## FreeRange

# Digital Design Foundation Modeling

James Mealy © 2018

v5.00

# Table of Contents

# Pretentions

Legal Stuff

FreeRange Digital Design Foundation Modeling

Release: xxx

Date: xxx

You can download a free electronic version of this book from one of the following sites:

freerangefactory.org

http://www.ee.calpoly.edu/faculty/bmealy/

The author has taken great care in the preparation of this book, but makes no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or models contained in this book.

We are more than happy to consider your contribution in improving, extending or correcting any part of this book. For any communication or feedback that you might have regarding the content of this book, feel free to contact the author at the following address:

bmealy@calpoly.edu

# Acknowledgements

Someday, I'll write something here.

Hey Dickson… Someday we'll work together on all the things we've not yet completed. I look forward to that day.

# One Person’s Viewpoint

**Rambling Commentary**

My inspiration for this project came from two primary sources. First, I feel that publishing companies, bookstores, book authors, and academic administrators should not hold knowledge ransom. Students seeking knowledge are sitting ducks in structured learning situations such as colleges and universities. Being that students are the lowest hanging fruit, they always are the first to have their wallets lightened by various well-connected entities. Second, every digital design textbook I’ve ever examined were filled with low-level details and techniques that are forgotten by students about five minutes after the final exam (which implies too much memorization and not enough understanding). The approach taken in this text is a giant step in the right direction (more details later). In the end, I hope this book serves as an alternative to shelling out money for overpriced textbooks full of knowledge that serves little purpose.

This book will have errors. Please accept my sincerest apologies for the errors. I did my best to remove errors, but writing and proofreading is timing consuming and painfully boring. Unlike several of my colleagues, I don’t bribe students into proofreading my writing. I do happily accept suggestions and corrections from students, but I do not hand out rewards.

I generated every digital design problem in this book. Once again, unlike many authors, I did not “assign” students to generate problems as assignments, and then use those problems in my text. I believe instructors who force students to create problems as graded assignments are unethical and are taking advantage of their positions as instructors.

I could spend the remainder of my life tweaking this text, but I need to move onto other things. Feel free to contact me with corrections and comments. Please feel free to write at this address: bmealy@calpoly.edu

There were two primary negative comments I received when I mentioned I was writing a textbook and was planning to give the book 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.

“You need 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.

Finally, this text is what it is. The quality and coverage is the best I can do given the various constraints I work under. I made the decision to embark on this project knowing it would likely be a career killer in the context of Cal Poly San Luis Obispo. Well, no need to wonder anymore; it’s definitely a career killer. I opted to directly support all students; if I had the chance to make the decision again, I would do no different. My previous books have found their way all over the world, seemingly helping students learn digital design and associated topics without lightening their wallets. What more could one ask for?

James Mealy

# Topic Coverage & Previous Books

I previously wrote another digital design textbook: Digital McLogic Design. That book was a multi-year project that quickly outgrew itself and lost its way. While using that book in my courses, I always felt there was a better approach to teaching digital design, but I could not quite nail it down. This book represents what I feel is a significantly better approach to learning digital design. Here is my reasoning:

- This text removes many low-level details associated with digital design in the standard approach to teaching digital design. I always found these details were the first things students forgot after leaving the final exam. This book instead concentrates on higher-level design principles. Not including the low-level details frees up more time to delve into design-oriented problems rather than learning how to represent a function in a bajillion different ways.
- I ejected 98% of the concept of "function reduction". This book proudly contains no Karnaugh Maps, or what I can "high-tech" tic-tac-toe. Karnaugh maps have severe limitations, and, no one will ever pay you do something by hand that a computer can do a bajillion times better and faster.
- About 80-90% of the material found in this book is new. I did reuse some of the images, but I also "cleaned up" all of the text. My previous approach was to be purposely verbose; I try to be direct and terse in this book.
- I suspect people will argue that the example problems in this text are sort of stupid. I can't say I disagree. My thoughts are that people wanting to learn digital design must learn about the operation of basic devices; they will learn about those devices by using them in the designs found in this text. When and if they are someday faced with the task of creating a digital circuit to solve a problem, they will be able to do it because they understand how to use various digital modules to create circuits that work. Additionally, when the class is over, they hopefully will recall the basic operations of digital circuits long after they forget how to implement a Boolean function using a MUX. Lots of fun stuff in digital design, but much it lacks a point.
- I removed HDL from the text. The previous text integrated VHDL and digital design, but no longer seems like the best solution. In the end, decoupling HDL from digital design allows students to learn either VHDL or Verilog. I've completed a first-pass version of a Verilog tutorial that I am using in my current digital design course offering. Additionally, there is the ***FreeRange VHDL Tutorial*** available from the freerangefactory.org site. If you want a hardcopy, Fabrizio (the co-author) printed a batch and has made them available on Amazon.com.

And really finally, a story… I got a co-op job at National Semiconductor in 1988. Part of my job was to create a digital circuit that tested various digital modules using a new fabrication process. My group tasked me to create a simple circuit at a meeting I attended early in the co-op. The truth was that I had no clue what to do or even where to start. I had already taken two "digital design" courses in college, but I could not for the life of me design a digital circuit that solved an actual problem. I suspect my boss recognized my dismay, and quickly jotted down a digital circuit for me (a bunch of modules that talked to each other). He made it look as easy as it should be. More than anything, I hope people reading this text can walk away knowing how to create digital circuits to solve problems. Good luck.

# Overview of Chapter Overviews

This textbook presents introductory digital design topics with an emphasis on actual design issues. This textbook initially provides a basis of digital design, followed by a novel approach to modular digital design, which we refer to as Digital Design Foundation Modeling. This text exclude many topics typically found in digital design textbooks in order to focus more on the important aspects of modern digital design.

**Chapter 1:** This chapter provides an outline of chapter structure in this text as well as an overview of the basic approach this text takes to teach digital design. This chapter also describes the basic tenets regarding the underlying theme of this text with the new digital design paradigm of Digital Design Foundation Modeling (DDFM). DDFM brings simplicity and structured-type design to the field of digital design. The descriptions this chapter uses to describe DDFM become clearer as the reader progresses the following chapters in this text.

**Chapter 2:** This chapter provides a description of the "digital things" by example. The examples include comparisons and descriptions of basic everyday items to give readers an intuitive feel for the descriptions.

