS STATEMENTS:** This chapter describes another form of the VHDL process statements; this new form is useful in testbench models.
- **VHDL WAIT STATEMENTS:** This chapter describes the four types of VHDL wait statements and provides examples of their usage in actual testbench models.

[1] We're all busy. Us, we're truly busy and have the results to prove it. Academic administrators: they do their best to look busy and violently wave their hands in lieu of actual results.

[2] If the circuits you design don't work, you may as well become an academic administrator instead. For such positions, no experience, expertise, not intelligence of any type is required.

Why This Chapter is Important

This chapter is important because it provides an overview and introduction to writing testbenches in VHDL. The VHDL language uses testbenches as a mechanism for verifying the proper operation of VHDL models using none other than other VHDL models.

99.2 Testbench Overview: VHDL's Approach to Circuit Simulation

Most of your VHDL career up to this point was focused on designing circuits that were intended to be synthesized, which is one of the powerful points of VHDL. However, keep in mind that one of the other powerful characteristics of VHDL is the ability to design models that can test other VHDL models. In other words, VHDL is such a versatile modeling tool that it can also act as a simulation mechanism. Unfortunately, in many instances, the main focus of digital design using VHDL is the design and generation of circuits; the testing/verification portion of circuit design is attenuated due primarily to time constraints.

Because testbenches are an important part of VHDL modeling, they are also an important part of the modern digital design process. In the initial stages of learning digital design, you're designs were most like simple enough so that you could verify their correct operation by examining the circuit models or testing the final circuit implemented on some type of real hardware. In all likelihood, you may not have even simulated your circuit. This is all fine, but the non-testing approach quickly breaks as your digital circuit become more complex.

As your digital designs become more complex, you're going to need to simulate them as part of the design process. In this context, simulation serves two purposes. First, simulation is going to be a great design tool. If you haven't realized it already, digital circuits can become complex. The complexity increases further when you are designing with non-ideal devices and you're forced to deal with the propagation delays associated with physical devices[3]. Secondly, simulation is a great debugging tool. If your circuit is not working and it's not obvious why, you'll know it's time to simulate. If you're truly doing the digital design thing correctly (and your designs are not trivial), you should be finding yourself spending as much time writing simulation models as you spend writing the hardware models themselves.

VHDL uses the term "testbench" to describe the mechanism VHDL uses to verify the functional correctness of your VHDL models. This chapter provides a vehicle to get you started writing testbenches, and thus allows you to verify the correct operation of your hardware models.

Finally, in the real world, the up-front verification of circuit operation is critical to the success of any project. As you know, the earlier you catch errors, the easier they are to fix and they'll have less tendency to generate more errors and induce bad design decisions along the way. This is massively important in the case of custom ASIC design when obtaining an actual design on silicon is going cost you about a million bucks[4]. In the end, if you play your cards right, you'll be using simulation as your

[3] The underlying thought here is that many digital circuits need to operate as fast as possible. In this case, many factors rear their ugly heads and conspire to undermine the proper operation of your circuit. All digital circuits stop functioning properly at some speed. Often times the goal in digital design is to push your circuit to operate as fast as possible. In many cases, making your circuit operate faster is a primary design constraint and will necessarily force you to redesign your circuit in order to meet required time constraints.

[4] Which is why prototyping with PLDs is a powerful alternative to paying for a custom ASIC.

primary design tool. Keep in mind that the original use of VHDL was as a tool to allow you to model and simulate digital designs.

99.3 Testbenches: VHDL's Approach to Circuit Simulation

A testbench can mean many different things. For this chapter, we'll consider a testbench to be a VHDL model that is separate from your VHDL circuit model. The testbench works in conjunction with the VHDL model with the purpose of verify proper operation of the VHDL model. The major difference between the testbench and the circuit you're testing is the fact that you probably intend to synthesize your circuit model while your testbench model is probably non-synthesizable. As you'll soon find out, the testbench models typically use VHDL constructs that don't synthesize. This is because the primary purpose of the testbench is to provide a set of "stimulus" to the model you're testing.

The testbenches we'll examine are written in VHDL. While this is not a requirement, there are many advantages to writing testbenches in the same language used to model the circuit. The main advantages are that you won't need to learn a new tool or a new language[5], and the entity used to test your circuit can be included with and tested using the same tools you used to model your circuit.

Testbenches are a deep subject; there are many approaches to testbench design as result of flexibility of the VHDL language. This chapter will get you started on using testbenches to verify your design. As you continue your digital design journey, necessity will force you to learn many more circuit verification techniques not presented in this chapter.

99.4 The Basic Testbench Models

VHDL test benches can span the gamut from quite simple to massively complex depending on the intended purpose of the design. Often times, the testbench model can become more complicated than the actual circuit you're testing. The result is that there are many different "models" associated with testing a circuit with a testbench. This section provides a quick overview of some of the more popular models, which are the ones we'll crank though later in this chapter. This is the quick overview part, provided to give you a quick feel for testbenches. The low-level details arrive later.

Figure 99.1 shows the most basic testbench model. This model comprises of two main components: the "stimulus driver" and the "design under test" (DUT)[6]. These two boxes are typically referred to by many different names but the functions are still the same (so don't become too hung up on the names). The DUT is the VHDL model you're intending to test; the stimulus driver is a VHDL model that communicates with the DUT by providing it with inputs to exercise the DUT. Here are a few important things to note regarding Figure 99.1.

- There is no magic in Figure 99.1. The truth is that the model depends on some human generating all the details of the "stimulus driver" box. You the human and you the designer of the circuit will need to decide what to test and determine if every important part of your circuit has been exercised by the stimulus driver enough for you to say, "yes, this circuit works"[7].

[5] This is not completely true; you'll soon find out that we'll be using several features in VHDL that we have not previously used.

[6] The DUT is sometimes referred to as a "UUT" which stands for "unit under test". Other times it is referred to as the "MUT", or model under test. Sometimes it is referred to as a "BBUT", or bowling ball under test. The stimulus driver is sometimes referred to as the waveform generator.

[7] Where as an academic administrator would say, "yes, this is a circuit" as these people are deathly afraid of committing to anything (unless of course they can blame their failures on innocent people).

- The dotted box labeled "testbench" represents the testbench model in terms of the VHDL language. Note that the dotted box has no inputs or output; thus, the VHDL entity will have no inputs or outputs either. This is strange, but you'll quickly get used to it.

- The testbench is really testing something. The model listed in Figure 99.1 will generally be used to generate a timing diagram. Using this model, the timing diagram will then need a visual inspection from some human in order to verify the circuit is working properly.

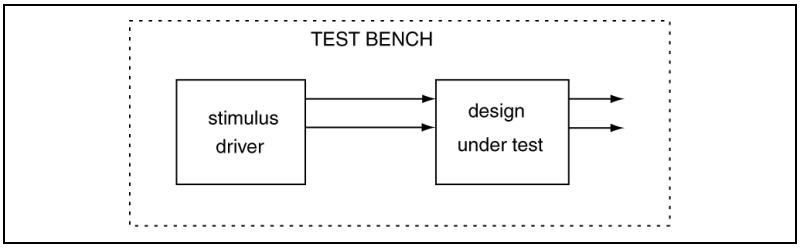


Figure 99.1: A block diagram for a basic VHDL testbench.

Figure 99.2 shows a slightly modified model of Figure 99.1. In many testbenches, the DUT sends feedback and/or intermediate results back to the stimulus driver. The stimulus driver can use this feedback as part of the testing (such as verifying the correctness of intermediate results) or sequencing of the testbench (waiting for a status signal indicating that some portion of the DUT is ready to be tested). The models of Figure 99.1 and Figure 99.2 are quite simple are relatively common.

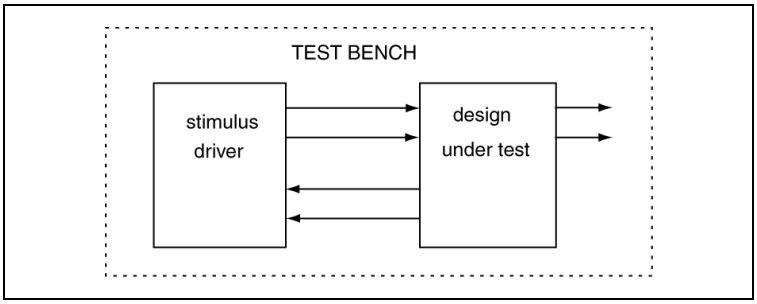


Figure 99.2: A block diagram for a more complicated VHDL testbench.

Figure 99.3 shows an extension of Figure 99.2. This figure shows that you can design your testbenches an extremely useful feature. As was present in Figure 99.2, the stimulus driver can contain pre-determined values that the model can use to verify the correctness of intermediate, or "expected" results. In this case, this set of predetermined results can determine whether the DUT passed the "test" or not. The notion here is that if all the results comparisons are happy, then the DUT passes the test. There are many methods you can use to implement the "results comparison" box of Figure 99.3.

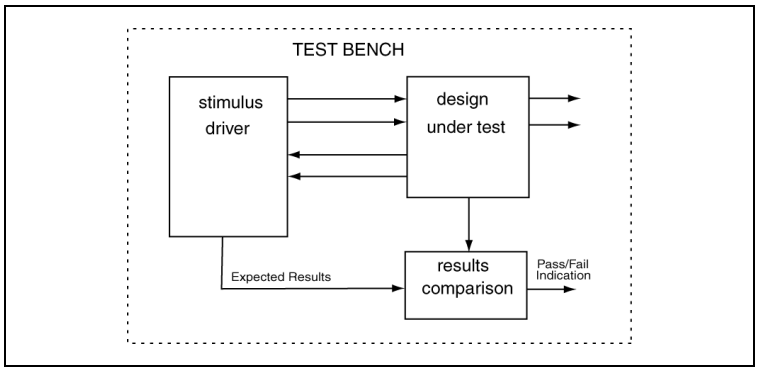


Figure 99.3: A block diagram for yet another VHDL testbench.

Figure 99.4 shows one final model we'll work with in this chapter. This model shows that the testbench can use external files for either storing the test vectors or writing the results of the test. Simple circuits don't require fancy testbenches, but as circuits become more complicated, other more clever approaches can be used to verify correct circuit operation.

One interesting thing to keep in mind here is that the external files for test vectors can be generated by some other test device such as the output of some other simulator you're using to test your design. Many times designs are first done on paper or modeled with computer programs; these devices can output test vectors more easily and completely than a human can do. VHDL has many facilities for accessing data from external files to use as test vectors[8].

As you know from previous models, the testbench can determine whether the DUT has passed the testing procedures or not. While a pass/fail indication is nice, it does not provided much information. When your circuit is large and complex, you'll surely want the testbench to give you information along the way, particularly if you design does not pass the testing procedures. Once again, VHDL has many options for outputting information to files. These files can then be compared directly to files generated by other test mechanisms to determine whether the DUT is working properly.

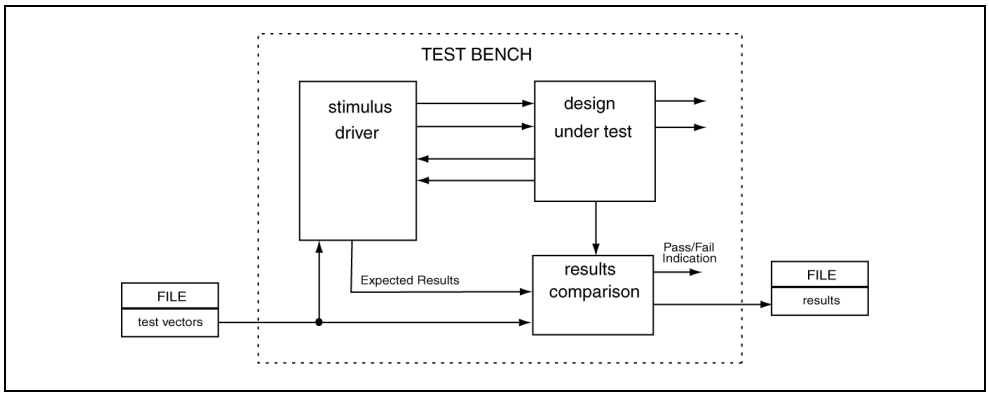


Figure 99.4: A block diagram for a VHDL testbench that reads test vectors from external files and writes the results to other external files.

[8] This chapter only describes a few of them.

VHDL has many facilities you can use to test your design. This chapter presents some of the basics regarding the use of testbenches for circuit verification. There is much more to the story, so if you find yourself being required to write amazing test benches, then you'll need to learn more or all of the details this chapter has opted to omit. VHDL is powerful, particularly in the context of circuit verification: learn what you need to as the need arises.

99.5 The Stimulus Driver

The heart of the testbench is the stimulus driver. This black box represents a majority of the code you'll be writing for your testbench. Being as important as it is, this section provides a short overview of possible structures for the stimulus driver. The basic structure of a testbench is simple but the issues are in the implementation details, which can seem daunting for anyone new to writing testbenches. Some of the details in this section were "hinted at" in the previous section in the provided black box testbench diagrams. This section paves the way for the examples we present later in this chapter.

99.5.1 The Stimulus Driver Overview

You have two possibilities regarding the verification of a circuit. The stimulus driver of course generates the testing waveforms, but whose job is it to actually verify your circuit is working? Verification of the proper operation of a circuit can be either manual or automatic. The two main ideas here is whether you want a human to examine the resulting output waveforms from a testbench in order to verify proper operation (manual) or whether you want the testbench to automatically state whether the circuit is working properly. Here is an overview of what is good and bad about each approach.

> Manual Verification: This is the simplest approach and is thus the approach taken by most beginning testbench writers. The testbench models shown in Figure 99.1 and Figure 99.2 are examples of manual circuit verification. The models in these figures only generate the test vectors for the DUT, which inherently produce some output signals. The notion here is that the human reader will need to examine the resulting waveforms in order to state whether the circuit is operating properly. This is a great option for beginners, but less great of an option if you have a complex circuit you need to test the crap out of[9].
>
> Automatic Verification: As your digital designs become more complex, you'll want to avoid manual verification. In truth, VHDL has many constructs that assist in the creation of testbenches that use automatic verification; this chapter presents only a few of them[10]. The testbench models in Figure 99.3 and Figure 99.4 show examples of automatic verification with the inclusion of the black box labeled "results comparison". There are many ways use VHDL code to structure the "results comparison" box; the examples in this chapter describe a few.

99.5.2 Vector Generation Possibilities

VHDL provides several approaches for your stimulus driver to supply test vectors to your DUT. More specifically, the list below describes the three main approaches to generating test vectors. Note that in any given testbench, you can you any combination of these three approaches.

[9] Keep in mind that as digital circuits become more complex, you're going to need to write more complex testbenches in order to be 100% certain that they are working properly.
[10] If you continue on in VHDL, you'll gather many more testbench writing skills. This chapter only aims to give you a quick taste of the possibilities.

Test Vectors generated "On The Fly": These vectors include any vectors that are not stored in internal VHDL structures (such as arrays) or stored in external files. This form of test vector generation works best for simple circuits or circuits that do not require extensive testing or complicated test vectors. In terms of simple stimulus drivers, you include whatever you need to in your VHDL code, but this approach quickly becomes unwieldy for large testbenches.

Test Vectors read from VHDL arrays: The VHDL language contains "arrays" which can be used to store constants. One approach to writing stimulus drivers is to store the test vectors as constants inside of arrays and access those constants using VHDL. The good part about this approach is that the test vectors will always be included with the testbench model (as opposed to reading test vectors from a file).

Test Vectors read from files: As with other computer languages, VHDL as the ability to read from and write to files. One of the main uses of this mechanism is storing testing information in external file. These external files include files where the test vectors will be read from and external files where the results will be written. The good part about this approach is that the test vectors can easily be generated from other computer programs such as a spreadsheet, some other type of simulator, or a higher-level language program you may be using to verify your circuit models.

99.5.3 Results Comparisons: The "assert" Statement.

The "assert" statement is both common and useful in the circuit verification using VHDL testbenches. The nice thing about assert statements is that they are simple to use and understand. There are many ways to verify the proper outputs from an DUT; using assert statements is one of those ways. An assert statement simply checks the Boolean value returned from the evaluation of the expression associated with the assert statement. If the expression evaluates as true, nothing happens. If the expression evaluates as false, the testbench provides information regarding of what went wrong.

Figure 99.5 shows an example of an assert statement. We'll see statements used in actual testbenches in a later section; this statement is shows the basic syntax of the statement. Here are a few important things to note about the assert statement shown in Figure 99.5.

- The expression in the parenthesis returns a Boolean value. If this expression is true, nothing happens. If this expression evaluates as false, the items associated with the "report" and "severity" statements occur. The things that happen are the printing of the text associated with the "report" and "severity" lines to somewhere or something, which is typically the console window associated with the simulator exercising the testbench[11].

- The "report" and "severity" lines are associated with an assert statement. You can include one of both of these options in your testbench. The "severity" line output one of four piece of text: "Note", "Warning", "Error", or "Failure". The human writing the testbench chooses the appropriate text for the "severity" line. The "report" line prints out a user-specified message.

[11] Depending on your particular simulator, other information is also include as part of assert statement failures.

```
assert (s_signal = '1')
   report "s_signal does != 1; do something!
   severity Warning
```

Figure 99.5: A quick overview of the four main types of "wait" statements.

There is more to say about assert statements but we'll only mention one final notion. Although there is only one form of the assert statement, they are slightly different based on where in the testbench code they can appear. Assert statements can appear as concurrent statements or as sequential statements. If the assert statement appears as part of a process statement, it is a sequential statement; otherwise, it is a concurrent statement. The difference here is that assert statements as part of process execute when they are encountered as execution travels through the other sequential statements in the process. Concurrent assert statements execute anytime there is a change in the any of the signals present in the expression of the assert statement. In this way, the concurrent assert statement acts like a sensitivity list for a process statement. Though these difference seem subtle, they are actually quite significant.

99.6 The Process Statement: A Re-Visitation

Process statements: you've seen them before, but in only one form. Here's what you know about a process statement: when a signal in the sensitivity list of the process changes, the process "executes"[12]. When a process statement executes, it commences stepping through the sequential statement contained in the statement. When all of the statements in the process have completed execution, the process terminates. Another way of looking at this is that activity on a signal in the sensitivity list wakes up the process; the process then executes its statements until it reaches the ends of the process, then the process goes back to sleep (and waits for more action on a signal in the sensitivity list)[13].

You've probably written about a bajillion process statements by this time in your digital design career. That truth is that all of the examples in this text used only one of two major forms of process statements. The form we've been using contains a sensitivity list to indicate which signals are important to the process. When the value of one of the signals on the sensitivity list changed, the process executes. This is the most common form of a process statement and is the form that you should use when you intend on synthesizing your circuit. The other form of a process statement does not contain a sensitivity list. As you may guess, this form is less useful than the other form for modeling circuits. This other process form, however, is quite useful when your VHDL code is modeling a testbench, so we'll describe it here in detail.

As a quick example, Figure 99.6 shows a D flip-flop modeled both with and without a sensitivity list. The process in Figure 99.6(a) activates and executes when there is a change on either the CLK or D signal. The process complete execution, deactivates, and waits for more changes on CLK or D. The process in Figure 99.6(b) does not start execution until the wait statement is satisfied; when the rising edge of the clock occurs, the process activates, executes, suspends executions, and waits for the next rising clock edge. As you can see, it's not that big of a deal.

[12] I really don't like the word executes because it sounds way too much like you're describing the operation of a piece of computer code. VHDL, on the other hand, describes the operation of a piece of hardware.

[13] This is knowingly a tough concept. I grapple with it constantly. I never quite understand it; I simply accept it. The notion of having sequential statements within a process statement and the fact that a process statement is a concurrent statement always thumps my brain. And in the end, the model can be used to generate hardware? This is sometime too much for my little brain.

```
----------------------------------------
-- D flip-flop using a sensitivity list
----------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_nowait of DFF is
begin
   process (CLK,D)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process;
end DFF_nowait;
```

(a)

```
----------------------------------------
-- D flip-flop using a sensitivity list
----------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait of DFF is
begin
   process
   begin
      wait until (rising_edge(CLK));
         Q <= D;
   end process;
end DFF_wait;
```

(b)

Figure 99.6: An example of a D flip-flop described without (a) and with (b) a wait statement.