**Chapter 3:** This chapter introduces the basic aspects of modeling, including definitions and examples. This supports the notion that all the work we do with digital design requires the use of many types of models. A main emphasis in this chapter is an introduction to hierarchical modeling.

**Chapter 4:** This chapter provides a basic overview of number systems with an emphasis on their importance in digital design. The topics in this chapter include the basic vernacular associated with number systems, an introduction to binary and hexadecimal number systems and their properties important to digital design, and closes with an overview of engineering notation.

**Chapter 5:** This chapter provides an overview of number conversions between radii typically associated with digital design (decimal, binary, and hexadecimal). This chapter also discusses other useful codes including BCD, one-hot, and unit distance codes.

**Chapter 6:** This chapter introduces Brute Force Design, this text's first true notion of digital design. This chapter presents digital design in the context of an example problem that we use to introduce the important aspects of modeling digital circuits, Boolean algebra, and basic logic gates.

**Chapter 7:** This chapter introduces basic aspects of timing diagrams including proper annotation and bundle notation.

**Chapter 8:** This chapter introduces the first foundation module: the ripple carry adder. This chapter starts by introducing basic digital modules including half and full adders.

**Chapter 9:** This chapter introduces the basic aspects of representing Boolean functions, with an emphasis on standard SOP and POS forms. This chapter also describes the more common ways to represent functions including compact minterm and maxterm forms.

**Chapter 10:** This chapter introduces the remainder of standard gates in digital design, which include NAND, NOR, XOR, & XNR gates. This chapter also shows how to configure basic logic gates to act as inverters, buffers, and switches.

**Chapter 11:** This chapter introduces the more useful circuit forms used in digital design. The eight forms covered in chapter can be generated from SOP & POS forms; this chapter places the most emphasis on AND/OR & NAND/NAND and OR/AND & NOR/NOR forms. This chapter also touches upon minimum cost concepts as they relate to the various circuit forms.

**Chapter 12:** This chapter introduces signed number using SM, DRC, and RC representations, with the most emphasis placed on RC representations. This chapter describes the number ranges associated with the various representations and describes the bit-extending both signed and unsigned number.

**Chapter 13:** This chapter also covers binary addition and subtraction using RC representations. This chapter emphasizes the importance of verifying the validity of mathematical operation results.

**Chapter 14:** This chapter introduces the concept of mixed logic and uses those concepts in design and analysis examples. This chapter uses the concepts of equivalent gates and equivalent signals in the design and analysis process. This chapter mentions the PLC approach but primarily focuses on DPI.

**Chapter 15:** This chapter presents a formal description of Modular Design (MD), which is the focus of design techniques in the remaining chapters. There are a few examples in this chapter, but the text adds more examples in later chapters after the presenting of various foundation modules.

**Chapter 16:** This chapter introduces the concept of decoders; there are two types of these foundation modules: generic decoders and standard decoders. Generic decoders are any device we can describe using a tabular format while standard decoders are special cases of decoders that function as device enablers. .

**Chapter 17:** This chapter introduces multiplexors (MUXes), which is another foundation module. MUXes act as "selector circuits" in digital design.

**Chapter 18:** This chapter introduces the comparator, which is another foundation module. This chapter derives simple comparators at a low level, and then abstracts more complete comparators at the block level.

**Chapter 19:** This chapter introduces the concept of parity, which leads to two foundation modules: the parity generator and parity checker. This chapter derives a simple parity generator at a low level, and then abstracts the design to the block level.

**Chapter 20:** This chapter introduces sequential circuits. This chapter covers the low-level details regarding basic NOR and NAND latches, and then abstracts these devices to modules. This chapter is the first chapter that uses PS/NS tables and state diagrams to describe sequential circuits.

**Chapter 21:** This chapter introduces the D flip-flop, the only flip-flop this text discusses. This chapter also describes synchronous and asynchronous control inputs to flip-flops. This chapter also provides an introduction description of state diagrams.

**Chapter 22:** This chapter introduces registers, which is essentially an extension of the previous chapter. The register is a digital design foundation module.

**Chapter 23:** This chapter introduces finite state machines (FSMs) by introducing the various submodules of FSMs in terms of circuits we previously presented. This chapter presents the concepts of FSM in the context of low-level counter designs. The chapter also includes description of the standard symbology associated with state diagrams.

**Chapter 24:** This chapter introduces some aspects of clocking basic sequential circuits. This chapter includes an overview of the vernacular associated with periodic clock signals. This chapter introduces setup and hold time issues in the context of maximum clocking frequencies of sequential circuits.

**Chapter 25:** This chapter introduces the use of FSM as controller circuits. This chapter uses FSMs to control simple blinking LEDs and then moves onto using FSMs as sequence detectors.

**Chapter 26:** This chapter introduces various types of counters, which is a digital design foundation module. A previous chapter introduced simple low-level counters; this chapter presents counters at a high-level by describing their basic attributes.

**Chapter 27:** This chapter introduces the shift registers, which is a digital design foundation module. Shift registers are essentially simple registers with special features. This chapter also introduces other related shifting-type operation including barrel shifts, arithmetic shifts, and rotate operations.

**Chapter 28:** This chapter presents the basic concepts of relatively large memory devices such as RAMs and ROMs. We include the RAM as a digital design foundation module. This chapter also covers basic structured memory performance, memory capacity parameters, and memory vernacular.

**Appendix:** This provides an overview of Digital Design Foundation Modeling Concepts.

**Digital Design Dictionary:** This is a glossary of popular words and expressions, (and other tidbits) having to do with digital design and computer design. This glossary also covers aspects of computer design.

**Index:** This is an index for the important words and phrases found throughout the text.

# 1 FreeRange Digital Design Foundation Modeling Overview

## 1.1 Introduction

This text divides topics into small subject modules, which we creatively refer to as chapters. The intention is to keep the subject matter as short as possible and bundled into relatively small readable portions. No one wants to read long pages of technical drivel, but people are more likely to read short pages of technical drivel.

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

- **Introduction**: Quick motivating prose overview of the main chapter topics
- **Chapter Acquired Skills**: The skills reader should have after working through the chapter
- **The Body of the Chapter**: In case you want the whole story (with example problems)
- **Chapter Summary**: The quick overview of chapter's main points
- **Chapter Exercises:** Drill-type problems that support the chapter material
- **Design Problems:** Problems that involve digital circuit design

**Main Chapter Topics**

**OVERVIEW OF TEACHING MODERN DIGITAL DESIGN**: Digital design evolved faster than digital design courses could keep up with; a quick overview of the issues is helpful.