The two forms of process statements are quite distinct. If your process statement does not include a sensitivity list, then it must include at least one wait statement[14]. If your process statement does include a sensitivity list, then the body of your process cannot include a wait statement. The sensitivity list of a process controls when the process will activate. In particular, when one of the signals on the sensitivity list changes, the process activates and the entire process executes from beginning to end. The notion of "from beginning to end" means that there is nothing to stop the process along the way. This is great for modeling circuits, but not so great for testing VHDL models.

On the other hand, process statements without sensitivity lists rely on "wait statements" to control the activation and deactivation of a process. Process statements using wait statements start and stop under control of the wait statements. The effect of this is that the process is either not executing (because it has been deactivated because of a wait statement") or the process is executing (because a wait statement condition was met and the process is "not waiting").

99.7 Attack of the Killer Wait Statements

This section covers the four forms of wait statements. Testbenches certainly don't always use all of these forms, but we include them here for both completeness and just in case you feel you need to use one of these forms. The following subsections provide a basic description and example of wait statements. Keep in mind that the basic function of a wait statement is to give you the ability to suspend execution of a process at any point in the process. This differs from statements using sensitivity lists in that once they start executing, they cannot be suspended until the end of the process is reached.

The four forms of a wait statement are straightforward in that the process is always "waiting" for something. Figure 99.7 shows the four things that can be "waited for"; the following sections expand on these basic wait statement types.

[14] There are several forms of wait statements; we'll get to those soon.

1. a change in a signal (WAIT ON)
2. a expression is true (WAIT UNTIL)
3. a specific amount of time (WAIT FOR)
4. an eternity (WAIT)

Figure 99.7: A quick overview of the four main types of "wait" statements.

99.7.1 The "wait on" Statement

The "wait on" statement is the most similar wait statement to a process sensitivity list. The "wait on" statement is waiting for a change in the list of signal in the wait on statement. When one of the signals in the wait statement list changes, the process resumes executing with the statement following the "wait on" statement.

Figure 99.8 shows an example of D flip-flop model using a "wait on" statement. One thing to keep in mind is that process execution is circular in nature, meaning that when the process reaches the end, it automatically continues execution at the beginning of the process. The model in Figure 99.8 "waits" for changes in either the CLK or the D signal; when such a change occurs, the process restarts execution at the if statement.

Figure 99.8 is for example purposes only. It is not even close to being a good way to model a D flip-flop. We'll see better uses for "wait on" statements when we start back looking at testbenches.

```
-------------------------------------------------------
-- D flip-flop using a "wait on" statement
-------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait_on of DFF is
begin
   process
   begin

      if (rising_edge(CLK)) then
         Q <= D;
      end if;

      wait on CLK,D;

   end process;
end DFF_wait_on;
```

Figure 99.8: An example of a D flip-flop modeled using a "wait on" statement.

99.7.2 The "wait until" Statement

The "wait until" form of a wait statement instructs a process to suspend execution until the evaluation of the expression associated with the "wait until" statement returns a value of true. The catch here is that the "wait until" statement requires that the associated expression return a boolean value in order for the statement to be valid.

Figure 99.9 shows an example of D flip-flop model using a "wait until" statement. This process chooses to expand the "rising_edge" function to show that the "wait until" statement really does evaluate an expression. The line in the process body of shown in Figure 99.9 that is commented out is also a valid statement as the "rising_edge" function returns a boolean value. The model in Figure 99.9 "waits" for two conditions to simultaneously occur: a change the clock signal (the CLK'EVENT[15]) and the current value of the CLK signal to be a '1'.

Once again, the D flip-flop model in Figure 99.9 is for example purposes only as it is not a good way to model a D flip-flop. We'll see better uses for "wait until" statements later in this chapter.

```
-------------------------------------------------------
-- D flip-flop using a "wait until" statement
-------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait_until of DFF is
begin
   process
   begin

      --wait until (rising_edge(CLK));

      wait until (CLK = '1' and CLK'EVENT);
         Q <= D;

   end process;

end DFF_wait_until;
```

Figure 99.9: An example of a D flip-flop modeled using a "wait until" statement.

99.7.3 The "wait for" Statement

The notion of a "wait for" statement means the associated process is forced to suspend execution and "wait" for the amount of time given in the "wait statement" argument. Once the stated amount of time has passed, the process resumes execution. Figure 99.10 shows the syntax for a "wait for" statement as well as a few examples.

[15] This is referred to as a "tick event"; this text did not cover this aspect of VHDL.

```
wait for time_expression
```

Examples:

```
-- using a value
wait for 25ns;

-- using a constant
wait for (CLOCK_PERIOD);

-- using an expression
wait for (CLOCK_PERIOD * 60);
```

Figure 99.10: The syntax and examples of a "wait for" statement.

The problem is that I don't have a great example for it. In lieu of a great example, we'll use a crappy example instead. Figure 99.11 shows the crappy example I speak of. The problem with this example is that the model will not synthesize as you expect most models to. The reason I included this example is to highlight the differences between models intended for synthesis and models intended for simulation (testbenches). The model in Figure 99.11 is trying to be a D flip-flop but will not synthesize due to the "wait for" statement.

```
------------------------------------------------------------
-- D flip-flop using a "wait for" statement
-- WARNING: this model does not synthesize
------------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
                Q : out std_logic);
end DFF;

architecture DFF_wait_for of DFF is
begin
   process
   begin

      -- this does not work!
      wait for 10ns;

      if (rising_edge(CLK)) then
         Q <= D;
      end if;

   end process;
end DFF_wait_for;
```

Figure 99.11: An example of a D flip-flop model attempting to use a "wait for" statement.

99.7.4 The "wait" Statement

The "wait" statement is the simplest of the wait-flavored VHDL statements. Testbenches use this statement in order to end a process. Consequently the "wait" statement (with no arguments) is used to

end the simulation process. Once again, we'll see examples of this when we start introducing some actual testbenches. Figure 99.12 shows the syntax of a "wait" statement; we include no examples.

```
wait;
```

Figure 99.12: The syntax for a "wait" statement.

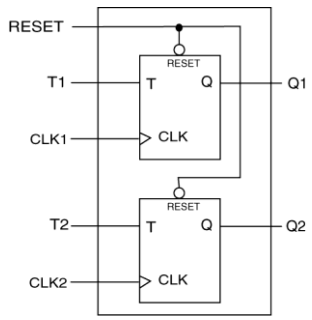
99.8 Finally, Getting Your Feet Wet: Some Example Testbenches

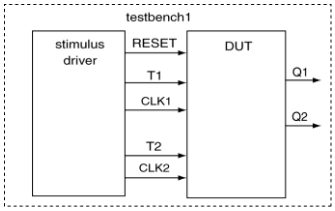
This section details some of the finer aspects of writing testbenches. This section by no means provides every possible testbench, but there should be enough information to get you started.

Example 99-1: The First Testbench

Write a testbench that models the circuit shown in the top portion of Figure 99.13(b). Figure 99.13(a) provides the VHDL model for this circuit.

```
entity tff_ckt is
   port (   CLK1,CLK2 : in std_logic;
          RESET,T1,T2 : in std_logic;
                Q1,Q2 : out std_logic);
end tff_ckt;

architecture my_ckt of tff_ckt is
   signal s_tff_Q1, s_tff_Q2 : std_logic; --:='0';
begin

   tff1 : process (RESET,CLK1,T1)
   begin
      if (RESET = '0') then
         s_tff_Q1 <= '0';
      else
          if (rising_edge(CLK1)) then
             s_tff_Q1 <= T1 XOR s_tff_Q1;
          end if;
      end if;
   end process;

   tff2 : process (RESET,CLK2,T2)
   begin
      if (RESET = '0') then
         s_tff_Q2 <= '0';
      else
         if (rising_edge(CLK2)) then
            s_tff_Q2 <= T2 XOR s_tff_Q2;
         end if;
      end if;
   end process;

   Q1 <= s_tff_Q1;
   Q2 <= s_tff_Q2;

end my_ckt;
```





(a) (b)

Figure 99.13: A pointless example circuit using T flip-flops; model (a) and block diagram (b).

Solution: Figure 99.14 shows the solution to Example 99-2. Have fun.

```
-------------------------------------------------------(1)
entity testbench1 is
end testbench1;

architecture stimulus of testbench1 is

   -- Misc declarations----------------------------------(2)
   constant CLK1_PERIOD: time := 40 ns;
   constant CLK2_PERIOD: time := 60 ns;

   -- declare DUT ---------------------------------------(3)
   component tff_ckt
      port (   CLK1,CLK2 : in std_logic;
             RESET,T1,T2 : in std_logic;
                   Q1,Q2 : out std_logic);
   end component;

   -- instantiate the device under test (DUT) ------------(4)
   signal s_t1, s_t2, s_q1, s_q2 : std_logic := '0';
   signal s_clk1, s_clk2 : std_logic := '0';
   signal s_reset : std_logic := '0';

begin
   -- instantiate the device under test (DUT) ------------(5)
   DUT: tff_ckt
   port map (CLK1 => s_clk1,
             CLK2 => s_clk2,
             RESET => s_reset,
             T1 => s_t1,
             T2 => s_t2,
             Q1 => s_q1,
             Q2 => s_q2);

   -- synthesize reset signal ----------------------------(6)
   s_reset <= '1', '0' after 20ns, '1' after 40 ns;

   -- set signal values ----------------------------------(7)
   s_t1 <= '1';    s_t2 <= '1';

   -- clk1 synthesis -------------------------------------(8)
   s_clk1 <= not s_clk1 after CLK1_PERIOD/2;

   -- clk2 synthesis -------------------------------------(9)
   clk2 : process
   begin
      wait for (CLK2_PERIOD * 0.75);
         s_clk2 <= '1';
      wait for (CLK2_PERIOD * 0.25);
         s_clk2 <= '0';
   end process;

end stimulus;
```

Figure 99.14: A block diagram for a basic VHDL testbench.

This is an instructive testbench model so we've included some fairly instructive comments. The following numbered items correspond to the numbers associated with the comment in Figure 99.14. Later testbench models will surely include less commentary (but probably not).

(1) This is the entity declaration; as advertised, there is no associated port clause as the testbench has no inputs and no outputs.

(2) This testbench uses some constants. As with programming using higher-level languages, the liberal use of constructs such as constant make your code more useful. This is particularly true when working with testbenches. If something changes, you don't want to search through a large VHDL model to make all the required modifications.

(3) This is the component declaration for the circuit that you intend on testing. Refer to the block diagrams in Figure 99.13(b) and realize that you've done this many times; it is no different when you're working with testbenches.

(4) This is a list of intermediate values used by the testbench. Note that these signals represent all of the signals shown in Figure 99.13(b). The important thing to notice here is that we assign initial values to all of the signals. This practice is almost a requirement, as you'll find out when you simulate circuits. If you do not assign these values, the simulator will not be smart enough to assign the values for you and you'll end up with unknown waveforms in your simulation results. If you assign these initial values later in your testbench model, these initialization values are overwritten.

(5) This is the instantiation of the device you're testing. Once again, this is nothing new and relation to the block diagrams in Figure 99.13(b).

(6) This is a concurrent statement used for a reset signal. This is a one-time statement so this implementation is probably the easiest approach. Note the statement is a list with commas (a VHDL feature we have not discussed). This statement uses the VHDL keyword "after" to provide the timing of this negative logic reset pulse. Note that the pulse is 20ns wide. This line executes three times during simulation: once to set the value, once to clear the value, and one more time to set the value again, which remains set throughout the simulation.

(7) This line contains two statements (in order to save space). These lines are concurrent statements that put the two outputs from the stimulus driver at known values. This line executes one time during the simulation.

(8) This is one approach to synthesizing a clock signal. This concurrent statement synthesizes a periodic clock signal (50% duty cycle) by toggling the signal every half CLK1 period. This concurrent statement is evaluated every half-CLK1 period throughout the simulation.

(9) This is another approach to synthesizing clock signals. This approach uses a process statement (without a sensitivity list) to generate the clock signal. This particular clock signal uses some special notation in order to provide CLK2 with a 25% duty cycle.

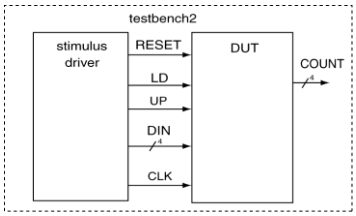
For a final comment, the output of this testbench is the Q1 and Q2 signals. Because of the associated T flip-flops are always asserted, they act to halve the frequency of the two clock signals. For a given simulation, you'll probably include all the available signals in your waveform output.

Example 99-2: The Second Testbench

Write the VHDL code that implements the testbench model shown in the black box diagram of Figure 99.15(b). For this circuit, consider the DUT to be a 4-bit up/down counter with a synchronous parallel load (as modeled by Figure 99.15(a)).

```
entity COUNT_4B is
   port ( RESET,CLK : in std_logic;
               LD,UP : in std_logic;
                 DIN : in std_logic_vector (3 downto 0);
               COUNT : out std_logic_vector (3 downto 0));
end COUNT_4B;

architecture my_count of COUNT_4B is
   signal  s_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
   begin
      if (RESET = '1') then
         s_cnt <= (others => '0');
      elsif (rising_edge(CLK)) then
         if (LD = '1') then     s_cnt <= DIN;
         else
            if (UP = '1') then  s_cnt <= s_cnt + 1;
            else                s_cnt <= s_cnt - 1;
            end if;
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

(a)



(b)

Figure 99.15: An example using a 4-bit counter; the VHDL model (a) and a diagram showing the testbench circuit. (b).

Solution: This example problem really did not ask much. We are using this problem as a stepping-stone for presenting meaningful testbench modeling concepts. Figure 99.16 shows an example testbench for this problem. Here are a few meaningful comments corresponding to the numbered comments of Figure 99.16.

(1) Although the intermediate signals were initialized when they were declared, this statement immediately changes them. The word "immediate" is important. In this context, it means the values change at time "zero" in the simulation. This is because the statements described by this comment are concurrent statements.

(2) You saw this statement used previously as a reset signal. The important thing to notice here is that the width of the resulting pulse is 20ns (40ns – 20ns), and not 40ns as this line of code may lead you to think. This represents a use of "absolute time" in the testbench. Testbenches are typically easier to write and understand when they use "relative time", which is why this chapter uses relative time so extensively.

(3) This code synthesizes a load pulse for the counter. Once again, the code uses absolute time, so the width of this load pulse is 30ns.

```
entity testbench2 is
end testbench2;

architecture stimulus of testbench2 is

   constant CLK_PERIOD: time := 50 ns;

   component COUNT_4B
   port ( RESET,CLK,LD,UP : in std_logic;
                    DIN : in std_logic_vector (3 downto 0);
                  COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up, s_ld : std_logic := '0';
   signal s_clk      : std_logic := '0';
   signal s_reset    : std_logic := '0';
   signal s_ld_val   : std_logic_vector(3 downto 0) := X"0";
   signal s_count    : std_logic_vector(3 downto 0) :=  X"0";

begin
   -- instantiate the counter
   DUT: COUNT_4B
   port map (CLK => s_clk,
             DIN => s_ld_val,
             RESET => s_reset,
             LD => s_ld,
             UP => s_up,
             COUNT => s_count);

   -- clock synthesis
   s_clk <= not s_clk after CLK_PERIOD/2;

   ------------------------------------------------------------(1)
   -- set initial signal values
   s_up <= '1';      s_ld_val <= X"E";

   ------------------------------------------------------------(2)
   -- synthesize reset signal
   s_reset <= '0', '1' after 20 ns, '0' after 40 ns;

   ------------------------------------------------------------(3)
   -- synthesize ld signal at 50ns
   s_ld <= '0', '1' after 50 ns, '0' after 70 ns;

end stimulus;
```

Figure 99.16: A block diagram for a basic VHDL testbench.

Example 99-3: The Second Testbench All Over Again

Write the VHDL code that implements the testbench model shown in the black box diagram of Figure 99.15(b). For this circuit, consider the DUT to be a 4-bit up/down counter with a synchronous parallel load (as modeled by Figure 99.15(a)).

Solution: There is an issue with the approach of the solution to Example 99-2. The issue is that beyond the setting initial values of signal and synthesizing one-time signals (such as resets) was that the testbench started becoming somewhat unintuitive. The primary reason for this issue was that we were not able to use “wait” statements in the solution because the solution did not use process statements.

When we use process statements, basic testbench writing becomes more intuitive and thus easier to write[16].

Figure 99.17 shows another testbench that exercises the 4-bit up/down counter. This solution uses both behavioral and dataflow statements in the testbench model. The fact that opted to put a majority of the "test" portion of the testbench in a process statement allows us to use wait statements. This testbench uses a process statement to perform most of the work. Note that there are two concurrent statements in this testbench: one generates the clock and the other does the testing work. This advertises the notion that concurrency in VHDL modeling also extends to testbenches. Once you model the clock synthesis, you're free to do the other test; the clock remains running. There are some other items worth noting in this testbench; the numbers below match the comments in the testbench model.

(1) The testbench uses a process statement. Because we want to use wait statements in this solution, the process statement does not include a sensitivity list.

(2) The first real work we do in the process statement is to reset the counter by sending it an active high pulse. Keep in mind that the reset signal is initialized to its non-asserted state. The statements associated with this comment say that the reset signal toggles at the beginning of the simulation and then toggles again 20ns later, resulting in a 20ns pulse.

(3) This code synthesizes a signal for a parallel load pulse. The important thing to note here is that the model is now working with relative time due to the nature of the "wait" statements. According to the given code, the load pulse is a 25ns wide positive pulse that occurs starting at 70ns. The 70ns time comes from the fact that this set of code waits 50ns after the 20ns delays associated with the reset pulse.

(4) This is a "wait until" statement; the process will pause at this statement until the condition associated with the "wait until" statement occurs. In this case, the counter is counting up and the wait statement "waits" until the counter reaches 10. At that time, the UP signal toggles, which forces the counter to count down (starting at the next clock cycle). This is important in terms of testbenches because what the model is doing is "waiting" for something to occur; when it occurs, the testbench assigns a value to some other signal associated with the testbench.

(5) This is an example of an assert statement (though not a great example). If execution of the process arrives at the assert statement, then we know the counter has passed the previous "wait until" statement. In this case, we know the counter to be at 10. The assert statement is saying, "if the counter is not 11 at this point, then print out this rude message". Assert statements are massively useful; this example will not be winning any awards. Figure 99.18 shows the resultant error message. Note that the simulator that generated this output included extra information in addition to the error message and severity level.

(6) The process ends with a final "wait" statement. This statement assures that no more statements in this process will evaluate by forcing the process to "wait" forever at this statement.

[16] This is less true with giant and complicated testbenches; we'll not go there now.

```
entity testbench3 is
end testbench3;

architecture stimulus of testbench3 is

   constant CLK_PERIOD: time := 40 ns;

   component COUNT_4B
   port ( RESET,CLK,LD,UP : in std_logic;
                     DIN : in std_logic_vector (3 downto 0);
                   COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up, s_ld : std_logic := '0';
   signal s_clk : std_logic := '0';
   signal s_reset : std_logic := '0';
   signal s_ld_val : std_logic_vector(3 downto 0) := X"0";
   signal s_count : std_logic_vector(3 downto 0) :=  X"2";

begin
   DUT: COUNT_4B -- instantiate counter
   port map (CLK => s_clk,
             DIN => s_ld_val,
             RESET => s_reset,
             LD => s_ld,
             UP => s_up,
             COUNT => s_count);

   -- clk synthesis
   s_clk <= not s_clk after CLK_PERIOD/2;

   ----------------------------------------------------(1)
   process
   begin
      s_up <= '1';   s_ld_val <= X"2";

      ----------------------------------------------------(2)
      -- synthesize reset signal
      s_reset <= '1';
      wait for 20 ns;
      s_reset <= '0';

      ----------------------------------------------------(3)
      -- synthesize loading signal
      wait for 50 ns;
      s_ld <= '1';
      wait for 25 ns;
      s_ld <= '0';

      ----------------------------------------------------(4)
      wait until s_count = X"A";
      ----------------------------------------------------(5)
      assert (s_count = X"B")
         report "This is a stupid test"
         severity Error;
      s_up <= '0';

      ----------------------------------------------------(6)
      wait;
   end process;
end stimulus;
```

Figure 99.17: A block diagram for a basic VHDL testbench.

Lastly for this example, Figure 99.18 shows a sample result of an example where the “assert” statements did not evaluate as being true. In this case, the testbench code shown in Figure 99.17 “waited” until the count output was a certain value and then tested to see if it was a different value. This

assert statement fails and the error message shown in Figure 99.18 is printed to the console[17]. Note that my particular simulator includes the exact time in the simulation where the error occurs.

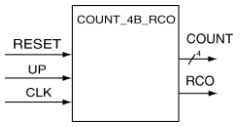
```
# ** Error: This is a stupid test
#    Time: 420 ns  Iteration: 2  Instance: /testbench3
```

Figure 99.18: A block diagram for a basic VHDL testbench.

Example 99-4: Another Counter Testbench

Write the VHDL code that implements a testbench model that verifies the up and down RCO output on the 4-bit counter model shown below.

```
entity COUNT_4B_RCO is
   port (RESET,CLK,UP : in std_logic;
                  RCO : out std_logic;
                COUNT : out std_logic_vector (3 downto 0));
end COUNT_4B_RCO;

architecture my_count of COUNT_4B_RCO is
   signal  t_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
   begin
       --RCO <= '0';
       if (RESET = '1') then
          t_cnt <= (others => '0');
       elsif (rising_edge(CLK)) then
          RCO <= '0';
          if (UP = '1') then
             t_cnt <= t_cnt + 1;
             if (t_cnt = X"E") then
                RCO <= '1';
             end if;
          else
             t_cnt <= t_cnt - 1;
             if (t_cnt = x"1") then
                RCO <= '1';
             end if;
          end if;
       end if;
   end process;

   COUNT <= t_cnt;

end my_count;
```



Solution: Figure 99.19 shows one possible solution this example. This solution is similar to the previous solution so we'll only comment on the new and significant items.

(1) The testbench initial resets the counter and starts the counter counting in the "up" direction. At that point, the testbench waits for the RCO signal to assert. Since the counter is counting in the

[17] This message is a result from the simulator I'm using. Your results may be different depending on the simulator that you're using.

up direction, the RCO signal should reset when the counter reaches 0xF. The test bench uses the “wait until” statement to monitor the RCO statement. When the RCO signal asserts, the first thing we need to do is wait a small amount of time; which is done with the following “wait for” statement. xxxWe need to do this because the simulator interprets the activity happening on the RCO and COUNT output signal to occur simultaneously. After the small delay, the testbench then verifies that the count does indeed match the max count value of 0xF.

(2) What we want here is have the counter continue counting in the up direction for about half of the count cycle. After that point we’ll change the direction of the count (COUNT = ‘0’) and verify the RCO is working in the down counting direction. This piece of code is another approach to inserting a delay in the testbench. Though we could have used a “wait for” statement here, we opted to use a loop construct. As of this writing, this text has not mentioned loops in VHDL, but they exist. This is an example of a simple look construct; you should find these straightforward. They are handy when you’re creating complicated testbenches.

(3) This piece of code changes the direction of the counter to count down after counting in the up direction for a few clock cycles.

(4) This code is similar to the code described in (1) above. This time, however, we are verifying that the counter displays a 0x0 when the RCO is asserted when the counter is counting in the down direction.

```
entity testbench4 is
end testbench4;

architecture stimulus of testbench4 is

   constant CLK_PERIOD: time := 40 ns;

   component COUNT_4B_RCO
   port ( RESET,CLK,UP : in std_logic;
                   RCO : out std_logic;
                 COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up : std_logic := '1';
   signal s_clk : std_logic := '0';
   signal s_reset : std_logic := '0';
   signal s_rco : std_logic := '0';
   signal s_count : std_logic_vector(3 downto 0) :=  X"0";

begin
   DUT: COUNT_4B_RCO -- instantiate counter
   port map (CLK => s_clk,
             UP => s_up,
             RESET => s_reset,
             RCO => s_rco,
             COUNT => s_count);

   s_clk <= not s_clk after CLK_PERIOD/2;

   process
   begin

      -- synthesize reset signal
      s_reset <= '1';
      wait for 20 ns;
      s_reset <= '0';

      ----------------------------------------------------(1)
      wait until s_rco = '1';
      wait for 1 ns;
         assert (s_count = X"F")
         report "RCO counting up is incorrect"  severity Error;

      ----------------------------------------------------(2)
      for M in 1 to 10 loop
         wait until rising_edge(s_clk);
      end loop;

      ----------------------------------------------------(3)
      s_up <= '0';

      ----------------------------------------------------(4)
      wait until s_rco = '1';
      wait for 1 ns;
         assert (s_count = X"0")
         report "RCO counting down is incorrect"  severity Error;

      wait;
   end process;
end stimulus;
```

Figure 99.19: A block diagram for a basic VHDL testbench.

The following comments deal with a slightly advanced issue in VHDL. Some people may not have read about or dealt with the issue of "signals" vs. "variables". If that is the case then you can skip the next few items and simply move onto the next examples. The notion of "signals" vs. "variables" is not overly complicated, so another option for you is to go read about the difference between signals and variables. The following discussion briefly mentions these differences.

In truth, the counter model shown in Example 99-4 has some issues that we needed to deal with in the testbench model. The problem with the counter model is that the original model was somewhat

confusing because it was implemented using signals. Figure 99.20 shows a clearer version of the counter. This difference between this model and the model provided as part of Example 99-4 is that the model uses a variable for the temporary count value. Because the results from operations done on variables are ready immediately, the model can look for the true RCO values for both the up and down counts. Note that in the model of Example 99-4, we had to generate the RCO signal one count in advance; in particular, "0xE" or "0x1" for the up and down counting directions, respectively. The two models are functionally equivalent, but the counter model shown in Figure 99.20 provides much less confusion.

```
entity COUNT_4B_RCO_VAR is
   port ( RESET,CLK,UP : in std_logic;
                   RCO : out std_logic;
                 COUNT : out std_logic_vector (3 downto 0));
end COUNT_4B_RCO_VAR;

architecture my_count of COUNT_4B_RCO_VAR is
   signal  t_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
      variable v_cnt : std_logic_vector(3 downto 0);
   begin
      if (RESET = '1') then
         v_cnt := (others => '0');
      elsif (rising_edge(CLK)) then
         RCO <= '0';
         if (UP = '1') then
            v_cnt := v_cnt + 1;
            if (v_cnt = X"F") then
               RCO <= '1';
            end if;
         else
            v_cnt := v_cnt - 1;
            if (v_cnt = x"0") then
               RCO <= '1';
            end if;
         end if;
      end if;
      t_cnt <= v_cnt;
   end process;

   COUNT <= t_cnt;
end my_count;
```

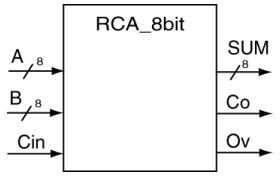
Figure 99.20: A more clear version of the RCO counter of the previous example.

Figure 99.21 shows a model for an 8-bit ripple carry adder. We'll use this model for the next few example testbenches. One thing worth noting in this example is that the RCA contains an overflow output in addition the carry-out output. For this RCA model, we've defined a carry out to be the condition where the sign bit of the two operands are equivalent but differs from the sign-bit of the result.

The next few testbenches test the RCA using the same set of test vectors. As you'll see, the main differences between the upcoming examples is where they obtain the test vectors from. There are several other differences between the upcoming examples; the example solutions briefly describe these differences.

```
entity RCA_8bit is
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
end RCA_8bit;

architecture RCA_8bit of RCA_8bit is
begin

   process(A,B,Cin)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      v_res := ('0' & A) + ('0' & B) + Cin;
      Ov <= '0';

      if (A(7) = B(7)) then
         if (A(7) /= v_res(7)) then
            Ov <= '1';
         end if;
      end if;

      SUM <= v_res(7 downto 0);
      Co <= v_res(8);
   end process;

end RCA_8bit;
```

(a)



(b)

Figure 99.21: The RCA model used as the DUT for the next three examples.

Example 99-5: Test Vectors Generated On the Fly

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 99.21. The testbench should use "on the fly" test vectors; use three sets of test vectors to test the RCA.

Solution: The approach taken by this problem is to generate the test vectors "on the fly". This is definitely the most straightforward approach, but it is not well suited for all testbenches. In particular, generating test vectors "on the fly" is only feasible for testbenches that are relatively short and don't have the need to exercise a large set of test vectors.

Figure 99.21 shows the final testbench for this example problem. There are a few relatively interesting things to note about this solution:

(1) The approach here is to provide the values directly to the augend and addend. These values are essentially "hardcoded" into the VHDL model. These hardcoded values create the notion of the test vectors are "on the fly". After the assignment of the operand values, the testbench "waits" using a wait statement. After the expiration of the wait statement, the testbench verifies that the sum, the overflow, and the carry-out values are correct. The verification of these values is also hardcoded and thus modeled as "on the fly".

(2) This represents the testing of the second set of test vectors. This is similar to the previous comments above, so not much to say here.

(3) This represents the testing of the third and final set of test vectors. Again, not much to say here.

```
entity testbench6 is
end testbench6;

architecture stimulus of testbench6 is

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
              Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin : std_logic := '0';
   signal s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_Ov,
             Co => s_Co,
             SUM => s_sum);

   process
   begin

      s_cin <= '0';
      wait for 20 ns;

      ------------------------------------------------------------(1)
      s_A <= X"C3";      s_B <= X"54";
      wait for 50 ns;
      assert (s_sum = X"17")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '0')
         report "overflow is not correct"  severity Error;
      assert (s_co = '1')
         report "carry-out is not correct"  severity Error;

      ------------------------------------------------------------(2)
      s_A <= X"82";     s_B <= X"84";
      wait for 50 ns;
      assert (s_sum = X"06")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '1')
         report "overflow is not correct"  severity Error;
      assert (s_co = '1')
         report "carry-out is not correct"  severity Error;

      ------------------------------------------------------------(3)
      s_A <= X"85";     s_B <= X"24";
      wait for 50 ns;
      assert (s_sum = X"A9")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '0')
         report "overflow is not correct"  severity Error;
      assert (s_co = '0')
         report "carry-out is not correct"  severity Error;

      wait;
   end process;
end stimulus;
```

Figure 99.22: A solution for Example 99-5.

The testbench model shown in Figure 99.22is an implementation of the testbench model shown in Figure 99.3 and repeated in Figure 99.23 for your viewing convenience. The solution to Example 99-5 effectively uses assert statements in order to implement the "results comparison" shown in Figure 99.23. In the context of Figure 99.23, the solution to Example 99-5 does not actually have an official "pass/fail indication". What will occur is that the assert statements will print a message to the console if there is an error.

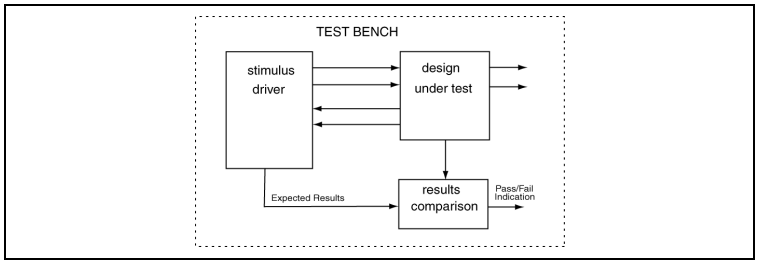


Figure 99.23: A block diagram for a basic VHDL testbench.

Example 99-6: Test Vectors Stored in Arrays

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 99.21. The testbench should use test vectors that are stored in an array object; use three sets of test vectors to test the RCA. The testbench should verify that the value of the SUM output of the RCA is correct and base the success of the testing on only the SUM value.

Solution: This example problem once again verifies the proper operation of the RCA. This example uses test vectors stored in arrays included in the testbench model as opposed to the "on the fly" test vector storage of the previous example. You should note that though these test vectors are stored in arrays, they're still "hardcoded" into arrays. The main advantage this approach has over the true "on the fly" approach is that all the test data is in one location instead of spread out through your code as it was in the "on the fly" approach. Figure 99.25 shows a solution to this example; here are a few more items to note in this solution.

(1) VHDL allows the use of arrays, but it considers arrays to be a "type" of their own. This means that you need to explicitly describe arrays before you use them. This line of code defines arrays of size "three" in both vector and scalar forms; the vector arrays hold sum and operand inputs while the scalar arrays hold values for the carry-out and overflow. The use of the constant for the array size represents generic programming practices and is therefore happy.

(2) This set of code defines constants of the array types defined in the previous step. Note that we've used hex notation for the 8-bit vectors to help the code appear neater. Also, note that these five array declarations use the two array definitions defined in the previous step.

(3) This testbench uses a signal to state whether the DUT is working properly or not. Somewhat peculiar is the fact that this testbench only tests the results of the addition operation but does not check the carry-out and overflow results. Thus, the final verification is based on whether

the SUM is correct but not the carry-out and overflow[18]. Note that we initialized the s_fail signal to '0'.

(4) In order to simplify this testbench, we placed a bulk of the testing inside of a VHDL iterative construct. This testbench uses a VHDL looping construct, which is good approach based on the notion that we need to perform the same testing but on different sets of vectors. Note that looping construct iterates from '0' to '2', which highlights the notion that arrays in VHDL are zero-based.

(5) This code tests whether the results of the addition operation are correct or not. If the SUM is not correct in any set of test vectors, the s_fail signal is asserted (s_fail = '1'). One way a user will be able to discern whether the test passed or failed is to monitor the value of the s_fail signal; the other way is to monitor the results of the assert statements.

(6) This set of code is similar to the previous examples in that the testbench uses assert statements to indicate if there is an error in some RCA operation. If there is an error, the assert statements print a message out to a console window in the test environment.

Finally, the solution to this example once again uses the model shown yet again in Figure 99.24. The only comment here is that the official "pass/fail" indication will be a signal on a waveform display in the simulator you're using for your testing. Nothing too fancy here.

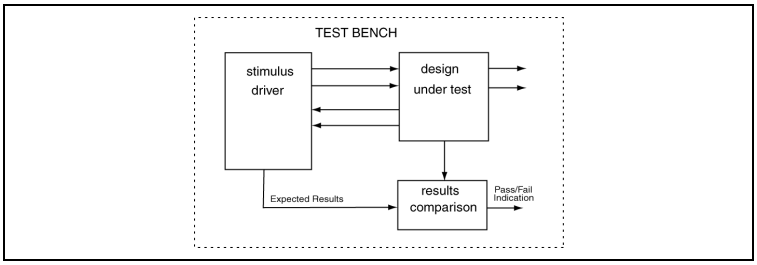


Figure 99.24: A block diagram for a basic VHDL testbench with "result comparison" capabilities.

[18] This was an arbitrary choice, and not a good one at that.

```
entity testbench7 is
end testbench7;

architecture stimulus of testbench7 is
   constant VEC_NUM : integer := 3;

   ------------------------------------------------------------(1)
   type vec_arr is array (0 to (VEC_NUM-1)) of std_logic_vector(7 downto 0);
   type bit_arr is array (0 to (VEC_NUM-1)) of std_logic;

   ------------------------------------------------------------(2)
   constant A_arr: vec_arr := (X"C3", X"82", X"85");
   constant B_arr: vec_arr := (X"54", X"84", X"24");
   constant Sum_arr: vec_arr := (X"17", X"06", X"A9");
   constant co_arr: bit_arr := ('1', '1', '0');
   constant ov_arr: bit_arr := ('0', '1', '0');

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
              Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

   ------------------------------------------------------------(3)
   signal s_fail : std_logic := '0';

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_Ov,
             Co => s_Co,
             SUM => s_sum);

   process
   begin
      s_cin <= '0';
      wait for 20 ns;

      ------------------------------------------------------(4)
      for N in 0 to (VEC_NUM-1) loop

         s_A <= A_arr(N);     s_B <= B_arr(N);
         wait for 50 ns;

         ------------------------------------------------(5)
         if (s_sum /= Sum_arr(N)) then
            s_fail <= '1';
         end if;

         ------------------------------------------------(6)
         assert (s_sum = Sum_arr(N))
            report "SUM is not correct"  severity Error;
         assert (s_ov = ov_arr(N))
            report "overflow is not correct"  severity Error;
         assert (s_co = co_arr(N))
            report "carry-out is not correct"  severity Error;

      end loop;

      wait;
   end process;
end stimulus;
```

Figure 99.25: The solution to Example 99-6.

Example 99-7: Vectors Stored in External Files

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 99.21. The testbench should use test vectors that are stored in an external file; use three sets of test vectors to test the RCA. The testbench should verify that the value of the SUM output of the RCA is correct and base the success of the testing on only the SUM value.

Solution: This example is similar to the previous example but the test vectors are stored in a file rather than "on the fly" or in arrays. Figure 99.26 shows a printout of the test vector file accessed by this examples. There are a few things to note about this file:

- The file uses a '$' delimiter to indicate that the given line is a comment and thus should be ignored by the testbench. As you'll see from the testbench model, this choice of comment symbols is arbitrary.

- Each non-comment line in the test file contains five data fields: three 8-bit values and two 1-bit values. We arbitrarily chose to delineate the individual data fields using a single space. We could have opted to more space between data fields or none at all; the testbench code can handle either case based on the mechanism the testbench model uses to access the test vector file.

```
$-------------------------------------------------
$ tb8.txt: Sample test vector file
$
$ file format (name(width):
$
$     A(8), B(8), Sum(8), Co(1), Ov(1)
$
$-------------------------------------------------
11000011 01010100 00010111 1 0
10000010 10000100 00000110 1 1
10000101 00100100 10101001 0 0
```

Figure 99.26: A file of test vectors for use by Example 99-7.

There are many new and happy things to note about the solution Example 99-7 shown in Figure 99.28. Here are a few of those things as referenced in the testbench code.

(1) This testbench reads the input vectors from a file, so we first must open the associated file. This statement declared a "file" object as a text type, associated the type with the file named "tb8.txt", and opens that file in a "read" mode. The label "f_test_vects" is essentially a file handle that the testbench uses to retrieve information from the file.

(2) In most cases regarding testbenches, variables are more intuitive to work with. The issue is that we need the associated values immediately; we don't want to insert wait statements in order to force an update of values represented by signals. You'll assuredly be working more with variables in your testbenches as opposed to signals. Get used to it.

(3) The testbench reads data from the file one line at a time. The testbench code reads a line from the test vector file and places it into a variable of a "line" type, which is named "v_vec_line". Note the use of the file handle in this line.

(4) It's always good practice to comment things, even files containing test vectors. VHDL has no concept of how a test vector file will indicate a comment line. What this line does is essentially treat every line in the test vector file as a comment line if the first character in the line is a '$'. The choice of '$' is completely arbitrary. Note the VHDL keyword "next" executes if the testbench finds a comment character.

(5) This set of lines reads the test vectors from the "line" that was previously read from the test vector file. This data is read directly into the associated variables, which essentially has the affect of assigning the values to those variables. The associated hardware then acts on this new information and generates results.

Figure 99.28 shows the final result to Example 99-7. The solution uses a small font size and does not use comments in an effort to keep the solution less than one page. Figure 99.27 shows the high-level block diagram model of the testbench (we originally presented this diagram earlier in this chapter). As with the previous example, the official "pass/fail" indication will be a signal on a waveform display in the simulator you're using for your testing.

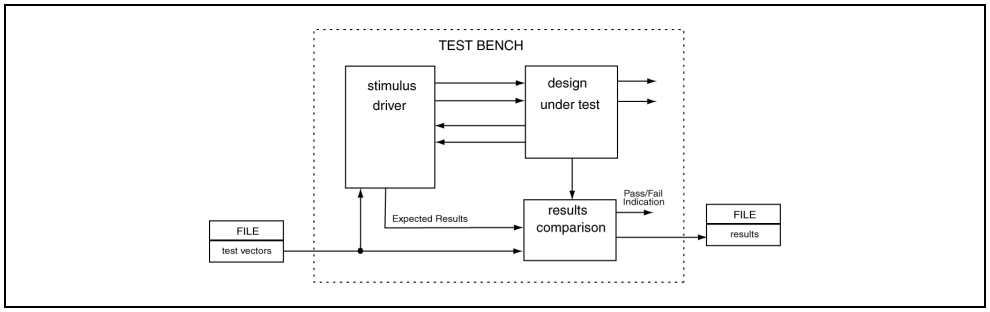


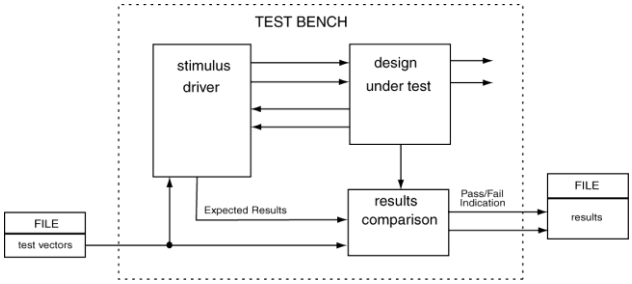
Figure 99.27: A block diagram for testbench solution of Example 99-7.