**OVERVIEW OF DIGITAL DESIGN FOUNDATION MODELING**: This chapter briefly describes this text's unique approach to digital design.

**Chapter Acquired Skills**

- Be able to provide historical context to digital design
- Be able to describe the basic approach of *Digital Design Foundation Modeling*.

## 1.2 Digital Design Overview

Even though we're only a few pages into the introductory verbage[1] of digital design, we're ready to grasp the main ideas behind modern digital design and relate them to this text's approach. If you had to describe digital design in one short sentence, it would be something such as:

**digital design**: the act of creating 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.

[1] Definition of verbage: part verbose, part garbage; pronounced *ver-baj*.

**Solving a Problem**: “Solving a problem” could mean many things; this text solves problems using digital circuits. Figure 1.1 shows a general diagram of a digital circuit that we use as a starting point for this text. We won’t initially know from the problem description what goes inside the box in the diagram, but the problem description generally tells us the “inputs” and “outputs” of the circuit as well as how the circuit should behave.

**Creating Digital Circuits**: The many approaches to digital design all involve creating a digital circuit and placing it (figuratively speaking) in the box of Figure 1.1. If your digital circuit manipulates the inputs in such a way as to always provide the requested functionality on the outputs, then your digital design works. The digital circuit you design establishes a structured relationship between the circuit’s inputs and outputs in such a way as to solve the given problem. In a nutshell, digital design is a matter of “creating” the interior of the Digital Circuit box in Figure 1.1.

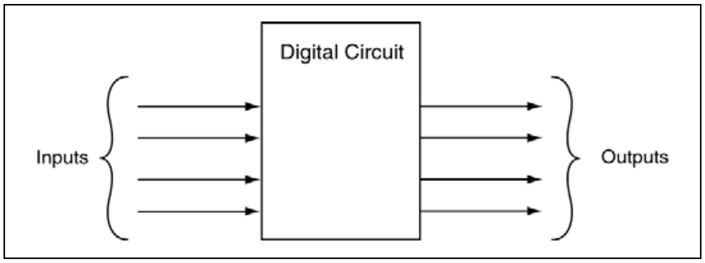


**Figure 1.1: “Digital Design” in a nutshell: a general model of a digital circuit.**

## 1.3 Historical Overview of Digital Design Courses

It was a different world when I first worked with digital logic (sometime in the mid-1800s); my digital design world revolved around the knowledge and topics presented in the course text. There was no laboratory associated with the digital design course. Because of this lack of hands-on experience, combined with the fact that the test/development equipment too costly for students, I relied on the course text to gather my digital knowledge. Computers were expensive and not practically available to students. There was no internet and software for digital designers either did not exist or was too expensive to be practical. Worst of all, all “designs” that were actually done were “paper designs[2]”.

## 1.4 The Approach We’ll Be Taking

Despite advances in digital technology, digital design textbooks remain mired in the dark ages of both engineering and educational technology. Despite these drawbacks, we see a steady increase in the price of digital design textbooks accompanied by a decline in their quality. As digital technology progressed, more resources became available to both digital design instructors and students, which obsolesced the standard approach to teaching digital design.

Although the goal of transferring knowledge from the text to the reader remains the same, it’s not universally accepted what topics should appear in a text. Typical digital design texts contain excessive amounts of low-level detail that you’ll quickly forget. Authors write digital design texts from a standpoint of presenting digital concepts in a manner that supports the easy generation of exam questions, which makes the text attractive to lazy instructors. Actual design problems are hard to generate and harder to grade[3], and thus rarely find their way onto exams or into textbooks.

The underlying theme of this textbook is to eject subject matter that does not support the development of viable digital designers; doing so allows us to spend more time with actual digital design. We acknowledge that

[2] A paper design was something you tried hard to convince someone else that it would work if you actually implemented it. The person you were trying to convince was often your instructor.

[3] The issue here is that digital design problems always have multiple solutions, which requires a higher level of expertise and effort to both genereate and grade properly. In this context, “expertise” requires more time.

advanced digital designers or instructors who read this text may feel that this text omits some important topics. However, with the knowledge we present in this text, anyone can easily pick up a standard digital design textbook and gather in the full details. In addition, because publishers actively generate new versions of textbooks to prevent instructors from using old textbooks, there are many excellent and low-priced textbooks available from used book websites[4].

## 1.5 The New Digital Paradigm: Digital Design Foundation Modeling

After many years of teaching digital design using a traditional approach, we formulated a new paradigm for presenting digital design. We refer to our new approach as ***Digital Design Foundation Modeling***, or ***DDFM***. This approach builds upon both *modular design* and *hierarchical design*, which are the main tenets of modern digital design. DDFM focuses on presenting digital design topics in the context of actual digital designs, while removing many of the antiquated topics associated with old-style digital design. The underlying goals of DDFM are to simplify the presentation of introductory digital design, and to provide a simple circuit model that describes all levels of digital design.

### 1.5.1 DDFM Overview

We provide the high-level details about DDFM in this section, but if you're new to digital design, you probably won't be able to grasp the big picture at this time. The focus of DDFM is to present digital design in a simple and organized manner, which facilitates and expedites learning the subject matter. These are the main tenets of DDFM:

- The main purpose of digital design is to solve problems using digital circuits
- We can best describe digital circuits in a modular and hierarchical manner
- Digital circuits are a set of digital modules that exchange information under the control of some entity
- We perform digital circuit design in a ***structured***[5] manner, meaning that we can model ***any*** digital circuit using a relatively small subset of digital modules, which we refer to as the ***digital design foundation modules***. Each foundation module performs a relatively small set of simple operations.
- We present the digital design foundation modules at a high-level by modeling the modules in terms of their data, control, and status signals, which allows us to use the modules in designs, while not requiring us to initially understand underlying implementation details.
- We classify the digital design foundation modules as either "controlled" or "controller" circuits
- We consider there to be four approaches to controlling a digital circuit:
    1) **NO CONTROL** (no flexibility in circuit behavior)
    2) **INTERNAL CONTROL** (controlling circuits using internal signals)
    3) **EXTERNAL CONTROL** (controlling circuits with devices such as buttons, switches, etc.)
    4) **CIRCUIT CONTROL** (controlling circuits using FSM or computer).
- We categorize digital design approaches into three categories:
    1) **BRUTE FORCE DESIGN (BFD)**
    2) **ITERATIVE MODULAR DESIGN (IMD)**
    3) **MODULAR DESIGN (MD)**

[4] Check out your local library, www.ebay.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.