```
entity testbench8 is
end testbench8;

architecture stimulus of testbench8 is
   ----------------------------------------------------(1)
   file f_test_vecs: text open read_mode is "tb8.txt";

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

   signal s_fail : std_logic := '0';

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_ov,
             Co => s_co,
             SUM => s_sum);

   process
      ----------------------------------------------------(2)
      variable v_vec_line: line;
      variable v_A, v_B : std_logic_vector(7 downto 0);
      variable v_sum_test : std_logic_vector(7 downto 0);
      variable v_ov_test, v_co_test : std_logic;
   begin

      s_cin <= '0';     wait for 20 ns;

      while not endfile(f_test_vecs) loop

         ------------------------------------------------(3)
         readline(f_test_vecs, v_vec_line);

         ------------------------------------------------(4)
         if v_vec_line(1) = '$' then
            next;
         end if;

         ------------------------------------------------(5)
         read(v_vec_line,v_A);        read(v_vec_line,v_B);
         read(v_vec_line,v_sum_test);
         read(v_vec_line,v_co_test);
         read(v_vec_line,v_ov_test);

         s_A <= v_A;     s_B <= v_B;
         wait for 50 ns;

         if (s_sum /= v_sum_test) then
            s_fail <= '1';
         end if;

         assert (s_sum = v_sum_test)
            report "SUM is not correct"  severity Error;
         assert (s_ov = v_ov_test)
            report "overflow is not correct"  severity Error;
         assert (s_co = v_co_test)
            report "carry-out is not correct"  severity Error;

      end loop;
      wait;
   end process;
end stimulus;
```

Figure 99.28: The solution to Example 99-7.

Example 99-8: Vectors Stored in External Files

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 99.21. Model your testbench solution using the following testbench model. Use the file shown in Figure 99.26 for the test vectors. Make sure verification of the DUT will fail if any of the DUT's outputs are not correct.



Solution: This example is similar to the previous example but there are a few extra items. First, from examining the testbench diagram shown in the problem description, you can see that this testbench will be different from the testbench of the previous example because the pass/fail indication must be written to a file. The problem statement also states that we need to verify every operation associated with the RCA; recall from the previous example that we only verified the results of the addition operation.

Figure 99.29 shows a solution to Example 99-8. As always, there are many items worth noting about the solution shown in Figure 99.28. Here are a few of those things as referenced using the parenthetical comments in the testbench code.

(1) This testbench reads the input vectors from a file and writes the results to another file. The testbench opens both of these files as text files, the test vector file in read mode, and the result file in write mode.

(2) This set of code replaces the assert statements that essentially performed the same function in the solution to the previous example. This testbench writes error messages to the previously opened results file; recall that the assert statements write messages to the console. This set of code include the time with the use of the VHDL "now" keyword. Note that the sum, carry-out, and overflow results as verified with this set of code. Also, note that if any of these results are not correct, the entire test fails as noted by the assignment of the "v_fail" signal in each of the "if" cases in this set of code. The last thing to note regarding this set of code is that the solution writes to a previously declared "line" type, and then write that entire line to the output file. Note that successive writes to a "line" type are appended to previous writes to that line. The "writeline" statement writes the line to the output file, which additionally has the affect of clearing the "line" type buffer. In the way, the next "write" statement writes to an empty "line" buffer.

(3) This set of code verifies checks the "v_fail" variable to discern whether the testbench was able to verify proper operation of the DUT. Note that anytime there was an error detected, the "v_fail" signal was set to true.

```
entity testbench9 is
end testbench9;

architecture stimulus of testbench9 is
   ------------------------------------------------- ----------------(1)
   file f_test_vecs: text open read_mode is "tb8.txt";
   file f_test_results: text open write_mode is "tb9_results.txt";

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
          Ov,Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (s_A, s_B, s_cin, s_ov, s_co, s_sum);

   process
      variable v_fail : std_logic:='0';
      variable v_vec_line: line;
      variable v_results_line: line;
      variable v_A, v_B : std_logic_vector(7 downto 0);
      variable v_sum_test : std_logic_vector(7 downto 0);
      variable v_ov_test, v_co_test : std_logic;
   begin

      s_cin <= '0';
      wait for 20 ns;

      while not endfile(f_test_vecs) loop

         readline(f_test_vecs, v_vec_line);

         if v_vec_line(1) = '$' then
            next;
         end if;

         read(v_vec_line,v_A);         read(v_vec_line,v_B);
         read(v_vec_line,v_sum_test);
         read(v_vec_line,v_co_test);
         read(v_vec_line,v_ov_test);

         s_A <= v_A;     s_B <= v_B;
         wait for 50 ns;

         ---------------------------------- ---- ------------- -----(2)
         if (s_sum /= v_sum_test) then
            write(v_results_line,String'("SUM is not correct at "));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;
         if (s_ov /= v_ov_test) then
            write(v_results_line,String'("Overflow is not correct "));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;
         if (s_co /= v_co_test) then
            write(v_results_line,String'("Carry_out is not correct"));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;

      end loop;

    ------------------------------------------------  -- - ------------(3)
    if (v_fail = '1') then
       write(v_results_line,String'("The DUT failed testing."));
       writeline(f_test_results,v_results_line);
    else
       write(v_results_line,String'("The DUT tested properly."));
       writeline(f_test_results,v_results_line);
    end if;

      wait;
   end process;
end stimulus;
```

Figure 99.29: A solution to Example 99-8.

Chapter Summary

- A “testbench” is the term VHDL uses to name a VHDL model whose job it is to verify the proper operation of a given digital circuits. The term “device under test”, or DUT, is used to refer to the circuit being tested by a given testbench. The form of a testbench is such that the DUT is instantiated and thus becomes part of the testbench. VHDL testbenches are comprised of VHDL code but unlike the DUT, the testbench is not synthesizable. The two main components of a testbench are the DUT and the stimulus driver.

- VHDL testbenches verify proper DUT operation by either manual or automatic testing. The person writing the testbench decides on an appropriate type of testing. Manual testing requires a human to examine the waveforms generated by the testbench to verify proper circuit operation. Automatic testing uses the versatility of VHDL to have the testbench model state directly whether the DUT model operates as expected.

- Test vectors for VHDL testbenches come from one of three sources: 1) “on the fly”, 2) from composite VHDL structures (such as arrays), or 3) from external files. Each of these sources are considered to emanate from the stimulus driver box of the testbench.

- VHDL can use “assert” statement to verify proper circuit operation. The assert statement consists of “report” lines and/or “severity” lines. If the expression associated with the assert statement does not evaluate as true, the VDL code executes the “report” and “severity” lines.

- VHDL process statements have two forms: either they use a sensitivity list or they use wait statements, but they cannot use both.

- There are four different types of wait statements: 1) wait for a change in a signal (WAIT ON), 2) wait until an expression evaluates are true (WAIT UNTIL), 3) wait for a specific amount of time (WAIT FOR), and 4) wait forever (WAIT).

Chapter Exercises

1) What are the two main purposes for simulating your digital circuits?

2) When will you know it's time to break down and simulate your digital circuit?

3) Briefly describe the two main approaches your testbench can use in order to verify a circuit is operating as expected.

4) Briefly describe the three main approaches to generating test vectors in a VHDL testbench.

5) Briefly describe the primary difference between an "assert" statement and an "if" statement?

6) List all the types of VHDL provided error messages that are associated with "severity" lines in VHDL "assert" statements.

7) Briefly describe whether is it possible to use neither a "report" line nor a "severity" line when using "assert" statements.

8) Are VHDL "assert" statements considered sequential statements or concurrent statements? Briefly describe the skinny on this notion.

9) Briefly describe the main differences between the two forms of wait statements.

10) Can a process statement include both a sensitivity list and a wait statement? Briefly describe why or why not.

11) How many types of wait statements are included in VHDL and what are they?

12) Consider a process statement that uses a "wait" statement. Briefly describe what happens when execution of the sequential statements in the process reaches the end?

13) Write a testbench that would completely test the following VHDL model. Use "on the fly" test vectors and use manual verification in your solution.

```
-------------------------------------------------------------------
-- RET D Flip-flop model with active-low synchronous set input.
-------------------------------------------------------------------
entity d_ff_ns is
    port (D,S,CLK : in  std_logic;
                Q : out std_logic);
end d_ff_ns;

architecture my_d_ff_ns of d_ff_ns is
begin
   dff:  process (D,S,CLK)
   begin
      if (rising_edge(CLK)) then
         if (S = '0') then
            Q <= '1';
         else
            Q <= D;
         end if;
      end if;
   end process dff;
end my_d_ff_ns;
```

14) Write a testbench that would completely test the following VHDL model. Use test vectors from arrays and use manual verification in your solution.

```
-------------------------------------------------------------------
-- RET T Flip-flop model with active-low asynchronous set input.
-------------------------------------------------------------------
entity t_ff_s is
    port ( T,S,CLK : in  std_logic;
                Q : out std_logic);
end t_ff_s;


architecture my_t_ff_s of t_ff_s is
   signal s_tmp : std_logic; -- intermediate signal declaration
begin
   tff:  process (T,S,CLK)
   begin
      if (S = '0') then
         Q <= '1';
      elsif (rising_edge(CLK)) then
         s_tmp <= T XOR s_tmp; -- temp output assignment
      end if;
   end process tff;

   Q <= s_tmp; -- final output assignment

end my_t_ff_s;
```

15) Write a testbench that would completely test the following VHDL model. Use test vectors from arrays and use automatic verification in your solution.

```
-----------------------------------------------------------------
-- Model for a universal shift register
-----------------------------------------------------------------
entity univ_sr is
    port (    SEL : in std_logic_vector(1 downto 0);
           P_LOAD : in std_logic_vector(7 downto 0);
            D_OUT : out std_logic_vector(7 downto 0);
              CLK : in std_logic;
            DR_IN : in std_logic;  -- input for shift left
            DL_IN : in std_logic); -- input for shift right
end univ_sr;

architecture my_sr of univ_sr is
   signal s_D : std_logic_vector(7 downto 0);
begin

   process (CLK,SEL,DR_IN,DL_IN,P_LOAD)
   begin
      if (rising_edge(CLK)) then

         case SEL is
            -- do nothing (don't change state) --------------
            when "00" =>  s_D <= s_D;

            -- parallel load --------------------------------
            when "01" =>  s_D <= P_LOAD;

            -- shift right ----------------------------------
            when "10" =>  s_D <= DL_IN & s_D(7 downto 1);

            -- shift left -----------------------------------
            when "11" =>  s_D <= s_D(6 downto 0) & DR_IN;

            -- default case ---------------------------------
            when others => s_D <= "00000000";
         end case;

      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

16) Write a testbench that would completely test the following VHDL model. Use test vectors from external files and use automatic verification in your solution.

```
entity my_fsm2 is
    port (   TOG_EN : in  std_logic;
             CLK,CLR : in  std_logic;
                Y,Z1 : out std_logic);
end my_fsm2;

architecture fsm2 of my_fsm2 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     if (CLR = '1') then
        PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      case PS is
      Z1 <= '0';

         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;
   end process comb_proc;

   -- assign values representing the state variables
   with PS select
      Y <= '0' when ST0,
           '1' when ST1,
           '0' when others;

end fsm2;
```

17) Write a testbench that would completely test the following VHDL model. Use test vectors from external files and use automatic verification in your solution.

```
entity clk_div is
    Port (    clk : in std_logic;
           div_en : in std_logic;
             sclk : out std_logic);
end clk_div;


architecture my_clk_div of clk_div is

   type my_count is range 0 to 100;        -- user-defined type
   constant max_count : my_count := 63;    -- user-defined constant

   signal tmp_sclk : std_logic;            -- intermediate signal for clock

begin
   my_div: process (clk, div_en)

      variable div_count : my_count := 0;

   begin
      if (rising_edge(clk)) then    -- look for clock edge
         if (div_en = '1') then     -- divider enabled
            if (div_count = max_count) then
               tmp_sclk <= not tmp_sclk;    -- toggle output
               div_count := 0;      -- reset count
            else
               div_count := div_count + 1;
            end if;
         else                       -- divider disabled
            div_count := 0;         -- reset count
            tmp_sclk <= '0';        -- turn off output
         end if;
      end if;
   end process my_div;

   s_clk <= tmp_sclk;   -- assign to output

end my_clk_div;
```

11 Glossover

(Bryan Mealy 2012 ©)

-A-

ABEL: An early hardware description language (HDL); it's still used today but it's tough to find someone who would admit to using it.

Absolute Time: A term used to describe one of two methods used to represent time in VHDL simulations. Absolute time refers to the notion that all references to time are based on an "absolute" number, such as the beginning of the simulation. VHDL can also use the notion of "relative time" (see "relative time").

Academic Administrators: A term referring to alien d-bags representing the largest obstacle to actual learning in an academic environment. .

Academic Exercise: Any amount of work that looks good and keeps you busy but actually has no meaningful purpose in the real world.

Academic Purposes: Any process or endeavor that requires time but has no lasting meaning or effect.

Academic-Types: That special type of person who is intent on being successful in academia at any cost. The hallmark of an academic-type student are that they generally gets good grades but typically don't know squat. The hallmark of an academic-type teacher is the one who generally places little or no effort into teaching; any effort they put into anything is primarily focused on advancing their careers (which in modern academia has nothing to do with providing quality teaching). The hallmark of an academic administrator are the ones who do nothing while placing amazing amounts of efforts into justifying their extremely overpaid academic existence.

Action State: The voltage level of a signal associated with notion that some action should take place when the signal is at this level; same as "active state".

Active Edge: A term that refers to either a "0→1" transition (rising edge) or a 1→0" transition (falling edge) of a signal that is used to synchronize changes in the state of the circuit.

Active State: The voltage level of a signal associated with notion that some action should take place when the signal is at this level; same as "action state".

ADC: An acronym representing "analog-to-digital conversion"; (see "Analog-to-Digital Conversion").

Addend: A number added to another number to form a sum.

Adder: A device that adds numbers. In digital design, there are many forms of adders, each with their own particular set of characteristics.

Adjacency Theorem: One of the basic theorems associated with Boolean algebra. This theorem facilitates the use of Karnaugh Maps to reduce Boolean functions. This theorem is sometimes referred to as the "Combining Theorem".

Administrator: A person who purposely creates problems and/or purposely prevents others from solving existing problems. And if you manage to known problems despite the efforts of administrators, they'll sure attempt to claim credit for your efforts.

Algebra: A mathematical system used to generalize arithmetic operations by using letter or symbols to stand for numbers based on rules derived from a minimal set of basic assumptions.

Algorithm: A step-by-step procedure for solving problems including the notion that the problem can be solved in finite number of steps.

ALU: An acronym referring to the "arithmetic logic unit"; (see "arithmetic logic unit").

Analog vs. Digital: The term *digital* refers to items that are discrete in nature while the term analog refers to items that are continuous in nature. While the world we live in is primarily analog, computers are primarily digital. One important function of digital design is to allow the successful interaction between computers the rest of the analog world.

Analog vs. Digital: The term *digital* refers to items that are discrete in nature while the term analog refers to items that are continuous in nature. While the world we live in is primarily analog, computers are primarily digital. One important function of digital design is to allow the successful interaction between computers the rest of the analog world.

Analog: A description of something that (such as a signal or data) that is expressed by a continuous range of values. The continuousness of analog implies that there are an infinite number of possible values in the given range.

Analog: A description of something that (such as a signal or data) that is expressed by a continuous range of values. The continuousness of analog implies that there are an infinite number of possible values in the given range.

Analog-to-Digital Conversion: A term that describes the translation of a signal represented by a voltage level (analog) to a signal represented by a given number of bits (digital). The term "ADC" is a shorthand representation of analog-to-digital conversion.

AND Plane: A structured array of logic that allows for the combination of Boolean variables and/or function outputs in such a way as to form product terms used to implement Boolean functions.

AND/NOR Form: One of the basic eight logic forms based but not commonly used in digital design. This form is derived from OR/AND form (POS form) by excessive use of DeMorgan's theorem.

AND/OR Form: One of the basic eight logic forms and one of the most popular four ways to describe a circuit using either Boolean equation or the circuit model of the associated Boolean equation. This form is often referred to as "sum of product" form or SOP form.

Annotations: This word is related to "notes". Any time you're attempting to describe something to someone, you should include as many "notes" or annotations as possible. Annotations should always be included with timing diagrams, block diagrams, state diagrams, and circuit schematics.

Architecture (VHDL): The part of a VHDL model that describes the operational characteristics of a circuit. The architecture is associated with a VHDL entity.

Architecture: A term that refers to the structure of a device; in particular, the modules contained in that device and how those modules are connected.

Architecture: In the context of digital hardware, the architecture of circuit describes the individual modules of a circuit and the connection between the modules.

Arithmetic Logic Unit: A circuit that has inputs for one or more operands, inputs for one or more controls, and output for the results. Arithmetic logic units generally contain status outputs describing various characteristics of the operations performed by the unit. The acronym "ALU" is often used to refer to this circuit.

Arithmetic Shift: A shift register operation on signed binary numbers that protects both the value and sign of the shifted number. Arithmetic shifts include both left and right shifts.

Arithmetic Unit: A term describing one of the main sub-modules of an arithmetic logic unit (ALU). The arithmetic unit generally handles operations that can be classified as "arithmetic" in nature such as addition, subtraction, multiplication, etc.

Assertation Levels: A term that references the notion that signal can be either negative or positive logic.

Asserted High: A term that refers to the notion that a given signal is a positive logic.

Asserted Low: A term that refers to the notion that a given signal is a negative logic.

Asserted: The notion that the current state of a signal (or voltage level) is associated with the action state. Whether a signal is asserted or not is independent of the logic level (negative or positive) associated with that signal.

Assignment Operator: A symbol that represents the transfer of information from one expression to another. The characters "<=" represent the assignment operation in VHDL while "=" is used as the assignment operator in C.

Asynchronous Input: An input to a sequential circuit that affect the circuit any time the signal is asserted (as opposed to being synchronized to some other signal in the circuit).

Augend: A number that is going to be used to increase (or added to) the value of another number.

Automatic Verification: A term that refers to the notion of a VHDL testbench's ability to discern whether a particular VHDL model is working as expected. The testbench designer can construct the testbench such that the testbench will directly state whether the model is working or not; this is opposed to "manual verification" which is the other main approach to VHDL model verification; (see "manual verification").

Axiom: A statement that is universally accepted as true.

-B-

Barrel Shift: A shift operation that is characterized by shifting more than one bit location in one clock cycle. Arithmetic shifts can include both left and right shifts.

Base: relative to a given number systems, the base is the same value as the radix.

Base: relative to a given number systems, the base is the same value as the radix.

BCD: An acronym used for binary coded decimal; (see "binary coded decimal").

Behavioral Style: A term that refers to using behavioral models in VHDL.

BFD: An acronym that referring to "brute force design"; this is essentially a pejorative synonym for the "iterative design".

Binary Coded Decimal: A number system that uses four bits (binary digits) to represent each digit in a decimal number. Four bits can provide up to 16 different values, which include digits (0-9) and sometimes alpha characters (A-F).

Binary Counter: A counter that counts in a binary sequence.

Binary Encoding: A term that refers to on of many different methods used to encode the state variables associated with the various states in a finite state machine (FSM). In particular, binary encodings the fewest number of single-bit storage elements possible to differentiate the various states in the FSM.

Binary Relationship: A relationship between two entities where at least one of the entities utilizes a binary exponential relationship (or a "powers of two" relationship).

Binary: A number system that uses two symbols to represent quantities. These symbols are typically '0' and '1' for digital design and computer applications.

Bit Stuffing: A phrase used to describe the notion of adding 0's to a number such that the addition of the 0's do not affect the overall value of the number.

Bits: A shorthand name for binary digits.

Bit-Stream: A term that refers to a contiguous set of bits on a single signal over a given time period. We oftern refer to bit-streams as "serial lines"; (see "serial lines").

Block-Style Comments: A commenting style where multiple lines of code can be commented by using a comment start delimiter and a comment end delimited (similar to "/*" and "*/" in the C programming language. VHDL does not support block commenting.

Bloviation: A technique used to enhance one's particular image of self-importance by wasting the time of others who are polite enough not to say anything. This is approach is used often by academic administrators who know people under them are generally to scared to do anything other than feign interest in the speaker.

Board-level Digital Design: Digital designs comprised primarily of discrete ICs interfaced in such a way as to achieve a meaningful result.

Boole, George: A 19th century mathematician who developed a two-valued algebra in order to mathematically model logical reasoning. The result of his work is Boolean Algebra and forms the basis of modern digital design.

Boolean Algebra: An algebra originally developed by George Boole in order to mathematically model logical reasoning. Boolean algebra forms the basis of modern digital design.

Boolean Equation: An equation that is contains Boolean algebra; these are sometimes referred to as Boolean expressions.

Boolean Expression: Another term referring to a Boolean equation.

Boolean Variable: A symbolic value that can represent one of two values; in digital design, these values are typically '1' or '0', but sometimes, "true" or "false", "on" and "off", etc.

Boring: A great description of anything you're not seeing the point of.

Borrow: A term referring to the notion that if a larger number is subtracted from a smaller number, the operation needs to access the next highest bit outside of the bit range associated with the subtraction. The borrow analogous to the "carry-out" bit associated with an addition operation. Often times, arithmetic modules use a signal bit to represent both the carry and borrow with the actual meaning of the bit being dependent on the operation that generated it.

Bottom-Up Design: A hierarchical design approach that starts at the lowest level of abstraction and works upwards. In this approach, the designer basically initially develops low-level modules that will be used by higher levels of the design.

Buffer: A device that accepts a signal as an input and outputs the exact same signal without a change in logic levels. Buffers typically used to increase drive output drive characteristics of a given signal.

Bummer: A brief description of the feeling you get when you find out that your precious circuit is not behaving as you expected it to.

Bundle Access Operator: A set of symbols, "( )", used in VHDL to access individual signals within an bundled signal.

Bundle Notation: The act of showing or describing bundles in circuit models such as schematic diagrams and timing diagrams.

Bundle: A set of signals that have been arbitrarily associated with each other (or group together) because they share a common purpose in a design.

Bus Notation: A synonym for the term "bundle notation"; the term bundle notation is preferable.

Bus: A term often used in place of the term "bundle". A bus is often used as a synonym for a bundle but is more appropriately a synonym for the term "protocol".

By Inspection: A term that refers to the notion that some problems can be solved in your brain, thus removing the need for expending extra time explicitly writing down solutions.

-C-

CAD: An acronym for "computer aided design"; (see "computer aided design").

Carry-Out: A bit indicating whether a "carry" has been generated by a digital device. Carry-out bits are generally associated with digital devices implementing arithmetic operations; carry-out bits are typically used to indicate the validity of mathematical operations and to allow the "daisy-chaining" or "cascading" of individual digital devices.

Cascade: A term referring a configuration of multiple digital devices; devices in a cascade configuration are placed in a series-type configuration. This term is often referred to as a "daisy chain".

Cascadeable: A characteristic of register, particularly counters and shift registers, that allows the effective bit-width of the device to be effortlessly extended by adding more modules to the design.

Case Sensitive: A term that refers to the notion that the syntax of a specific programming language or hardware description language differentiates between upper and lower case of alpha characters. The C programming language is case sensitive while VHDL is not case sensitive (about 99.9% of the time).

Case Statement: A statement that supports selection construct associated with multiple conditional statements. The case statement in VHDL is one of three main sequential statements that can appear in the body of process statements.

Cave: A dark place where I spent my time writing this text.

Central Processing Unit: A term used to describe a module that handles the processing of data in a computer. The notion of central refers to the notion that computers typically used only one processing unit in a central location (this is less true in modern digital design). The central processing unit, or CPU, is typically modeled as having two sub-modules: a control unit and an arithmetic logic unit.

Characteristic Table: A set of data presented in a tabular format that describes the operation of a digital circuit. The term characteristic table is most often associated with the description of sequential circuits since they include the notion of "state"; (see "state").

Chip Enable: A signal used in digital design to "turn on" or "turn off" a circuit. When a device is not enabled, the device has a predetermined output, which must be stated. When the device is enabled, the device works as a normal device.

Chip Select: A term used to describe whether a specific input is "turned on" or "turned off". See "chip enable" for a more complete description.

Circuit Forms: A term that refers to the notion that digital circuits can be represented in many different ways associated both Boolean equation-type descriptions and subsequent circuit-type descriptions. The notion of "circuit forms" is based on the notion of functional equivalence.

Clark Method: A method used to mitigate the amount of grunt-work required when applying the "New FSM Techniques" to implementing finite state machines (FSMs); (see "New FSM Techniques").

Classical FSM Approach: An approach to implementing finite state machines (FSMs) that uses maximum reduction techniques with every aspect of an FSM implementation. The classical FSM approach can be tedious and is constrained by the basic limitations of Karnaugh maps. The "New FSM Techniques" can be applied to mitigate some of the constraints of this approach at the cost of "less reduced" Boolean expressions; (see "New FSM Techniques").

Clear Condition: A state of a storage element where the current value is '0'. This is also referred to as a "reset condition"; (see "reset condition").

Clear State: The state of a storage element or a signal where the current value is '0'. This is also referred to as a "reset state"; (see "reset state").

Clear: When used as a verb, this term refers to making the value of a signal or storage element a '0'. This term is synonymous with "reset"; (see "reset").

Cleared: A term referring to the notion that a signal or storage element has been set to '0'. This term is synonymous with "reset"; (see "reset").

Clock Edge: A term that generally refers to an "active" edge (either the rising or falling edge) of a synchronous circuit. Changes in many circuit outputs are typically synchronized to an edge of a clock signal.

Clock Input: A signal that is generally used to synchronize digital circuits. Clock signals are typically periodic.

Clocking Waveform: A term used to describe an attribute of a waveform in that clocking waveforms are generally understood to be periodic in nature; (see "clocking waveforms").

CMOS: An acronym standing for: Complimentary Metal Oxide Semiconductor. Most modern digital integrated circuits are created from transistors made with CMOS technology.

Code Word: A phrase used to refer to a single set of digits that are designated as belonging to a given set of other sets of digits that form a given code.

Code-Word: A term sometimes used to describe the obtainable count values in a counter.

Coding Style: A term that refers to the notion that the syntax rules of a language allow you to write viable code that can have about any form. There are accepted forms of coding style for every language; following these coding styles will make your code more readable and understandable to human readers of your code not unlike yourself.

Combinatorial Logic: Digital logic that does not have memory, or the ability to store the values of bits.

Combinatorial Process: One half of a two-process approach to modeling finite state machines (FSMs) using VHDL; the other half of the FSM model is the "synchronous process"; (see "synchronous process"). The combinatorial process is responsible for modeling both the "next state decoder" and the "output decoder" in the standard FSM model; both of these decoders are generally implemented using combinatorial circuits.

Combining Theorem: One of the basic theorems associated with Boolean algebra. This theorem facilitates the use of Karnaugh Maps to reduce Boolean functions. This theorem is sometimes referred to as the "Adjacency Theorem".

Comments: A term that refers to text appearing in code that is ignored by the compiler or synthesizer. Comments in VHDL are designated by two consecutive dashes; all text after these dashes is ignored by the entity interpreting your code. Comments are generally used to explain portions of code that are not patently obvious to provide history-type information regarding the particular file.

Compact Maxterm Form: A form that describes a Boolean function by listing the truth table entries that have outputs of '0' in terms of the decimal index into the associated with that particular row of the truth table. This form uses the capital PI summation signal.

Compact Minterm Form: A form that describes a Boolean function by listing the truth table entries that have outputs of '1' in terms of the decimal index into the associated with that particular row of the truth table. This form uses the capital Greek summation signal.

Comparator: A digital device that compares two signals and determines whether they are equal or not; the two signals can be either single signals or bundles. Comparators are typically referred to as "n-bit comparators which indicates the width of the input signals; outputs of comparators typically include information about the two inputs such as equality, less-than, and/or greater-than. Comparators are one the standard digital circuits used in digital design.

Complementary Outputs: A term used to describe two outputs of a circuit that always represented inverted versions of each other. The various flavors of flip-flops typically have complementary outputs.

Complex Programmable Logic Device: An integrated circuit that can be configured to implement various logic functions or relatively small digital systems. Generally referred to as a CPLD, complex programmable logic devices are generally less complex and less flexible than FPGAs.

Complexicated: A term applied to items that are both complex and complicated.

Component Declaration (VHDL): The notion of making a it known to the VHDL synthesizer that an external design unit, or component, will be used in a particular design.

Component Instantiation (VHDL): The notion of including a previously declared design unit into the current level of design by properly referencing and mapping the design unit.

Component Mapping (VHDL): A term referring to the notion of connecting the signals associated with an instantiated design unit from a different level of a design to the signals in the current level of the design.

Computationally Expensive: A term that describes the notion of there being a "cost" associated with computer operations. All operations performed by digital circuits require a given amount of time to complete, but not all operations are equivalent. For example, it is more computationally expensive to generate the square root of a number than it is to decrement a number; this is due to the fact that the square root operation will require more steps to complete than a decrement based on the application of an underlying algorithm used to implement the two operations.

Computer Aided Design (CAD): The act of using a computer to automate or simplify the design process. Or, a design that is in some part completed by use of a computer and associated software.

Computer: Any electronic device that reads instructions from memory and carries out those instructions on data. A given circuit can officially be labeled a compute if it has the three main components of a computer: memory, CPU, and I/O.

Concurrency: The notion of two or more things occurring at the same time. Concurrency is one of the underlying factors in VHDL in that many of the statements in VHDL are interpreted as being concurrent in that they can describe multiple hardware entities that work in parallel, and thus supporting the concept of parallelism.

Concurrent Signal Assignment: A term that refers to four types of statements in VHDL that in interpreted as occurring at the same time. The four types of concurrent signal assignments, or CSA, are signal assignment, selective signal assignment, conditional signal assignment, and process statements.

Configurability: The ability of a device to select one of several pre-set options as to internal and/or external operations of the device.

Conspicuous Consumption: A term coined by Thorstien Veblem that describes the pecuniary motivations of modern society.

Control Signals: These are signals represented as outputs from a controlling device and as inputs to a device being controlled. Finite state machines (FSMs) are typically used as controllers and contain both control outputs and status inputs.

Control Tasks: A set of functionality that performs a specific set of duties and can be described independently of other sets of functionality; these sets of functionality are designed to perform the duties of controlling specific entities. In terms of digital design, control tasks are typically imeplemetned with microcontrollers or finite state machines (FSMs).

Control Unit: A term describing one of the sub-modules of a central processing unit (CPU). The control unit is generally responsible for controlling the sequencing of processing associated with the datapath in order to obtain the desired result.

Controller: A circuit that is used to control another circuit. Controller circuits generally have both status inputs (status signals) that allow the controller to know the state of the circuit it controls and control outputs (control signals) which are used to directly control some external circuitry. Finite state machines (FSMs) are typically used as controller circuits.

Count Enable: A signal used to allow a counter to count when asserted or disable counting when not asserted.

Counter Design: The notion of designing a sequential circuit that represents a counter. Finite state machines (FSMs) are often used to design simple counters; more complex counters can typically be easily modeled in VHDL.

Counter Overflow: The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its largest representable value to its smallest value..

Counter Underflow: The notion of a counter being decrement beyond its ability to represent values; unless otherwise stated, underflow is generally characterized as the counter transitioning from its smallest representable value to its largest representable value.

Counter: A sequential circuit that progresses through a repeatable sequence of code words. Changes in code words are typically synchronized to a system clock signal. Counter types are generally characterized by the code words in their sequence; typical counter types include binary counters, decade counters, Johnson counters, etc.

Counter: A sequential digital device that has outputs that represent numbers in an arbitrary sequence. Counters are not necessarily associated with a given sequence of numbers such as a binary counter. Counters often have auxiliary inputs to support extra features on the counter such as pre-sets, clears, and parallel loads. Counters are typically described in terms of "width" which refers to the number of single-bit storage elements used to represent the given numbers in the associated count.

Covered: A term that refers to certain aspects of Karnaugh mapping. For generic K-maping, the term refers to the notion of including all "1's" or all "0's" into grouping. In the context of static logic hazard removal, the term "covered" refers to the notion that all groupings that are a unit-distance apart share a grouping in order to provide a grouped path between those groupings.

CPLD: An acronym for "complex programmable logic device"; (see "complex programmable logic device").

CPU: An acronym referring to the "central processing unit"; (see "central processing unit").

CSA: An acronym referring to "concurrent signal assignment; (see "*concurrent signal assignment*").

Cut and Paste Engineer: An engineer that understands using previous work via the "cut and paste" feature of a computer is an efficient way of working. Cut and paste engineers are also known as "CPEs".

Cycles per Second: A term used to describe the number of times a signal changes state in a time span of one second. Cycles per second is synonymous with the term "Hertz"; (see "Hertz").

-D-

D Flip-flop: A shorthand notation for a "data flip-flop"; (see "data flip-flop").

Daisy Chain: A term referring a configuration of multiple digital devices; devices in a daisy chain configuration are placed in a series-type configuration. This term is often referred to as a "cascading".

Data Flip-flop: A flip-flop that changes the output state when the "data" input to the flip-flop is at a different value than the output of the flip-flop and an active edge occurs on the clocking input the circuit. The "next state" of a D flip-flop is a function of the D input only.

Data Inputs: These are the signals on a MUX that can appear on the MUX's outputs. The MUX will choose between one of the data inputs to be the output of the MUX.

Data Selection Inputs: The signal on a MUX that are used to determine which of the MUX's data inputs will appear on the MUX output.

Dataflow Model: A type of VHDL model that refers to the notion that the synthesized logic associated with the model is somewhat evident from the VHDL statements used in the model. If a given VHDL model contains no process statements, the model is considered a dataflow model.

Dataflow Style: A term that refers to the use of dataflow modeling in VHDL.

Datapath: A term describing one of the main submodules of a central processing unit (CPU). The datapath handles the crunching of numbers including mathematical and logic-type operations. The main component in the datapath is the arithmetic logic unit (ALU).

Debugger: A tool used to remove errors from hardware of firmware designs. Debuggers are generally associated with software and firmware development, but they are appropriately can be used to debug circuit designs as they often need help also.

Debugging: The act of removing errors from designs including hardware, firmware, and software.

Decade Counter: A counter that counts in a binary coded decimal (BCD) sequence.

Decimal: A number system that uses ten symbols to represent quantities. These symbols are typically '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9'.

Declarative Region(VHDL): The region of a VHDL architecture where intermediate signals and other items required by the given architecture are listed.

Declarative Region: A part of a VHDL architecture statement that lists items such as component declarations and intermediate signal.

Decoder: A combinatorial (or non-sequential) digital device that establishes a functional relationship between the device input(s) and output(s). There are two general types of decoders: generic decoders and standard decoder. Standard decoders are a subset of generic decoders.

Decrement: An operation typically associated with counters where '1' is subtracted from the current value being stored by the counter.

DeMorgan's Theorem: One of the basic theorems in digital design; this theorem is used to simplify circuits, generate other function forms, and design advanced bowling balls.

DeMorganize: A verb that refers to the act of applying DeMorgan's theorem.

Dependent PS/NS Style: One of many approaches to modeling finite state machines (FSMs) using VHDL. The hallmark of this approach is to have one process that generally handles the combinatorial features of the FSM (see "combinatorial process") while the other process handles issues involved in modeling the state registers (see "synchronous process").

Dependent Variable: A variable is influenced by other variables (namely the independent variables) and is subject to change based on changed in other variables. In digital design, the dependent variable is typically the output while the independent variable is typically the input; thus, dependent variables are a function of independent variables.

Design Automation: A term used to refer to a design approach that reduces the grunt work associated with a given design process.

Design Under Test: A term referring to the VHDL model that is being verified by a VHDL testbench. The "device under test" is a model of a digital circuit that is instantiated as part of the testbench model. The term "device under test" is sometimes referred to as the "unit under test" or the "model under test".

Device Verification: The act of verifying that a device is working as expected. Often times verification means one type of device implementation is used to verify the associated device model is working as expected.

Digit: A symbol used in a number system.

Digital Design: The creation of digital circuits to solve problems. And a somewhat longer version: The creation of a digital circuit that establishes a structured relationship between the circuit's inputs and outputs in such a way as to solve a given problem.

Digital Logic Circuit : A digital logic circuit is an electronic circuit that provides you with some desired result and is implemented using digital logic devices.

Digital Self-Flagellation: A medical term describing the condition associated with performing excessive amounts of digital design.

Digital: A description of a something (such as a signal or data) that is expressed by a finite number of discrete values (or states). These discrete values include the entire "range" of possibilities, but does not include any of the "in-between" values.

Diminished Radix Compliment: A term that refers to a standard but not common method of represented signed binary numbers where the left-most bit in the set of number is considered the sign bit and the other bits are considered the magnitude bits. This term is often listed as DRC.

Dinosaurs: A aptly descriptive term for old (literally or figuratively)

Diode: A two-terminal semiconductor device formed from placing an n-type material on a p-type material, thus forming a "PN junction" which has many delightful characteristics.

Direct Mapping (VHDL): A technique used by VHDL structural models that explicitly links the inputs and outputs of instantiated modules directly to the corresponding inputs and outputs of the next highest level in the hierarchy. The alternative approach to direct mapping is indirect mapping; (see "*indirect mapping*").

Direct Mapping Operator: An operator, "=>", used in VHDL module instantiation to map internal inputs of modules to external connections.

Direct Polarity Indicators: The use of parenthetical values (H) or (L) to indicated the logic level associated with a given signal.

Distance: A term used to characterize the difference between two binary numbers; the distance between two binary numbers is defined as the minimum number of bits of one number that must be toggled in order to equal the second number.

DMUX: A special type of decoder; this term is sometimes used in digital design-land but does not have a solid definition. DMUXes, whatever they are, can be considered a special type of decoder.

Don't Care Transition: A term that refers to a state-to-state transition in a finite state machine (FSM) that occurs independently of any conditions in a given FSM. These transitions are often referred to as unconditional transitions.

Don't Cares: A slang but common term used to describe input combinations associated with Boolean functions as not having specified outputs. This term is derived from the fact that the given input variable combination will never occur so the output does not matter (thus, you "don't care").

Down Counter: A counter that counts only in the "down" direction (count value becomes less)

DPI: An acronym used for "direct polarity indicator"; (see direct polarity indicator).

DRC: An acronym referring to diminished radix compliment; (see "*diminished radix compliment*").

Dumbtarted: A term applied to technical people who go through life with blinders on; these people typically go into management (or administration in a academic setting) due to their complete lack of technical competence and ongoing inability to learn new things.

DUT: An acronym referring to "device under test"; (see "device under test").

Duty Cycle: A term used to describe the percentage of a period that a given signal is in a "high" state. This term always refers to a periodic signal.

Dynamic hazard: A type of hazard associated with the condition where the output is expected to change value (non-static).

Dynamic logic hazard: A type of hazard based on the changing of one input variable (the "logic" part) where the output is expected to change value (the "dynamic" part).

-E-

Enable Signal: A signal that controls the general operation of a circuit in a manner such the circuit outputs are active when the enable signal is asserted and inactive when the enable signal is not asserted.

Engineer: A person who solves problems and strongly shuns worthless administrative tasks.

Engineering Notation: An approach to representing numbers that uses both numerical and exponential parts. The numerical part of the number typically contains both an integral and fractional part. The exponential part of the number is represented as ten raised to powers that are even divisible by three. Often times the exponential portion of the notation is replaced with suffixes that indicate the particular value of three.

Entity (VHDL): The part of a VHDL model that describes the interface of the circuit. The entity is associated with a VHDL architecture.

Enumeration Type: A feature in higher-level computer and hardware description languages and allow users to define their own types in the models they generate. Enumeration types generally allow you to specify how the types are represented internally, but you must explicitly state this desired representation or one will be assigned for you.

Equivalence Gate: Another name for an XNOR gate; see "XNOR gate" for a full definition.

Equivalence Operator: A symbol, "=", used in VHDL to establish a relation between two expressions. This operator is an binary operator that returns a Boolean value which states the two expressions are either equal or not equal.

Equivalent Circuit Forms: A term that refers to the notion that digital circuits can be represented in many different ways but all these ways are functionally equivalent.

Error condition: A condition in a cirucit that is not correct. This may be an ongoing condition such as a bug or a temporary condition such as a glitch. The condition may also be permanent or intermittant.