[5] This is an analogy to structured computer program design

Figure 1.2 shows a digital circuit containing various modules. We define a digital circuit as a controlled interaction between a set of sequential and combinatorial circuits (the two types of digital circuits). Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it solves the given problem. Figure 1.2 also shows the modularity (the various modules) and the hierarchical (modules within modules, or boxes within boxes) characteristics of digital circuits.

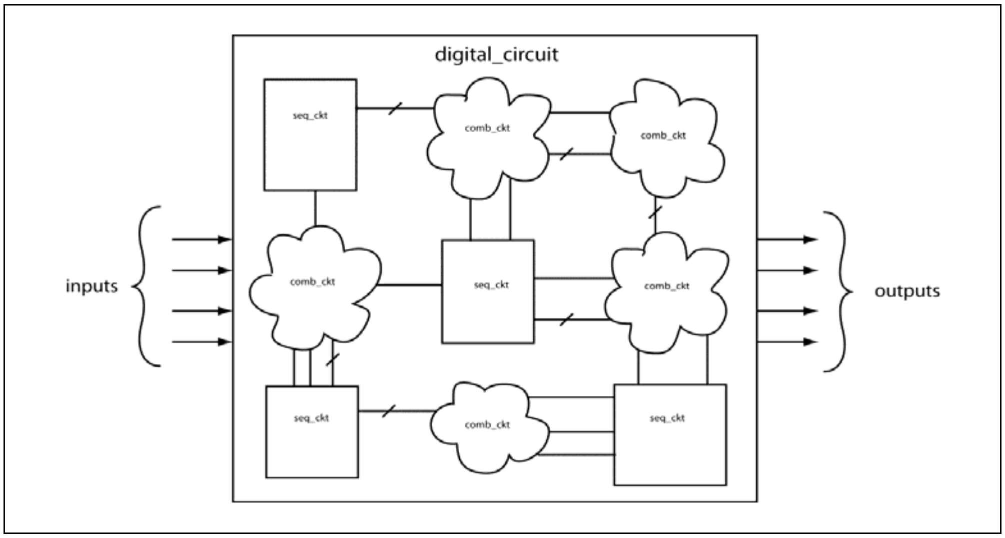


**Figure 1.2: A generic digital circuit containing a set of digital modules.**

Figure 1.3(a) shows the standard approach to modeling digital circuits, where we classify all digital circuit signals as either inputs or outputs. Figure 1.3(b) and Figure 1.3(c) shows how DDFM further classifies inputs and outputs by first separating digital modules into “controlled circuits” and “controller circuits”. Figure 1.3(b) shows that we further classify the inputs to controlled circuits as either “data” or “control” and classify the outputs of controlled circuits as either “data” or “status”. This means the various circuit elements in Figure 1.3(b) are able to 1) pass data from their data inputs to their data outputs under the direction of the “control” inputs, and, 2) describe characteristics of the data transfers using the status outputs. Similarly, the status outputs of the controlled circuit form the status inputs of the controller circuit. The controller circuit of Figure 1.3(c) inputs the status signals of controlled circuits and manages the controlled circuits by outputting the appropriate control signals to control the controlled circuits.

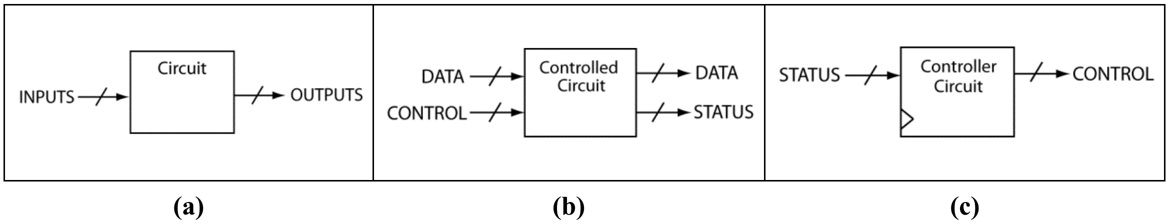


**Figure 1.3: Old digital circuit model (a); models for controlled (b) and controller circuits (c).**

The DDFM paradigm allows us to model all digital circuits as a controller that controls a set of modules. We then consider the solution to any digital design problem as a matter of using a controller to properly control the dataflow through a set of controllable modules. Figure 1.4 shows an example of many circuit modules controlled by a controller circuit; the controller circuit is either a finite state machine (FSM) or some type of computer control, such as a microcontroller. Figure 1.4 includes three different module shapes showing that controllable modules can either be combinatorial or sequential circuits, as well as off-the-shelf computer peripherals.

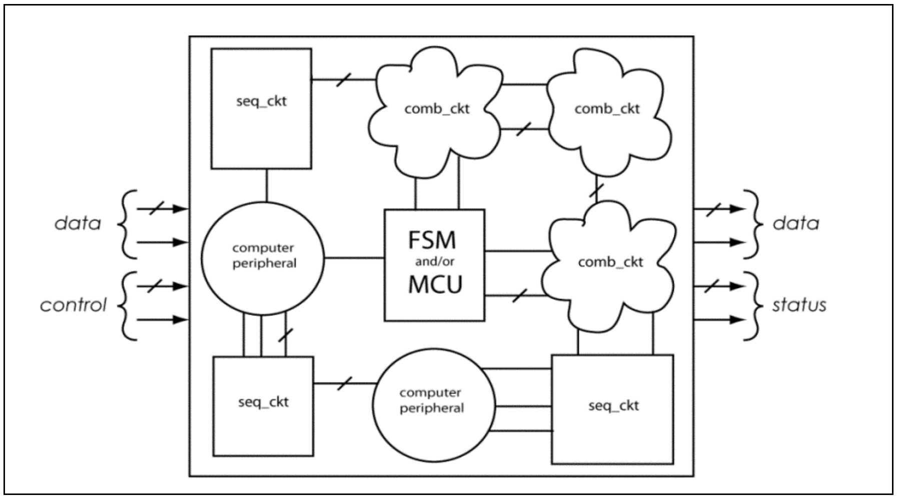


**Figure 1.4: Our unifying digital circuit model.**

### 1.5.2 The Three Approaches to Digital Design

Part of DDFM includes categorizing digital design into three different approaches, which we discuss in more detail later in the text. With some combination of these three approaches, you can create any digital circuit.