Error Correction: A reference to the ability to correct one or more errors. Digital circuits can be designed to detect errors, and, if errors are detected, they can correct errors. Error correction circuits generally include "extra" bits along with the "standard" bits (and associated circuitry) in order to detect errors and subsequently correct errors.

Error Detection: A reference to the ability to detect one or more errors. Digital circuits can be designed to detect errors; "parity generators" and "parity checkers" are two common digital circuits used to detect error(s) in a set of bits. Error detection circuits generally include "extra" bits along with the "standard" bits (and associated circuitry) in order to detect errors.

Essential prime implicants: A grouping in a Karnaugh map that is a prime implicant (see "prime implicant") and can be covered in one way only.

Even Parity: A condition that describes a characteristics regarding a set of bits; in particular, whether a set of bits has an even number (or zero) number of bits of value '1'.

Excitation Table: A set of data in a tabular format that describes the operational characteristics of a digital storage element. In particular, excitation tables describe the input conditions required to attain a given state change.

EXNOR Gate: A less common name for an XNOR gate; see "XNOR gate" for a full definition.

EXOR Gate: A less common name for an XOR gate; see "XOR gate: for the full definition.

Expression: A set of items such as variables and constants that are combined via operators according to a known set of rules and used to generate another value by the process of evaluation of the expression.

-F-

Factory Programmed: A term referring to a device that contains connections that are made (or not made) on the silicon level; mask programmability is often referred to as "*factory programmed*" as it is generally done at the associated fab (IC fabrication facility). An

Falling Edge: A "1→0" transition of a given signal that is typically used to synchronize some other action in a circuit. ..

Falling-Edge Triggered: A term used to describe the notion that changes in a circuit are synchronized to a "falling edge" of some signal in the circuit. This term is often abbreviated as "FET".

Fast Division: A term describing a circuit that performs a division operation in a relatively fast manner. Shift registers are widely known for the ability to perform fast division (right shifting) at the cost of including a truncation in the operations.

Fast Multiplication: A term describing a circuit that performs a relatively fast multiplication operation. Shift registers can typically perform fast multiplication operations (left shifting) at the cost of a loss of precision on the lower order bits due to the fact that 0's are stuff in the lower order bits. Fast multiplication in shift registers are limited to multiplying by powers of two.

FET: An acronym referring to "falling-edge triggered"; (see "falling-edge triggered").

Field Programmable Gate Array: A logic device that can be programmed to implement many aspects of a digital circuit. Usually referred to as FPGAs, these devices can be quite large and complex on a low level. Modern FPGAs have complex architectures and include standard internal devices such as memory, CPUs, specialized arithmetic circuits, etc.

Flat Design: A term used to describe VHDL models that do not use a structural modeling approach. Flat designs are inherently non-hierarchical in nature. .

Forbidden State: A condition in a sequential circuit that is generally not allowed to happen to ensure an arbitrary characteristic of that circuit.

FPGA: An acronym for "field programmable gate array"; (see *field programmable gate array*).

Fractional Portion: A phrase referring to the digits on right side of the radix point.

Frequency: The number of times a signal changes state in a given time period. If that time period is one second

FSM Analysis: The act of using a given sequential circuit to generate an associated state diagram.

FSM Design: The act of generating a sequential circuit that can be used to solve a given problem. FSMs can be designed from a word descriptions, timing diagrams, or state diagrams.

Fun Stuff: A synonym widely used for anything having to do with digital design.

Function Forms: A common term used to describe the notion that Boolean expressions or functions can appear completely different but provide equivalent outputs for a given set of inputs.

Function Forms: A reference to the fact that a given Boolean function can be represented in many different ways; each of these ways are considered functionally equivalent. There are many standard function forms out there, two of which are SOP and POS forms. .

Function hazard: A hazard that is present due to the simultaneous changing of two or more input variables for a given circuit.

Function Realization: The notion of "realization" in digital design essentially means that you did something. A function realization would typically be a Boolean equation-based solution to a given problem.

Function: In digital design, a function is an equation that describes an input/output relationship of a module in terms of digital logic.

Functionally Complete: The act

Functionally Equivalent: The condition that exists when various function representations describe the same input/output relationship. This can be thought of as different ways of saying the same thing.

Functionally Equivalent: Two Boolean equation forms that provide the same output for a given set of inputs despite the fact that the equations are different.

Fuse Blowing: A term that refers to the act of removing the connection between two signals. The term "blowing a fuse" means that a previously made connection has been purposely removed. The notion of having fuses is one of the mechanisms that give a hardware device the characteristic of programmability.

Fuse: A term used to describe a temporary connection made between two signals. Fuses can be "blown" or left alone (connection broken or left untouched).

-G-

G: An abbreviation used for the metric prefix "Giga"; this prefix is used in engineering notation.

Gate Array: A generic term used to refer to devices that can be customized for a particular application. This term is generally synonymously used with the term *complex programmable logic device.*

Generic Decoder: One of two types of decoders; generic decoders are generally used to replace the notion of "Boolean functions" by implementing Look-up Tables (LUTs). The term "decoder" is often used in place of the term "generic decoder".

Giga: A standard metric prefix meaning 10^{-9}; the prefix is abbreviated as "G".

Glitch: An temporary unwanted error condition in a circuit. Glitches are typically characterized as low glithces (1-0-1) or high glitches (0-1-0).

Glue Logic: Relatively simple logic present in modular designs that is used connect major sub-modules to other modules.

Gray Code: A type of binary code that is a subset of unit distance codes.

Ground: A term refer to the reference voltage in electronics. In digital electronics, this signal is generally considered a logical '0'.

Ground: A term used to indicate the logic '0' in a digital circuit. In a real circuit, ground is one of the two voltages used to power a circuit. This term is often referred to as "GND" and indicated with a down-pointing arrow in a circuit diagram.

Group of Fours: A phrase used in conjunction with translating binary numbers to a hexadecimal or BCD representation; typically four bits at a time are converted, thus group of "four".

Group of Threes: A phrase used in conjunction with translating binary numbers to an octal representation; typically three bits at a time are converted, thus group of "three".

Grunt Work: Work that you know needs to be done due to its overall importance to your project but is work that is generally boring and requires little intelligence to complete which makes it similar to tasks required of academic administrators.

Guessing: An approach used by simulators to extrapolate an output(s) based on a given input(s) for a particular design.

-H-

Half Adder (HA): A one-bit adder that has outputs for sum and carry-out; the input only include the two bits being added.

Hang States: A state in a state diagram that, once entered, can never be exited. Hang states are generally undesirable conditions associated with finite state machine (FSM) design. Hang states are often associated with self-loops from the given hang state.

Hard-Core Microcontroller: Any "microcontroller" (see "microcontroller") that is not a "soft-core microcontroller" (see "soft-core microcontroller"). Hard-core microcontrollers exist on pre-fabricated integrated circuits as opposed to being synthesizable on programmable logic devices as is the case with soft-core microcontrollers.

Hardware: A term referring to technical entities that are not software or firmware. In the context of digital design, hardware generally refers to digital circuitry in the form of devices synthesized on programmable logic devices (PLDs) or discrete integrated circuits (ICs) on a printed circuit board (PCB).

Hazard: A condition present in a circuit that may under some conditions cause an unwanted condition, or error condition, in that circuit.

Hertz: A measure of frequency defined to be the number of time a signal changes state in a time span of one second. This term is abbreviated as "Hz".

Hex: A shorthand notation for hexadecimal; also a synonym for numbers with a radix of 16.

Hexadecimal: A term used to describe numbers with a radix of 16.

Hierarchical Design: An approach to digital design that utilizes various levels of abstraction in order to promote efficient design and understandable designs.

Hierarchical Design: Designs that are described at multiple levels. The notion of VHDL structural modeling is a mechanism that supports hierarchical design.

High-Impedance: A term that indicates the value of a signal is not being "driven" by some entity in the circuit. When a signal is not being "driven", there is not current flowing in the physical implementation of that signal. No current flowing indicates a "broken circuit". If a device is in a high-impedance state, the device is figuratively not in the circuit.

Hold Condition: A condition in a sequential circuit where the output does not change state when given the proper opportunity; same as "hold state".

Hold State: A condition in a sequential circuit where the output does not change state when given the proper opportunity; same as "hold condition".

Hold-1 Transition: A feature of a state-change in the context of a single bit where the present state is a '1' and the next state is also a '1'.

Hold Time: An attribute of physical sequential circuits defined as the amount of time circuit's control signals must remain stable after the active clock edge of the circuit.

Horse-Sense: A problem solving approach emanating from the notion that you never stop applying intuition to your solutions even though many solutions can be done by rote. Horse-sense can be figuratively described as taking a few steps back and examining your approach before you declare your righteousness.

Hybrid FSM: A finite state machine (FSM) that contains both Mealy and Moore-type outputs.

Hz: An abbreviation typically used for "Hertz"; (see "Hertz").

-I-

Identifier: A set of symbols used by a language to form a name that is assigned to differentiate between items such as variables, functions, entities, architectures, and bowling balls.

If Statement: A type of sequential statement in VHDL, also known as a conditional statement. "if" statements can appear in the body process statements are and typically used in behavioral descriptions of digital circuits.

Illegal State Recovery: The notion associated with finite state machine (FSM) design in that if the FSM finds itself in a state that it is not intended to be in, the FSM has a built-in method to exit that state and return the FSM to an expected state. Illegal state recovery design generally requires more hardware but will avert the death of an FSM by avoiding hang states.

IMD: An acronym referring to "iterative modular design"; (see "iterative modular design").

Inactive State: A term used to indicate that the current voltage level of a signal, or state, is not associated with the active state of that signal.

Inclusive OR Gate: The actual name for a simple OR gate. This name is related to the fact that there is another gates referred to as an "exclusive OR" gate (XOR).

Incompletely Specified Functions: Boolean functions that do not have an output specified for every possible input combination. The main aspect of this type of function is that there are "don't cares" associated with the outputs of those particular input combinations.

Increment: An operation typically associated with counters where '1' is added to the current value of counter.

Indentation: A set of white spaced used to differentiate related sub-areas of computer programming or hardware design code. Proper use of indentation increases the readability and understandability of text-based code; general rules for indentation are found in style-files associated with the language.

Independent PS/NS Style: One of many approaches to modeling finite state machines (FSMs) using VHDL.

Independent Variable: A variable representing a value that can change and thus affect the dependent variable. In digital design, the independent variable is typically the input while the dependent variable is typically the output.

Indirect Mapping (VHDL): A technique used by VHDL structural models that links the inputs and outputs of instantiated modules to the corresponding inputs and outputs of the next highest level in the hierarchy via a list of signal names. The connections are implicit and based upon the order the signal appear in the associated entities. The alternative approach to direct mapping is indirect mapping; (see "*indirect mapping*").

Indirect Subtraction by Addition: An algorithm that performs subtraction by first changing the sign of the augend and adding it to the addend. The advantage of this approach is that changing the sign of a binary number is not complicated and the hardware associated with the addition operation (an adder) can also be used to perform subtraction.

Initial State: Problems dealing with sequential circuits must be provided with the values being stored by the memory elements in the circuits; the initial values are referred to as the "initial state" of the circuit.

Instance (VHDL): A term that refers to an instantiated design unit appearing in the statement region of a VHDL architecture.

Integer-Based Math: A form of mathematics performed on digital devices that is considered faster than alternatives such as using floating point math. The speed of integer math comes at the cost of lower precision in the results, which is acceptable for many applications.

Integral Portion: A phrase referring to the digits on the left side of the radix point.

Integrated Circuit (IC): A piece of semiconductor that include a complete circuit that generally is able to complete some given task. Most ICs are generally packed full of items such as transistor, resistor, capacitors, and inductors.

Interface (specification): A term used to describe VHDL entities because they list the inputs and outputs of a given digital circuit.

Intermediate Signals (VHDL): A term given to signals that are required by a design but do not appear on the list of signals included in the VHDL entity. Intermediate signals are also referred to as "internal signals".

Internal Signals (VHDL): A term given to signals that are required by a design but do not appear on the list of signals included in the VHDL entity. Internal signals are also referred to as "intermediate signals".

Iterative Design: A digital design approach that is based on exhaustively listing all possible inputs and listing a unique output for each of the input combinations. Iterative design is typically based on the use of a truth table.

Iterative Modular Design (IMD): One of the three approaches to performing digital design. The IMD approach uses multiple instances (the iterative part) of pre-defined circuits (the modular part) in digital designs, thus creating hierarchical design. The IMD approach can be used to design some digital circuits and is considered a more powerful approach than "brute force design" in that truth tables and K-maps are typically not part of the IMD process.

-J-

JK Flip-flop: A flip-flop that may change the output state according to when the "JK" inputs to the flip-flop. The JK flip-flop has the ability to hold state, toggle, set, and clear on the active edge of the flip-flop's clock input. The "next state" of a JK flip-flop is a function of both the JK inputs and the present state of the flip-flop.

Juxtapositional Notation: Placing numbers side by side and giving the numbers different weights ; using this notation allows for the representation of more numbers than are present in the set of numbers representing the number system.

-K-

k: An abbreviation used for the metric prefix "Kilo"; this prefix is used in engineering notation.

Karnaugh Map Compression: The act of making Karnaugh maps smaller by translating one or more of the independent variables into map entered variables (MEVs).

Karnaugh Map: A tool that allows for visual application of the adjacency theory to reduced Boolean functions. Karnaugh Maps employ a special number system onto a grid of cells; each cell represents a row in the truth table associated with the given function.

Kilo: A standard metric prefix meaning 10^{-3}; the prefix is abbreviated as "K".

Kludgy: (pronounced "clue-gee")"A term used to describe something that works but is far from being an optimal approach. Electronic circuitry and computer programs often include this term for things that officially to officially work but no one really knows why based on the overall low quality of the design.

K-Map: The shorthand name for "Karnaugh Maps" (see "Karnaugh Map").

-L-

Large Scale Integration: A type of integrated circuit that contains a more transistors than a medium scale integrated (MSI) IC. This term often described with the acronym "LSI".

Latch Generation: A term that refers to the notion of storage being automatically generated by the VHDL synthesizer. The generation of latches is generally not an intended operation as latches are not overly useful and require extra hardware resources to implement. One of the general rules in using a hardware description language such as VHDL is to avoid the unintended generation of latches.

Latch: A term used to describe a sequential circuit that has the ability to store one bit of data. Latches are considered "level sensitive" devices in that they generally always react immediately to circuit inputs.

Leading Zeros: Zeros ('0's) placed in front of (taking up the left-most positions) a given number. Because of the location of these 0's, the do not affect the magnitude of the number being represented.

Learning by Rote: A learning approach typically used by students in order for them to deal with the lack of teaching skills of instructors.

Least Significant Digit: A phrase referring to the digit position with the lowest weighting in a juxtapositional notarized number system.

Legend: A special type of annotation associated with type of visual representation of something. In particular, all timing diagrams, circuit diagrams, and particularly state diagrams should contain legends in order to increase the readability of the diagrams.

Legends In Their Own Minds: A characteristic typically associated with all academic administrators.

Level of Abstraction: The act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances. Particular to digital design is the notion of using black boxes that perform some function but it is not generally known the details of how those functions are implemented at a lower level.

Level Sensitive: A term that refers to the notion that a digital device react to input signals anytime they may change. On the contrary, some circuits are considered edge-sensitive.

Libraries: A storage area for previously designed modules and/or syntactical term definitions required for use in the typical design practice.

Lingo: Special vernacular used in the description of something that only people who typically spend considerable time working with that something actually understand. Lingo is often strongly associated with technical slang.

Local Variables: A type of variable typically found in computer programming languages; local variables are located on the stack and do not have permanent storage.

Lock-Step Process: A set of entities that wait on signal from each other in order to properly sequence their overall operations.

Logic Analyzer: A device that tests a given digital circuit implementation by displaying the state of the digital inputs and outputs at various time interval. Logic analyzers generally have one of two types of displays: timing diagrams and state listing. The timing diagrams are happy timing diagrams; the state listing shows the circuits inputs and outputs at given time intervals or when changes in signals occur.

Logic Gate (or just "Gate"): A physical hardware entity that implements a logic function.

Logic hazard: A hazard that is present due to the changing of a single input variable for a given circuit.

Logic Unit: A term describing one of the main sub-modules of an arithmetic logic unit (ALU). The logic unit generally handles operations that can be considered "logic" such as ANDing and ORing, etc. Logic units are typically assigned to handle shifting and rotation operations also.

Look-Up Table: Also known as LUTs, a structure commonly used in engineering and software applications. In algorithmic programming languages, this term is used to describe the approach of pre-calculating and storing values and referencing the results as needed. In VHDL, LUTs are used to implement many of the standard digital modules.

LSD: An acronym used for least significant digit; (see least significant digit).

LSI: An acronym for "large scale integration"; (see "large scale integration").

-M-

M: An abbreviation used for the metric prefix "Mega"; this prefix is used in engineering notation.

m: An abbreviation used for the metric prefix "mili"; this prefix is used in engineering notation.

Macrocells: A sub-block of a PLD that can be both programmable and/or configurable. This term is basically used to describe the architecture of PLDs.

Magnitude Bits: The portion of a set of bits that refers to the magnitude portion of the number being represented by the set of bits. Signed binary number representation always have both magnitude bits and a sign bit

Manual Verification: A term that refers to the notion of a VHDL testbench's that does not "automatically" verify the proper operation of a VHDL model. Manual verification requires that the user examine the simulation results in order to determine whether the circuit is working or not.

Map Entered Variable: A variable that appears in a Karnaugh map or truth table where typically only 1's and 0's are entered.

Mask Programmable: A term referring to a device that contains connections that are made (or not made) on the silicon level; mask programmability is often referred to as "*factory programmed*" as it is generally done at the associated fab (IC fabrication facility).

Maximum Clock Frequency: A term that refers to the highest clock frequency a sequential circuit can be clocked and still operate properly. The maximum clock frequency of a circuit is based on physical attributes of the devices in the circuit such as setup and hold times

Maxterm Expansion: Another term referring to Standard POS form (see "Standard POS form").

Maxterm: A sum term associated with a given function that includes one instance of every independent variable in the function. Maxterms are associated with conditions that produce a logic '0' on the function's output. A minterm is synonymous with a Standard Sum Term.

MCU: An acronym referring to a "microcontroller"; (see "microcontroller").

Mealy's Fifth Law of Digital Design: Always first consider modeling a digital circuit using some type of a look-up table (LUT); keep in mind that LUTs are implemented in digital design using generic decoders.

Mealy's First Law of Digital Design: if in doubt, draw some black box diagrams.

Mealy's Second Law of Digital Design: if your digital design is running into weird obstacles that require kludgy solutions, toss out the design and start over from square one.

Mealy's Third Law of Digital Design: Don't rely on the VHDL synthesizer; create your VHDL models by having a remote vision of what underlying hardware should look like in terms of standard digital modules.

Mealy's Fourth Law of Digital Design: Model circuits using many smaller sub-modules as opposed to fewer larger sub-modules. In this case, sub-modules should be true "modules" but can also be process statements.

Mealy-Type FSM: A class of finite state machine (FSM) that is characterized by having outputs that are a function of both the present state of the FSM and the external inputs to the FSM. Mealy-type FSMs are typically modeled as having a "next state decoder", "state variable storage", and an "output decoder".

Mealy-type Outputs: An external output to a finite state machine (FSM) that exhibits Mealy-type qualities; Mealy-type qualities refer to the notion that the external output is a function of both the current state of the FSM and the values of the external inputs to the FSM.

Medium Scale Integration: A term that roughly refers to the number of transistors on an integrated circuit. The exact number of transistors associated with medium scale integration is not really quantifiable; medium scale integration is generally know as the next step beyond small scale integration; usually referred to as MSI.

Mega: A standard metric prefix meaning 10^{-6}; the prefix is abbreviated as "M".

Memory Element: A digital device that is capable of storing an arbitrary number of bits. Memory elements are typically associated with state variable storage in finite state machines (FSMs). Memory elements are often referred to as "storage elements".

Memory Inducing: A term used in the context of using VHDL to model memory elements in digital circuits.

Metastable: A term referring to an unwanted condition in a sequential circuit resulting from not meeting the setup and/or hold times of that circuit. This term is sometimes referenced as "metastability".

MEVs: An acronym used to refer to map entered variables; see "map entered variables".

micro: A standard metric prefix meaning 10^{6}; the prefix is abbreviated as "μ".