**Brute Force Design (BFD):** Our first approach to digital design. Although simple, its simplicity limits its practicality in non-trivial designs.

**Iterative Modular Design (IMD):** Our second approach to digital design. Although IMD removes some of the limitations of BFD, it is only applicable to a few of circuits.

**Modular Design (MD):** Our final and most powerful approach to digital design, and is thus where this text expends most of its efforts.

## 1.6 Chapter Summary

- There are two basic types of digital logic circuits combinatorial, which are circuits where outputs are a function of the circuit's inputs (these circuits can't store information). Sequential circuits outputs are a function of the sequence of the circuit's inputs (these circuits can store information).
- The two basic tenets of digital logic are 1) **Digital logic circuits are inherently hierarchical.** We generally describe digital circuits at a level, which allows us to transfer as information as quickly as possible. Abstracting digital designs to higher levels aids in understanding and designing circuits. 2) **Digital logic circuits are modular in that they are decomposable into a set of standard digital modules, which we refer to as digital design foundation modules.** We make circuit descriptions an aggregate compilation of foundation modules to help us understand the circuits.
- Digital Design Foundation Modules is based on the following attributes:
  - The main purpose of digital design is to solve problem using digital circuits.
  - Digital circuits are a set of digital modules that exchange information under the control of some entity.
  - We can complete any digital circuit design by using a relatively small subset of digital modules we refer to as the digital design foundation modules.
  - We can present the digital design foundation modules at a high-level by primarily describing the functionality of the circuit in terms of its associated data, control, and status signals.
  - We classify the digital design foundation modules as either "controlled" or "controller" circuits.
  - There are four approaches to controlling a digital circuit:
    1) **NO CONTROL** (no flexibility in circuit behavior)
    2) **INTERNAL CONTROL** (using internal signals)
    3) **EXTERNAL CONTROL** (using buttons, switches, etc.)
    4) **CIRCUIT CONTROL** (using FSM or computer).
  - There are three approaches to designing a digital circuit:
    1) **BRUTE FORCE DESIGN**
    2) **ITERATIVE MODULAR DESIGN**
    3) **MODULAR DESIGN**

## 1.7 Chapter Exercises

**1)** List and briefly describe the basic definition of digital design.

**2)** Briefly explain why there is no good off-the-shelf textbook for digital design courses.

**3)** List a few websites where you can purchase inexpensive digital design texts.

**4)** Briefly describe the main goals of Digital Design Foundation Modeling.

**5)** Briefly describe the three main types of design.

**6)** Briefly describe the four ways you can control a digital circuit.

# 2 The Battle of Analog and Digital

## 2.1 Introduction

The first step in learning anything is to become familiar with the terminology associated with the subject matter. This chapter starts by defining the notions of “digital” and “design”. We won’t be doing any digital design in this chapter, but we gain a common foundation for introducing later subject matter.

### Main Chapter Topics

**ANALOG AND DIGITAL:** This chapter provides a description of the inherent differences between things that are “analog” and “digital”.

### Chapter Acquired Skills

- Be able to describe things that are digital or analog in nature.

## 2.2 Analog Things and Digital Things

Since the term “digital” is quite important in this text, we need to give it solid definition. You can best understand the concept of “digital” when you see it alongside the definition of the relative opposite of digital, or “analog”. We can best describe these terms with examples.

**Example 1**: In doing the sustainability thing, I installed compact fluorescent (CF) lights in place of my incandescent lights as well as dimmers on incandescent light I did not replace. While the CF lights use less power, the intensity of their light is not adjustable: the CF light is all the way on or all the way off. While incandescent lights use more energy, I can save energy by using a “dimmer” to adjust the light’s output intensity. I am thus hypothetically able to adjust the dimmer to provide an infinite number of light intensity levels (but only one 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 have both wheelchair ramps and stairs leading to their entrances. The wheelchair ramp represents 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 you can stop at; the individual stairs represent these levels. The difference here is discrete levels (for the stairs) vs. continuous levels (for the ramp). This example differs from the previous example in that 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 such as the steps is the hallmark of digital while the continuousness of things such as the ramp is the hallmark of analog. What’s bugging me, though, is how to characterize an escalator…

**Example 3**: Stringed instruments create sound with a vibrating string connected 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, which effectively changes the length of the vibrating string. The difference between these instruments is that guitars have frets on the fingerboard 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)[1]. The violin, on the other hand, has no frets, so you can effectively generate an infinite number of pitches from a given string dependent upon where you place your finger. In other words, the guitar generates a discrete number of pitches while the violin provides a continuous number of pitches.

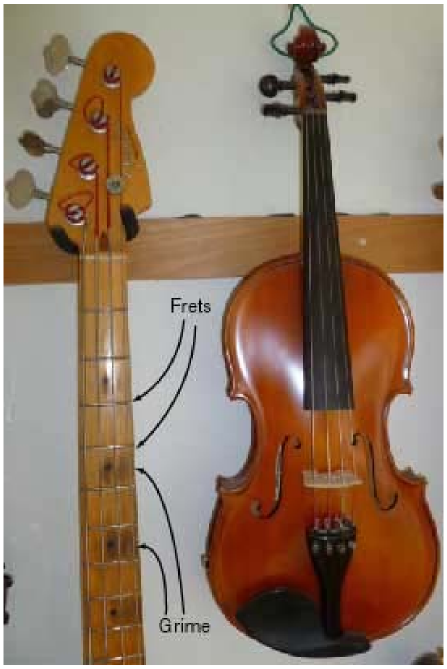


**Figure 2.1: "Frets" on a bass guitar fingerboard and the “fretless” violin fingerboard on the right.**

The basis of all digital logic is the use of circuit elements whose inputs and outputs can only be one of two values. We typically describe these two values as ON-OFF, TRUE-FALSE, HIGH-LOW, GOOD-BAD, BLACK-WHITE, TEACHER-ADMINISTATOR, etc. Either a “high voltage” or “low voltage” drives the actual circuit, but we generally describe circuits using more general terms. We typically represent the inputs and outputs of digital circuits using 1’s and 0’s, which are placeholders for the high and low voltage values[2].

So why do we use digital circuits to solve my problems? We’re still all living in an analog world, but computers are only capable of operating in the digital realm[3]. Since a computer is generally a giant digital circuit, understanding digital design is the unstated first step in successfully designing and/or programming computers. The starting point for mastering anything inherently digital is learning digital design[4].

[1] We’re not considering using your fingers to stretch the string (which changes the frequency of the note).
[2] The notion of "voltage" may scare off some budding digital designers 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.
[3] To be clear, computers are made with transitor, which work because of the voltages attached to them. Transitors in digital circuits are either “all the way on” or “all the way off”, which provides them with their discretness.
[4] And protecting yourself from robots.

**Digital**: A description of a something (such as a signal or data) 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) 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.

## 2.3 Chapter Summary

- We divide the world into two 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. 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 (thus continuous).
- The notion of digital things in the context of "digital design" generally only requires on two discrete values. These values are most often associated with ON/OFF, HIGH/LOW, or TRUE/FALSE. Most often in digital design, we describe these discrete values with '1' and '0'.
- The notion of digital in "digital design" 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 results from high and low voltages associated with making the transistor operate. Since the voltage levels determine the physical characteristics of the devices, different digital devices use different voltage levels. Because of all these different voltage levels, we represent the discrete values of transistors in digital circuits as 1's or 0's.

## 2.4 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 his or her 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)** 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.

**3)** In reference to analog and digital cameras, describe the difference between analog zoom and digital zoom.

**4)** There are analog computers out there. Briefly describe what an analog computer entails. Feel free to look this up online.

# 3 The Wonderful World of Modeling

## 3.1 Introduction

The ability to “model” something corresponds to both the ability to understand and implement that thing. Because of this, modeling is at the heart of all engineering fields. The approach we take to designing anything is to first find an appropriate model for that thing.

Digital design uses many different types of models to help us understand the characteristics of digital design. There are many different paths to solving a problem using digital design, but all of these paths follow the same path: model your solution, and then use that model to help you create a digital circuit that solves your problem. This chapter introduces the basic aspects of modeling in the context of digital design.

**Main Chapter Topics**

**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.

**Chapter Acquired Skills**

- Be able to describe the basic purpose of models
- Be able to use black box diagrams to create models of anything
- Be able to describe how model relates to modern digital design

## 3.2 The “Modeling” Approach to Anything

Until now, we’ve been careful not to limit our use the word “model” or “modeling”. The truth is that everything we do in digital design is a matter of generating the correct model. Below are two good definitions of the word model. We could not clearly define “model” with one definition, so we use two different but similar definitions. Note that the two definitions contain a different amount of detail, which is an important characteristic in modeling.

**model** (def. 1): a description of something.

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

We use models to represent or describe things. The above definitions do not state how we use models to describe something, which implies that there is no one absolutely correct model of something. Anything that presents information by describing something is by definition a model; some models are more useful than other models based on the amount of information they provide. Because there is no one “correct” model for anything, there can be different “valid” models of the same thing where, the different models provide varying amounts of information. The best model is the one that provides the reader with the most appropriate amount of information in the clearest manner for the problem at hand, which means the efficacy of models is inherently contextual. For example, a model providing significant amounts of information is not overly useful to someone expecting a simple model.

Models are important in digital design for one basic reason: models transfer information to the entity reading the model. The entity reading the model could be software, your lab partner, your teacher, or your pet cockroach. If you created a good model, then your model quickly promotes an understanding of the thing you're modeling. If you've created a bad model, no one knows what you're attempting to convey.

The concept of models should be nothing new; there are an endless number of things in the real world that represent something without really being that thing. Using models is so useful that we tend to forget that we're actually using and/or relying on them. The following list provides a few examples that may give you an idea of what you're missing.

> **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 (or model) of actual women and men. These are bad models because they are an attempt to destroy people's self-image in order to inspire them to consume more crap. I refer to this as "crapitalism".
>
> **Example 2**: Role Models – These models are the people that society expects us to highly revere. 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 are often disappointed when a better description of role models appears in the police blotter[1].
>
> **Example 3**: The Weather Report (weather prediction) –The satellite images indicate there is a storm somewhere and thus that rain arrives a week in advance. Weather forecasters base this prediction on models of previous weather patterns. There's nothing to stop the storm from changing its path, but probabilistically speaking, it will rain.
>
> **Example 4**: Graphical User Interfaces (GUIs) – Practically every computer-type device uses some type of GUI, which contain graphical representations of items such as button, switches, sliders, elevator bars, etc. These items are models of the things they're mimicking. Pixels on a display form models of buttons; the device interacts with the model to make something meaningful occur when something actuates the button model.
>
> **Example 5:** Video Games – 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 much louder and smell funny when you fire them.

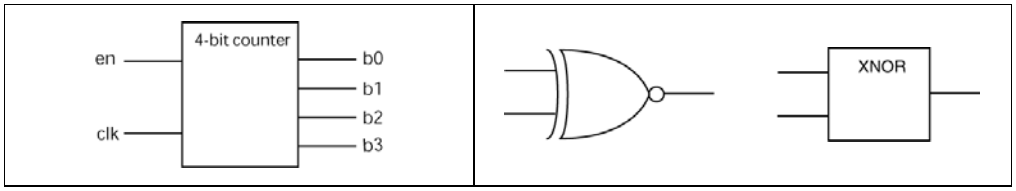
## 3.3 The Black Box Diagram in Digital Design

Digital design uses several different types of models. Recall that there is no one correct model of any given thing; either the model is useful because it helps you understand something, or it's not useful because it provides you with nothing useful. Here's a short list of models we use in digital design; we fill in the details later.

- **The black box diagram**: The black box diagram, or BBD, is a box that graphically shows the inputs and outputs to the digital circuit. Figure 3.1(a) shows an example of a black box model used in digital design. The word "black" in black box has a figurative meaning in that we don't know what's inside the box.
- **The digital circuit element** : We model basic digital devices using special symbols; when digital designers see these symbols, they know how the device operates. Boxes with descriptive labels sometimes replace these special symbols. Figure 3.1(b) shows an example of a digital circuit element model and its corresponding BBD.
- **The timing diagram**: Timing diagrams graphically describe the operational characteristics of a digital circuit based on the status of signals plotted as a function of time. Figure 3.2 shows an example of a timing diagram for some unspecified digital circuit.

[1] Or even worse, as academic administrators.