Microcontroller: A digital device that is a complete computer on a signle integrated circuit. Being complete computers (by definition of a computer), microcontollers contain an arithmetic logic unit (ALU), a finite amount of memory (for both data and instructions) and input/output capabilities (in order to interface with the outside world). Microcontrollers are programmable at various levels including higher-level languages and assembly languages. Microcontrollers typically control other digital and/or analog devices.

mili: A standard metric prefix meaning 10^3; the prefix is abbreviated as "m".

Minimum period: A term that refers to the smallest period of a clock signal associated with a sequential circuit can be clocked and still operate properly. The minimum period of a circuit is based on physical attributes of the devices in the circuit such as setup times

Minterm Expansion: Another term referring to Standard SOP form (see "Standard SOP Form").

Minterm: A product term associated with a given function that includes one instance of every independent variable in the function. Minterms are associated with conditions that produce a logic '1' on the function's output. A minterm is synonymous with a Standard Product Term.

Minuend: A number from which another number is subtracted.

Mixed Logic Design: A digital design that contains signals in both negative and positive logic representations.

Mixed Logic: A term referring to the notion that a given circuit or system uses both positive and negative logic.

Models in Digital Design: a model is a representation or a description of something using a certain level of detail. The main purpose of the model in digital design is to transfer information to the entity using the model. There are four main types of models used in digital design: black box model, timing diagrams, written descriptions of digital circuits, and VHDL models.

Modern Digital Design: Modern digital design is truly design oriented as opposed to historical approaches which were not designed oriented due to the unavailability of implementation tools. Modern digital design is driven by Hardware Description Languages such as VHDL and Verilog. The availability of HDLs and the relative low cost of PLD-based hardware allow digital designs to be implemented and tested significantly more quickly than historical design techniques.

Modular Design: A design technique that primarily utilizes pre-defined black boxes (or modules) as the basis of the design. This design approach in one of the three approaches to digital design and is considered the most powerful and efficient approach. Modular designs are generally hierarchical in nature.

Moore-Type FSM: A class of finite state machine (FSM) that is characterized by having outputs that are a function of the present state of the FSM only. Moore-type FSMs are typically modeled as having a "next state decoder", "state variable storage", and an "output decoder".

Moore-type Outputs: An external output to a finite state machine (FSM) that exhibits Moore-type qualities; Moore-type qualities refer to the notion that the external output is exclusively a function of the current state of the FSM.

Most Significant Digit: A phrase referring to the digit position with the highest weighting in a juxtapositional notarized number system.

MSD: An acronym used for most significant digit; (see "*most significant digit*").

MSI: An acronym for "medium scale integration"; (see "medium scale integration").

Multiplexor: A standard digital device used to select between a set of two or more signals. Multiplexors generally have data input, data selection inputs, and data outputs. Most often multiplexor's have a binary-type relationship between data selection inputs and data inputs; the characteristic is sometimes used to provide a standard name to the multiplexor such as "2:1", or "4:1", or "8:1" MUX, etc.

MUX: A shorthand term that refers to a "multiplexor"; (see "multiplexor").

-N-

n: An abbreviation used for the metric prefix "nano"; this prefix is used in engineering notation.

NAND Gate: One of the standard logic gates; a NAND gate performs an AND function with a complimented output. A different way to model a NAND gate is an AND gate with an active low output. NAND gates can have two or more inputs.

NAND Latch: A sequential circuit comprised on two NAND gates connected such that they have the ability to store one bit (the circuit contains feedback). NAND latches are considered the negative logic version of NOR latches.

NAND/AND Form: One of the basic eight logic forms but not commonly used in digital design. This form is derived from OR/AND form (POS form) by excessive use of DeMorgan's theorem.

NAND/NAND Form: One of the basic eight logic forms and one of the most popular four ways to describe a circuit using either Boolean equation or the circuit model of the associated Boolean equation. This form is directly related to the AND/OR form but is comprised of exclusively NAND functions (for the Boolean equation) or NAND gates (for the circuit representation).

nano: A standard metric prefix meaning 10^9; the prefix is abbreviated as "n".

Native VHDL Type: A "type" that is provided by the particular distribution of VHDL. VHDL has many native types but also allows you to create you own types by also including the notion of "enumeration types"; (see "enumeration types").

N-bit Adder: A term used to describe the number of bits in the operands and/or result of a circuit that performs addition.

N-bit Counter: A counter that uses "n" bits (n is an integer) to represent each value in its sequence of values.

N-bit Register: A register that can store "n" bits (n is an integer).

Negative Logic: A term used to indicate that a given circuit considers the notion of '0' to be the active level for the signals in that circuit.

Negative Logic: A term used to indicate that the '0' state of a signal represented the active state of that signal.

New FSM Techniques: A set of techniques applied to finite state machine (FSM) implementation the removes the need for using Karnaugh map and thus allows for the implementation of more complex FSMs. One important characteristic of new FSM techniques is that the resulting equations are not necessarily in reduced form as they are with "classical FSM techniques"; (see "classical FSM techniques").

Next State Decoder: A combinatorial digital circuit that is typically used in the modeling of finite state machines (FSMs). The primary function of the next state decoder is to provide excitation logic to the storage elements (generally flip-flops) associated with the FSM.

Next State Forming Logic: This is less common term that refers to the "next state decoder" typically associated with a finite state machine (FSM); (see "next state decoder").

Next State Logic: A term referring to the combinatorial circuitry that makes up the "next state decoder"; (see "next state decoder").

Next State: The notion that a given sequential circuit has the ability to change the value of the bits it is currently storing at a later time. This term is generally combined with "present state" to describe the operation of sequential circuits.

Noise: A term referring to an undesired transition (either "0→1" or "1→0") in the value of a signal. In digital design, a standard form of noise is a "glitch"; (see "glitch").

Nonessential prime implicants: A type of prime implicant that is not necessary to include when generating the minimum covering in a Karnaugh map function reduction.

Non-Resetting Sequence Detector: A "sequence detector" (see "sequence detector") that can use parts of previously detected sequences in its current search for the next sequence.

Noob: A slang description of a very special person.

NOR Gate: One of the standard logic gates; a NOR gate performs an OR function with a complimented output. A different way to model a NOR gate is an OR gate with an active low output. NOR gates can have two or more inputs.

NOR Latch: A sequential circuit comprised on two NOR gates connected such that they have the ability to store one bit (the circuit contains feedback). NOR latches are considered the positive logic version of NAND latches.

NOR/NOR Form: One of the basic eight logic forms and one of the most popular four ways to describe a circuit using either Boolean equation or the circuit model of the associated Boolean equation. This form is directly related to the OR/AND form but is comprised of exclusively NOR functions (for the Boolean equation) or NOR gates (for the circuit representation).

NOR/OR Form: One of the basic eight logic forms but not commonly used in digital design. This form is derived from AND/OR form (SOP form) by excessive use of DeMorgan's theorem.

Not Asserted: The notion that the current state of a signal (or voltage level) is associated with the non-action state. Whether a signal is asserted or not is independent of the logic level (negative or positive) associated with that signal.

N-type: A semiconductor that has been doped with material containing extra electrons.

Number System: a language system consisting of an ordered set of symbols (called digits) with rules defined for various mathematical operations.

Number: a collection of digits; a number can contain both a *fractional* and *integral* part.

-O-

Object Oriented: A design approach that partitions system entities into objects. For digital design, these objects are considered black boxes or modules.

Object-Level Design: Designs that utilized previously designed objects. In digital design, these objects are generally previously designed black boxes.

Octal: A term used to describe numbers with a radix of 8.

Odd Parity: A condition that describes a characteristics regarding a set of bits; in particular, whether a set of bits has an odd number of bits at a value of '1'.

Old Dude: A person that is characterized by being impatient, arrogant, and condescending to those who may know less they do (but usually don't); many dinosaurs in academia fall into this category. This term has nothing to do with age as anyone can adopt this set of counter-productive attitudes.

One's Compliment: An operation that can be performed on a binary number; taking a 1's compliment of a binary number entails toggling the value of each bit in the number.

One-Cold Encoding: A term that refers to on of many different methods used to encode the state variables associated with the various states in a finite state machine (FSM). In particular, one-cold encoding uses one storage element for each state in the associated FSM. The codes applied to states have ensures that only one storage element is a '0' in any given state; while all other storage elements are '1'.

One-Hot Encoded: One of many methods typically used to encode the state variables associated with a finite state machine (FSM). The one-hot encoding method uses one 1-bit storage element for each state in the given FSM; at any one time (thus in any given state), only one of the state variables are at a '1' values while all the other state variables are at a '0' values.

One-Hot Encoding: A term that refers to on of many different methods used to encode the state variables associated with the various states in a finite state machine (FSM). In particular, one-hot encoding uses one storage element for each state in the associated FSM. The codes applied to states have ensures that only one storage element is a '1' in any given state; while all other storage elements are '0'.

On-The-Fly: A term that refers to one method of accessing test vectors in a VHDL testbench. This term basically refers to the notion that the test vectors for a given testbench are hard-coded as part of that test bench. Other testbench options for accessing test vectors are reading from hard-coded arrays or reading from external files.

Open-Circuit: A circuit condition that describes a lack of connection between two signals.

Operator Precedence: A set of pre-defined rules that establish the execution order of operators associated with program or model code.

OR Plane: A structured array of logic that allows for the combination of Boolean variables and/or function outputs in such a way as to form sum terms used to implement other Boolean functions.

OR/AND Form: One of the basic eight logic forms and one of the most popular four ways to describe a circuit using either Boolean equation or the circuit model of the associated Boolean equation. This form is often referred to as "product of sum" form or POS form.

OR/NAND Form: One of the basic eight logic forms but not commonly used in digital design. This form is derived from AND/OR form (SOP form) by excessive use of DeMorgan's theorem.

Output Decoder: A combinatorial digital circuit that is typically used in the modeling of finite state machines (FSMs). The primary function of the output decoder is to

Overflow: A condition that indicates the result of a mathematical operation has exceeded the top end of the range of numbers associated with the bit-width of the operands. Overflow is often considered to include underflow; (see "*underflow*").

-P-

PAL: An acronym for "programmable array logic"; (see *programmable array logic*).

Paper Design: A design that is done only on paper with no intention of every actually implementing the design. Such designs are proven to work with only violent hand-waving arguments. Such designers generally end up as administrators as their hand waving arguments are backed up by their innate intimidation tactics.

Parallel Inputs: A term referring to an input that simultaneously acts on a set of entities. In particular, a parallel input to the state variables of a finite state machine (FSM) act on all the individual storage elements in a simultaneous manner.

Parallel Load: A characteristic of a register indicating that all the storage elements in the device can simultaneously latch external values.

Parallel: A condition that describes a set of multiple items considered all at the same time.

Parallelism: The notion of doing two or more things at the simultaneously, particularly in the state of engineering, computer science, and bowling. .

Parenthetical Bundle Indexing: Because bundles contain more than one signal, the name of the bundle needs to be modified in order to reference the individual signals in the bundle. There many ways to do this but this notation is the most common. This notation assumes that indexes from zero to one less that the number of signals in the bundle will be used with the index with the highest number being the most significant bit in the signal.

Parity Bit: A bit included and/or associated with a set of bits that indicates whether those bits exhibit the condition of "even parity" or "odd parity". The parity bit can also be viewed as being able to give a set of bits either even or odd parity by including the parity bit with the set of bits being considered.

Parity Checker: A digital circuit that is used to verify that a circuit has either "odd" or "even" parity. The parity checker is one the standard digital circuits used in digital design.

Parity Generator: A digital circuit that generates a bit that is associated with a set of bits that describes the parity of those bits (either "odd" or "even" parity). The parity generator is one the standard digital circuits used in digital design.

Parity: A word used to describe a condition associated with a set of bits. The given set of bits can be in a parallel configuration (parity considered at one point in time over more than one signal) or a serial configuration (parity considered over a span of time for one signal). The notion of parity provides information regarding the number of bits at a value of '1' in a given set of bits.

Period: The amount of time a given signal requires before it repeats itself.

Periodic: A term used to describe an attribute of a signal. A periodic signal is defined as having a set period that repeats itself ad

Periodic Waveform: A term used to describe an attribute of a waveform. Periodic waveforms are generally used as clocking signals for sequential circuits and often referenced as "clocking waveforms"; (see "clocking waveforms").

Pin Count: A term referreing the number of external pins on the integrated circuit. This term usually refers to the number of pins used for input/output requirements of the device. The main issues here are that the cost of a specific device increases as the pin count increases.

PLA: An acronym for "programmable logic array"; (see *programmable logic array*).

PLC: An acronym representing the "positive logic convention"; (see positive logic convention).

PLD: An acronym for "programmable logic device"; (see *programmable logic device*).

Positive Logic Convention: An approach to representing mixed logic that uses overbars on signals to indicate negative logic and no overbars to represent positive logic.

Positive Logic: A term used to indicate that the '1' state of a signal represents the active state of that signal.

Present State: The notion that a given sequential circuit is currently storing a given value but that value can change to a new value. This term is generally combined with "next state" to describe the operation of sequential circuits.

Prime implicants: A grouping in a Karnough map that cannot be completely convered by any other single grouping.

Process Body: A part of a VHDL process statement that include the declarative region of and the statement region of a process statement.

Process Statement: A type of concurrent statement in VHDL used in behavioral modeling.

Product of Sums (POS) Form: A function form that is characterized by sum terms that are logically multiplied together. .

Product Term: A set of Boolean variables that are ANDed or logically multiplied together.

Product Term: An expression in a Boolean equation that can be characterized as a logical multiplication of variables.

Programmable Array Logic: A type of programmable logic device characterized by having a programmable AND plane and a non-programmable OR plane.

Programmable Logic Array: A type programmable logic device characterized by having both programmable AND plane as well as a programmable OR plane.

Programmable Logic Device: An integrated circuit that can be configured to implement various logic functions and/or digital systems. Generally referred to as a PLD, a programmable logic device covers the entire class of programmable logic devices including FPGAs, PLAs, PALs, and CPLDs.

Prop delays: A shorthand version of "propagation delay"; see "propagation delay".

Propagation delay: The time delay associated with the propagation of a signal through an electronic circuit. Propagation delays are generally associated with phyical aspects of the ciruit and are inherent in all electronic devices to one degree or another.

Proto-Board: A device used for prototyping electronic circuits; the proto-board is comprised of many tiny holes in which the stripped end of a wire was pushed into in order to make an electrical connection. The integrity of proto-boards diminishes over time as the actual connections as based on the elastic properties of some very tiny pieces of metal.

Protocol: A pre-defined set of rules that describe a mechanism that digital entities can use to communicate with each other. Any entity that complies with the protocol can communicate with any other entity also in compliance with the protocol.

PS/NS Table: A set of data in tabular format that describes the operational characteristics of a sequential circuit. The acronyms PS & NS are short-hand notation for "present state" and "next state", respectively. The information in PS/NS tables can be visually represented using "state diagrams"; (see "state diagrams").

P-type: A semiconductor that has been doped with material containing extra holes (or lack of electrons).

-Q-

Q: The letter typically used to refer to the "state" of a single bit storage element. In terms of finite state machines (FSMs), this term refers to the present state.

Q+: The term typically used to refer to the "next state" of a single bit storage element used in a finite state machine (FSM)

-R-

Radix Compliment: A term referring to a standard and most common method of representing signed binary numbers. The left-most bit in a number in radix complement form is considered the sign bit. This form

Radix Point: a symbol used to delineate the fractional and integral portions of a number.

Radix: the number of digits in the ordered set of symbols used in a number system.

Rapid Prototyping: The ability to quickly generate a working model of a device that exhibits the functionality of the expected final device.

RC: An acronym referring to radix compliment; (see "*radix compliment*").

RCA: An acronym referring to a ripple carry adder; see "ripple carry adder" for details.

RCO: An acronym referring to "ripple carry out"; (see "ripple carry out").

Redundant State: A state in a finite state machine (FSM) that is not essential to the overall operation of the FSM. While technically correct, we typically omit redundant states in FSMs because they represent basic inefficiencies in FSM specification.

Register: A register is can store two or more bits of data. There are many types of registers including shift registers and counters; when the term "register" is used, it typically refers to "simple registers" and not counters and shift registers. Registers typically store data on the active edge of an input signal, which is typically a clock edge.

Register: A term referring to a digital circuit comprised of an arbitrary number of single-bit storage elements connected in a manner is that can be modeled as "parallel". Registers are typically described by their "width", or number of associated single-bit storage elements.

Relational Operators: A set of operators used in VHDL conditional statement to determine the relation between two expressions.

Relative Time: A term referring to the notion that any reference to time in a VHDL testbench is based on a previous time reference, as opposed to always the same reference as is one in "absolute time". Relative time references have the characteristic that they "accumulate" through a testbench.

Repeated Radix Division: An algorithm used to convert the integral portion of a number from decimal to any other radix.

Repeated Radix Multiplication: An algorithm used to convert the fractional portion of a number from decimal to any other radix.

Reset Condition: A state of a storage element where the current value is '0'. This is also referred to as a "clear condition"; (see "clear condition").

Reset Pulse: A signal that is used to reset a sequential circuit. This signal is typically short in duration (thus the term "pulse") and can either be a '1' pulse or a '0' pulse.

Reset State: The state of a storage element or a signal where the current value is '0'. This is also referred to as a "clear state"; (see "clear state").

Reset: When used as a verb, this term refers to making the value of a signal or storage element a '0'. This term is synonymous with "clear"; (see "clear").

Resetting Sequence Detector: A "sequence detector" (see "sequence detector") that can't use parts of previously detected sequences in its current search for the next sequence. In other words, when the sequence detector finds the correct sequence, the sequence detector must start looking for the first bit in the desired sequence.

RET: An acronym referring to "rising-edge triggered"; (see "rising-edge triggered").

Ripple Carry Adder (RCA): A digital device that is used to add two digital values. The RCA is comprised of a series of one-bit adder elements that are connected in a series configuration such that the carry from lower-order bits propagates, or "ripples" in the direction of higher-order bits.

Ripple Carry Out: A signal typically found on counters that indicates when the counter has reached its maximum count value. This value is often used in some devices to indicate underflow. This signal often aids in cascading multiple counter devices. .

Rising Edge: A "0→1" transition of a given signal that is typically used to synchronize some other action in a circuit. .

Rising-Edge Triggered: A term used to describe the notion that changes in a circuit are synchronized to a "rising edge" of some signal in the circuit. This term is often abbreviated as "RET".

Rotates: A specialized shift-type operation often associated with shift registers characterized by shifting all bits in the register in one direction (either left or right) and replacing the MSB by the LSB (rotate right) or the LSB by the MSB (rotate left).

Routing: The act of physically connecting two entities. This term is often used in the context of printed circuit board development and PLD architectural/implementation issues.

RRD: An acronym used for repeated radix multiplication; (see "repeated radix division").

RRM: An acronym used for repeated radix multiplication; (see repeated radix multiplication).

Rubylith: Some red plastic stuff that was used to fabricate integrated circuits in the early days of IC design and manufacturing.

-S-

Scalar: A term used to signify that a given item cannot be sub-divided into sub-items.

Secret Sauce: A term that describes the notion that there is something not being told to you or provided for you. In free software distributions, often times the vendor removes the secret sauce from the free version of the software and only provides it for those who have the wherewithal to shell out the big bucks.

Selective Signal assignment: A type of concurrent statement used in VHDL; selective signal assignment statements are analogous to the case statement in VHDL behavioral modeling.

Self-commenting: A phrase referring to the notion of using the names of items (such as signals and black-boxes, variables, etc.) to indicate what the probable purpose of those items.

Self-Commenting: The use of identifiers (see "*identifier*") that given the human reader an idea as to the purpose or functionality of a particular items such signals, entities, architectures, variables, etc.

Self-Correcting: A term that refers to the notion that a finite state machine (FSM) has the ability to return to a desired state in the event that it finds itself in an undesired or unused state. The notion of self-correction must be intentionally designed into the FSM by the associated digital designer.

Self-Loop: A condition in a finite state machine (FSM) indicating a state transition from a particular state returns to that state in one state transition. This condition can also be viewed with the notion that the FSM never actually exited that given state.

Self-Loop: A condition in a state diagram where the state of the sequential circuit does not change when given the opportunity.

Semiconductor: A substance that has an electrical conductivity based on external factors. This term is also used to described specific devices made from semiconductors such as transistors, diodes, etc.