- **The written description**: We can model the operation of a digital component with a written description. If there is not an accompanying BBD with the written description, the description allows you to generate one. Figure 3.3(b) shows a written description of a digital circuit.
- **The Hardware Description Language (HDL)** : You can use a HDL (Verilog or VHDL) to describe the operation of digital circuits. Figure 3.3(a) shows a VHDL model of a circuit.



**(a)** **(b)**

**Figure 3.1: A BBD (a), and a digital circuit element model with its corresponding BBD (b).**

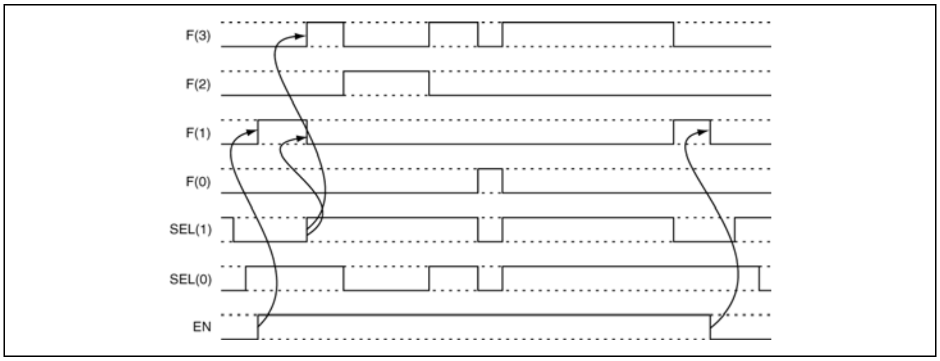


**Figure 3.2: An example of a timing diagram.**

| | |
|---|---|
| <pre>entity dff is<br>   port (D,S,R : in std_logic;<br>           CLK : in std_logic;<br>         Q, nQ : out std_logic);<br>end dff;<br><br>architecture dff of dff is<br>begin<br>   process(D,S,R,CLK)<br>   begin<br>      if (R = '0') then<br>         Q <= '0';  nQ <= '1';<br>      elsif (S = '0') then<br>         Q <= '1';  nQ <= '0';<br>      elsif (rising_edge(CLK)) then<br>         Q <= D;<br>         nQ <= not D;<br>      end if;<br>   end process;<br>end dff;</pre> | 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 3.3: Two functionally equivalent models: an example of a VHDL model (a), and a written description of a digital circuit (b).**

## 3.4 Modeling with Black Box Diagrams

The BBD is useful in designing anything, particularly digital circuits. Unlike modeling techniques such as using an HDL, black box diagrams do not burden you with constraints such as syntax rules. This being the case, don’t forget the overall purpose of models: models quickly transfer information to the reader.

In digital design, we're most concerned about the inputs to and the outputs from digital circuits. We use a box to represent a digital circuit; lines going into and out of the box represent the inputs and outputs of the circuit, respectively. Figure 3.4 shows a few examples of a BBDs. Figure 3.4(a) shows the BBD with inputs and outputs on the left and right sides, respectively. Figure 3.4(b) shows an equivalent model where we indicate the inputs and outputs using arrowheads on the signal lines (arrows entering the box are inputs; arrows leaving the box are outputs). Figure 3.4(c) is another equivalent model that uses "self-commenting" signal names to differentiate the circuit's I/O (input and output).

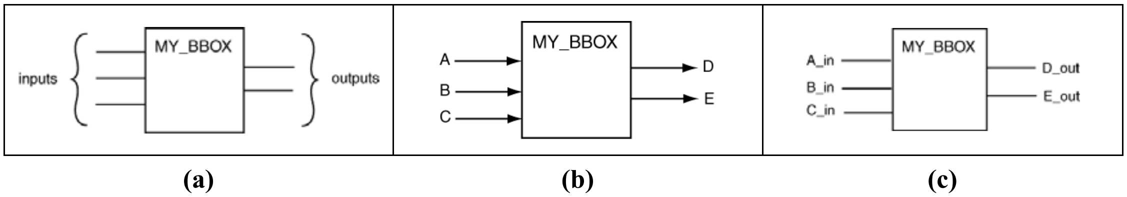


**Figure 3.4: A few examples of basic black box diagrams.**

The models in Figure 3.4 are tough to write about because there are no hard rules for BBDs. However, here are a few strong guidelines you should follow:

- The "flow" of digital models usually goes from left to right[2]. Thus, inputs are on the left side of the box while outputs are on the right side of the box.
- Put arrowheads on signals if it's not obvious what signals are inputs and outputs
- Label all signals unless there is some compelling reason not to
- Place labels on boxes if the reason for the box's existence is not patently obvious

The models in Figure 3.4 represent the first step in black box modeling. One of the hallmarks of any type of design is the ability to abstract the design across many levels. For our purpose, a high-level model of something may not be that useful if we are hoping for a low-level model (and vice versa). These different levels of a model make up a hierarchy of a particular design; each level offers a different type and/or amount of information.

Figure 3.5 shows an example containing two BBDs; the models in Figure 3.6 use these two models. Figure 3.6(a) shows a BBD that sports a two-level hierarchy. The upper-level is the MY_BIG_BOX model; the lower level contains four previously defined models (defined in Figure 3.5). The model in Figure 3.6(b) is somewhat similar to the model in Figure 3.6(a), with some important differences:

- From Figure 3.6(a), we don't know which signals are inputs and outputs by examining the model, as this higher-level model does not contain arrowheads on the signals nor do the signals contain self-commenting names.
- The black box named Z_BOX on the lower level was not previously defined; what's in this box is therefore a mystery and we hope someone defines it elsewhere.
- The interior BBD at the lower level is true to what Figure 3.5 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 characteristics of the Z_BOX remains a mystery.
- The two models in Figure 3.6 are almost identical. The model in Figure 3.6(b) contains less information than the box in Figure 3.6(a) as it does not list the internal connections.

[2] This is not always the case, but following this convention where possible makes your BBDs more readable.