Sensitivity List: A part of a VHDL process statement that shows which signals will case the process statement to be evaluated.

Sequence Detectors: A device that can determine when a specifed binary sequence appears on a given digital signal. Sequence detectors are often implemented using finite state machines (FSMs); such FSM can either be "resetting" or "non-resetting" in nature.

Sequential Logic: Digital logic that has memory, or the ability to store the values of bits. It is generally understood that the ability to store bits comes from the notion of the circuit or an element in the circuit having feedback from an output of the circuit to an input.

Sequential Statement: A type of statement that can appear in a VHDL process statement. Sequential statements are evaluated in the order they appear in the process statement though the process statement itself is a concurrent statement.

Serial Lines: A term that refers to a signal signal that sends or receives a contiguous set of bits over a given time period. We typically refere to "bit-streams" that are received over serial lines; (see "bit-streams").

Serial: A condition that describes a set of multiple items considered one at a time.

Set Condition: A state of storage element where the current value is '1'.

Set or Clear Method: One of the "new FSM techniques" associated with JK flip-flops where expression are written for each state transition that "sets" (0→1) for the J excitation inputs and or "clears" (0→1) for the K excitation inputs (see "new FSM techniques", "special J reduction" and "special K reduction").

Set or Hold-1 Method: A part of the "new FSM techniques" associated with D flip-flops where expression are written for each state transition that "sets" (0→1) or "holds-1" (1→1); (see "new FSM techniques").

Set Pulse: A signal that is used to set a sequential circuit. This signal is typically short in duration (thus the term "pulse") and can either be a '1' pulse or a '0' pulse.

Set State: The state of a storage element or a signal where the current value is '1'.

Set Transition: A feature of a state-change in the context of a single bit where the present state is a '0' and the next state is also a '1'.

Set: When used as a verb, this term refers to making the value of a signal or a storage element a '0'.

Set-Clear Method: A part of the "new FSM techniques" associated with T flip-flops where expression are written for each state transition that "sets" (0→1) or "clears" (0→1); (see "new FSM techniques").

Setup Time: An attribute of physical sequential circuits defined as the amount of time a circuit's control signals must remain stable before the active clock edge of the circuit.

Shift Register Cell: A single bit-storage element that forms the building block of a shift register.

Shift Register: A sequential circuit that is comprised of individual bit storage elements connected in such as way as to facilitate a "shift" operation between elements. The shift operation generally indicates that each storage element in the register simultaneously transfers its value to a contiguous storage element. Shift operations are generally synchronized to a system clock.

Short: A short-hand notion referring to a short circuit; (see *short circuit*).

Short-Circuit: A circuit condition that describes a connection between tow points.

Sign Bit: A bit in a set of bits representing a binary number that is used to signify a sign bit. The sign bit location of the binary number it traditionally the left-most bit in the set of bits.

Sign Magnitude: A term that refers to a standard but not common method of representing signed binary numbers where the left-most bit in the set of numbers is considered the sign bit and the other bits are considered the magnitude bits. This term is often referred to as "SM".

Signals (VHDL): A term that refers to a declaration of internal connections of a VHDL architecture.

Signed Binary Numbers: a set of bits (1's and 0's) that are used to represent a numbers that are either negative, zero, or positive.

Signedness: A term that refers to the notion that a set of bits is a representation of a signed number.

Silicon: The main semiconductor material used in the creation integrated circuits; silicon is the 14th element in the table of elements and is quite plentiful on planet earth.

Simple Register: A device that can store two or more bits of data. A "simple" register is a register that is not a counter or shift register (or various versions of these). Additional features of a simple register include parallel loading and other parallel actions such as clearing and setting.

Simulation: The act of verifying your circuit is working without actually implementing the circuit.

Simulator: A device that tests a given circuit by providing a mechanism to list and/or change circuit inputs and views the resulting changes in circuit outputs. A simulator is a common design and debugging tool.

Slanted T Symbol: A circuit symbol referring to a connection to the value of a '1' in a circuit. Most often, the value of '1' is the voltage value used to provide power to the circuit.

Slash Notation: A graphical representation used in schematics to indicate the number of individual signals contained in a bundle.

SM: An acronym referring to signed magnitude; (see "*signed magnitude*").

Small Scale Integration: A type of integrated circuit that comprises of up to approximately a hundred transistors; usually referred to as SSI.

Soft-Core Microcontroller: A "microcontroller" (see "microcontroller") is modeled using a hardware description language (HDL) and is synthesizeable on a programmable logic controller (PLD).

Sorting: A typical hardware and/or software operation that arranges a set of values based on some pre-determined criteria such as magnitude.

Special J Reduction: A technique associated with "new FSM techniques" and JK flip-flops where expression associated with the J excitation input can be reduced by inspection. In particular, for the case where the complement of the associated state variable is appears in the expression for that variable, it can be removed from the J excitation input based on known characteristics of JK flip-flops and the "set of clear method; (see "set or clear method"). The technique is associated with "0→1" transitions only.

Special K Reduction: A technique associated with "new FSM techniques" and JK flip-flops where expression associated with the K excitation input can be reduced by inspection. In particular, for the case where the associated state variable is appears in the expression for that variable, it can be removed from the K excitation input based on known characteristics of JK flip-flops and the "set of clear method; (see "set or clear method"). The technique is associated with "1→0" transitions only.

Speed-Wrap: An antiquated approach to prototyping electronic circuits. In particular, a wire with a plastic coating was pushed between two posts that had sharp edges. The sharp edges would cut through the plastic and make a connection with the wire.

Spiritually Enriching: A term that refers to the act of performing any of the various aspects of digital design. .

SR Latch: A one-bit storage element with that has a S (set) and a R (reset) input that are used to either set of clear the output of the latch, respectively.

SR: An acronym representing "shift register"; (see "shift register").

SSI: An acronym for "small scale integration"; (see *small scale integration*).

Standard Decoder: A special type of decoder that contains a **n:2^n** relationship between the number of inputs and outputs. The standard decoder is a subset of decoders in general.

Standard Product of Sums Form (Standard POS Form): A description of a Boolean function that includes an explicit listing of the standard product terms that imply a non-active state (0's) on the function's output. Standard POS form is also referred to as a maxterm expansion.

Standard Product Term: A product term that includes one instance of each independent variable; also known as a minterm.

Standard Sum of Products Form (Standard SOP Form): A description of a Boolean function that includes an explicit listing of the standard product terms that imply an active state (1's) on the function's output. Standard SOP is also referred to as a minterm expansion.

Standard Sum Term: A sum term that includes one instance of each independent variable; also known as a "maxterm"; (see "maxterm").

Standard: A set of rules or guidelines that everyone agrees to follow or be faced with the notion of choosing a slow death or becoming an academic administrator.

State Bubble: A visual representation of the values that can be stored by a sequential circuit. State bubbles can represent either the stored bits or some symbolic reference to the stored bits.

State Diagram Symbology: A term referring to the various standard set of symbols used to represent various aspects of state diagrams and the finite state machine (FSM) they represent. Representing state diagrams is not a science; it's more of an art form.

State Diagram: A visual representation of a PS/NS table used to describe the given values that a sequential circuit can store (or the "state") and the conditions required to for the circuit to transition from one state to another state.

State Registers: A sequential circuit used in the modeling and implementation of finite state machines (FSMs). The state registers are typically comprised of single-bit storage elements that are used to store the values associated with the "present state" of a given FSM. .

State Transition Inputs: A term that describes the inputs to the "synchronous process"; (see "synchronous process") that control the functioning of the state variables associated with a given FSM model. These inputs typically include parallel load, clears, and pre-sets.

State Transition: The characteristic associated with a sequential circuit where the values stored by that circuit change.

State Variable Transition Table: A set of information in tabular format that lists every state-to-state transition associated with a state diagram. For each transition, the conditions that govern that transition and the state changes for the associated state variables are also listed. This table is used in conjunction with the "new FSM techniques"; (see "new FSM techniques").

Statement Region (VHDL): The region of a VHDL architecture that support the various forms of VHDL statements including concurrent signal assignment statements and component instantiations.

Static logic hazards: A hazard that is present due to the changing of a single input variable for a given circuit where the given output is not expect to change (thus remain "static").

Status Signals: These are signals represented as outputs from a device being control and provide status information to a controller device. Finite state machines (FSMs) are typically used as controllers and contain both control outputs and status inputs.

Stimulus Driver: A term referring to one major portion of a VHDL testbench; the other portion of the testbench is the "device under test". The stimulus driver's main function is to provide inputs to the device under test. The stimulus driver can use the state of the DUT's output to generate conditional stimulus to the DUT. The stimulus driver can be modeled for either manual or automatic verification of the DUT.

Stimulus: A term referring to the application of test vectors to a device under test. The stimulus is generally in the form of exercising the digital inputs to the device under test.

Stone-Age Unary: A number system that uses one physical entity for each thing being counted.

Storage Element: A digital device that is capable of storing an arbitrary number of bits. Storage elements are typically associated with state variable representation in finite state machines (FSMs). Storage elements are often referred to as "memory elements".

Structural Style: A term referring to the use of structural modeling in VHDL.

Structured Digital Design: The notion that modern digital design is similar to typical computer program design. Specifically, any well-designed digital circuits can be decomposed into one of only a few standard and relatively simple digital circuits. This concept closely relates to object-level digital design.

Structured Programming: A term that refers to the notion that any properly written program can be decomposed into a set of four or five simple programming constructs. The notion here is that poorly written code cannot be composed into these constructs (aka spaghetti code).

Sub-Minterms: A subset of a standard minterm. Sub-minterms are generally used in the derivation and description of mapped entered variables (MEVs).

Subtractor: A device that subtracts one number from another number. In digital design, there are many forms of subtractors, each with their own particular set of characteristics.

Subtrahend: A number that is subtracted from another number.

Sum of Products (SOP) Form: A function form that is characterized by product terms that logically summed together. .

Sum Term: A set of Boolean variables that are ORed or logically summed together.

Sum Term: An expression in a Boolean equation that is characterized as a logical summation of variables.

SVTT: An abbreviation for "state variable transition table"; (see "state variable transition table").

Switching time: A term that is used to quantify the amount of time required for a signal to switch from high-to-low or low-to-high.

Symbology: A set of visual symbols used to describe the overall functioning of a device. Often times there is a specific set of "symbology" associated with a given classification of the thing being desribed; at other times, special symbols can be created by the user and described via a "legend" (see "legend") associated with the description.

Synchronous Circuit: A circuit that has some functionality that is synchronized to some event in the circuit, typically an active edge of a clock signal.

Synchronous Input: An input to a sequential circuit that only has an effect on the circuit based on an active edge of some other signal in the circuit.

Synchronous Process: One-half of a two-process approach to modeling finite state machines (FSMs) using VHDL; the other half of the FSM model is the "combinatorial process"; (see "combinatorial process"). The synchronous process is responsible for modeling the state registers and any logic that control the state registers such as parallel load, clears, and presents. The synchronous process implements the "state register" block associated with the standard FSM model.

Synthesize: A term typically used in digital design indicating the notion of using a model of something in one form and converting that model to another form. The two most common usages of this term on in hardware design languages where the act of synthesizing a VHDL model creates a new type of model that can eventually be converted into actual hardware. The other common usage of this term is to use some entity (such as a microcontroller for FSM) to recreate signals shown on a timing diagram.

System Clock: A clock signal for a given circuit that is typically used for all parts of the circuit. System clock signals are typically used to synchronize the various parts of a circuit by using a single signal in which all parts of the circuit can act upon.

-T-

T Flip-flop: A shorthand notation for a "toggle flip-flop"; (see "toggle flip-flop").

Tab Character: A type of white-space that includes any number of single spaces. Tab characters should never appear in the text of any type of code.

Tedious Grunt Work: A special form of "grunt work" that has a higher grunt factor than most of other "grunt work"; (see "grunt work").

Tedium: A frustrating state of affairs resulting from "doing" but not "learning".

Terms of Convenience: A phrase referring to an irreverent set of words that are typically not used together in the same context. This text has way too many "terms of convenience".

Test Vectors: A term referring to the set of data that is applied to a device under test. For a given VHDL testbench, test vectors can be stored in using one of three approaches: 1) "on the fly", 2) in hard-coded arrays, and/or 3) stored in external files.

Testbench: The term given to VHDL models whose primary purpose is to verify the correct operation of other VHDL models. The two main parts of a testbench are the "stimulus driver" and the "device under test' (DUT). Generally speaking, the stimulus driver provides input to the DUT.

Theorem: A proposition that can be proved true from a given set of axioms.

Throughput: A term that describes the amount of useful information that is processed by a circuit. Typical throughput metics include intructions per second (IPS), floating point operations per second (FPS), etc.

Tied High: A term used to indicate an input to a gate is connected to a logical '1'. In a real circuit, this term generally refers to connecting an input to the high voltage used to power your digital circuit.

Tied Low: A term used to indicate an input to a gate is connected to a logical '0'. In a real circuit, this term generally refers to connecting an input to the low or ground voltage used to power your digital circuit.

Tied-To: A commonly used, but slang notation indicating an electrical connection for a given device. Two of the more common uses of this term include "tied to ground" (a signal is connected to ground, or '0') and "tied to power" (a signal connected to power, or '1').

Time Slots: A term that refers to finite periods of time. Time slots are often used to describe the amount of time associated with a given state in a finite state machine (FSM).

Timelessness: The feeling you get when you read this text. No matter how hard you try, you simply can't make that feeling go away.

Timing Analysis: The act of analyzing a given timing diagram in order to do fun things like gather information of verify whether the circuit is actually operating correctly.

Timing diagram annotation: A special notation used to indicate or highlight certain properties or conditions in a given timing diagram. The underlying purpose of timing diagram notation is to convey certain information to the reader; the quality of the timing diagram notation is judged by how efficiently that information can be conveyed.

Timing Diagrams: A graphical representation of the operational characteristics of a circuit based on the notion of observing circuit operation over a given span of time. The horizontal axis is typically used to represent time in timing diagrams while the vertical access is used to list signals and show the state of those signals. Timing diagrams have two primary uses: they serve as design aids and they serve to verify the proper operational of circuits.

Tiny Electronic Things: A term referring to entities that enhance the "conspicuous consumption" tendencies of normally intelligent people by increasing a person's personal need to "keep up with the Jones's".

Toggle Flip-flop: A flip-flop that changes the output state when the "toggle" input to the flip-flop is asserted and an active edge occurs on the clocking input the circuit. The "next state" of a T flip-flop is a function of both the T input and the present state of the T flip-flop.

Toggle: A term that refers to changing the value of a bit; the act of toggling a bit changes the bit value from either '1' to '0' or '0' to '1' depending on the initial value of the bit.

Top-Down Design: A hierarchical design approach that starts at the highest level of abstraction and works downwards. In this approach, the designer fills in the lower levels of abstraction as the design progresses.

Truncation: A term used to describe the removal of one or more digits from a value. The digits removed are contiguous and are generally either the most significant or least significant digits in the given number.

Truth Table: A matrix that shows all possible input combinations and the associated output values.

Two's Compliment: As a noun this term refers to an alternate and more popular method of describing radix compliment (RC) form; (see "*radix compliment*"). As a verb, this term refers to the notion of changing the sign of a signed binary number in RC form.

Two-Valued Algebra: An algebra based on only two variables. This term commonly refers to Boolean algebra.

-U-

UDC: An acronym used for unit distance code; (see "unit distance code").

Unasserted: A term used to indicate that the current voltage level of a signal is not associated with the active state of that signal.

Unconditional transition: A term that refers to a state-to-state transition in a finite state machine (FSM) that occurs independently of any conditions in a given circuit. These transitions are often referred to as "don't care transitions".

Un-Dead: A term used to describe a circuit element that is enabled (or not disabled). Similarly, a dead circuit has an output that is pre-determined and does not change so long as the circuit remains dead.

Underflow: A condition that indicates the result of a mathematical operation has exceeded the bottom end of the rang of numbers associated with the bit-width of the operands. Underflow is often characterized as a special case of overflow; (see "*overflow*").

Unit Distance Code: A binary code where the differences between to binary numbers in the sequence differ by a unit distance (a distance of one).

Universal Shift Register: A shift register that can perform more operations than simple shifting. These other operations can include rotation, barrel shifting, parallel loading, resetting, etc.

Unsigned Binary Number: a set of bits (1's and 0's) that are used to represent a numbers greater or equal to zero. Unsigned binary numbers can be used to represent zero and positive numbers.

Unused State: A condition generally associated with finite state machine (FSM) design. This condition is present because of the binary relationship associated with some methods used to encode state variables which leave some combinations of the associated storage elements intentionally unused. The FSM could thus unintentionally find itself in these unused states and potentially cause undesired operation of the FSM. .

Up Counter: A counter that counts only in the "up" direction (count value becomes greater).

Up/Down Counter: A counter that can counter either up (count value increases) or down (count value decreases) according to a selection input on the device.

User-Level: A term used to describe the number of bits in the operands and/or result of a circuit that performs addition.

USR: An acronym representing "universal shift register"; (see "universal shift register").

-V-

Variable Assignment Operator: The VHDL operator used to assigned values to variables: ":=".

Variable: A VHDL type used to store intermediate results. Variables can only be declared in the declarative regions of process and are only visible in those processes in which they are declared. The results of variable assignments are ready for immediate use in the process and are not "scheduled" for assignment once the process completes as is the case the with signals. .

Vcc: A term referring to the power connection in electronics. In digital eletronics, this signal is generally considered a logical '1'. Sometimes the term "Vdd" is used in place of "Vcc", but not often.

Vdd: A term referring to the power connection in electronics. In digital eletronics, this signal is generally considered a logical '1'. Usually the term "Vcc" is used in place of "Vdd".

Vector: A term used to signify that a given item that can be decomposed into two or more sub-items.

Verilog: A modern hardware description language (HDL) that is used quite widely in North America but less so in other areas of the world. Verilog syntax has a strong resemblance to C programming syntax.

Very Large Scale Integration: A type of integrated circuit that contains a buttload of transistors (certainly more transistors then large scale integration (LSI) ICs). This term often described with the acronym "VLSI".

VLSI: An acronym for "very large scale integration"; (see "very large scale integration").

-W-

Wait Statement: A "wait" statement is a VHDL sequential statement that is used to suspend execution of process statements. Only process statements that do not include process sensitivity lists can use wait statements. There are four forms of wait statements in VHDL; most of these forms are particularly useful in modeling VHDL testbenches.

Wanker: Any person who pretends to be something they're not; this includes talking big while knowing small. All academic personnel seem to have a hopeless case of wankerism as well as a healthy case of apathy towards their condition.

Wankerism: A term describing the collective mindset of wankers. Academic administrators always strive to take wankeristic tendencies to new heights.

Waveform: A term referring to a visual representation of a signal over a given amount of time.

Weights: This refers to the values assigned to various digit locations when juxtapositional notation is used. The weights are typically powers of the radix for a given number system but can be just about anything as weight assignments are arbitrary.

White Space: A term describing the areas of text that have no printed characters in them; white space generally includes space characters, tab characters, and blank lines.

Width: A term that describes the number of signal in a bundle or the number of bits associated with digital devices that operate in parallel such as "comparators" and "ripple carry adders". f

Wire-Wrap: A method used for prototyping electronic circuits that entail stripping the plastic coating off of a wire and wrapping it around a metal post that was electrically connected to an electronic device.

Wrapper: A term used to describe an addition to an item that abstracts, simplifies, and/or extends the usage of that item. Wrappers in VHDL generally includes an interface that is used to customize the usage of an established model.

-X-

XNOR Gate: A shorthand name for an exclusive NOR gate, one of the standard logic gates; an XNOR gate performs an XOR function with a complimented output. XNOR gates can also be considered to perform an XOR function with an active low output. XNOR gates are also known as "equivalence gates" as the gate output indicates when the gate's two inputs are equivalent. XNOR gates by definition always have two inputs.

XOR Gate: A shorthand name for an exclusive OR gate, one of the standard logic gates; an XOR gate performs an XOR function which is typically defined using a truth table or a Boolean equation. XOR gates indicate when the gate's two inputs are not equivalent. XOR gates by definition always have two inputs.

-Y-

Y: The letter often used as a label in finite state machine lingo to refer to external inputs.

-Z-

Z: The letter often used as a label in finite state machine lingo to refer to external outputs.

µ: An abbreviation used for the metric prefix "micro"; this prefix is used in engineering notation.

Index of Stuff

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X