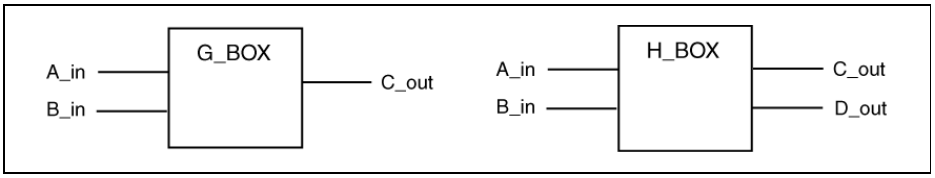


**Figure 3.5: Two example high-level BBDs.**

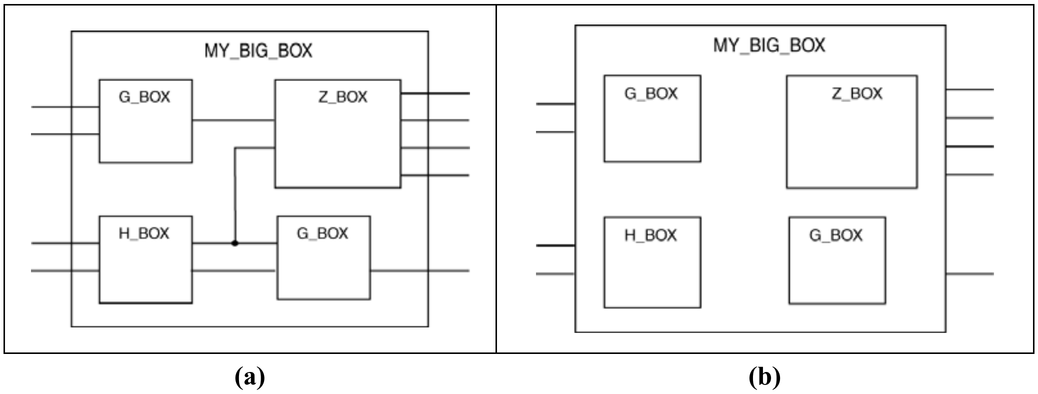


**Figure 3.6: Examples of lower-level BBDs with similar features but varying amounts of detail.**

**Example 3.1: Black-Box Design: Problem Version 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 the vagueness in the description. This is not necessarily a bad thing, particularly if you know nothing about hot water heaters. The problem is expecting you to do something; but you probably won't provide your solution to the Maytag Company for immediate fabrication. In addition, you should definitely become comfortable with the vagueness of the problem statement: bad problem descriptions are typical in most engineering pursuits[3].

The first step in all design problems should be to draw a box and place a somewhat meaningful label on it; Figure 3.7(a) shows the result of this step. This step isn't much, but it's an important starting point, particularly if you have no idea of what to do. Drawing the top-level BBD is the first step in any engineering problem.

The next step is to extrapolate something about solution from the problem statement. You know the water heater heats water; therefore, there must be a cold water input as well as a hot water output. Once you include these in your model, your BBD appears similar to Figure 3.7(b). For the last step, reread the problem description. You know that the heater is a natural gas heater, so it must have an input for natural gas. Figure 3.7(c) shows the model with a gas input.

Because the problem statement did not provide us with direction to the level of detail desired for the solution, we can declare ourselves done. The model in Figure 3.7(c) is somewhat descriptive, especially if you know nothing about water heaters. The moral is that you started with nothing and ended up with an instructive model.

[3] Good descriptions are somewhat contraining. Often descriptions are bad because the person generating the description does not know what they're doing and are expecting to pass the blame to people attempting to interpret their description.

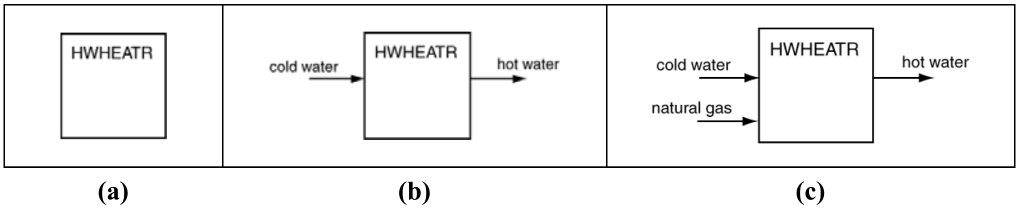


**Figure 3.7: A possible thought process for this example.**

**Example 3.2: Black-Box Design Problem Version 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 need to know something about hot water heaters. The best approach is to realize that you probably know or can figure out how a hot water heater works. Also, note that when this problem asks for subsystems, it's requesting that your solution be hierarchical in nature.

Step One: Let's not reinvent the wheel (the hallmark of all design): borrow from the previous example's solution wherever possible. Figure 3.8(a) shows the result of this step (though we change the name of the black box from the previous example).

Step Two: List all the subsystems that a hot water heater would require. There must be a storage tank for the hot water. There must be a control unit[4] to maintain a constant water temperature by turning on the gas when the water cools and turning it off when it reaches the desired temperature. There is a gas burner, so there must be a fume exhaust (and we maybe should have included it in the previous example). Figure 3.8(b) shows the final solution.

Once again, we declare this problem done. We could do more but we opt not to because our solution satisfies the original problem statement. Our final solution in Figure 3.8(b) shows a two-level hierarchical design; the top-level is the HWHEATER2 module and the lower level shows the three subsystems, which we model as black boxes inside the top-level black box.

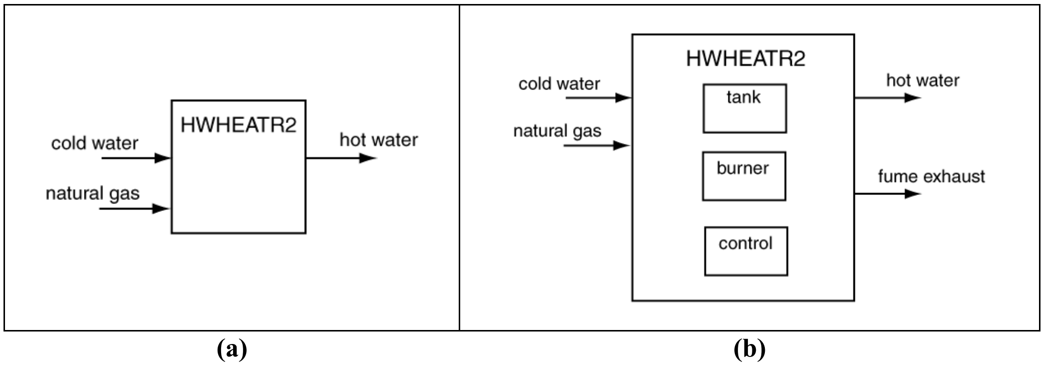


**Figure 3.8: A possible solution to this example.**

[4] 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 a specific temperature.

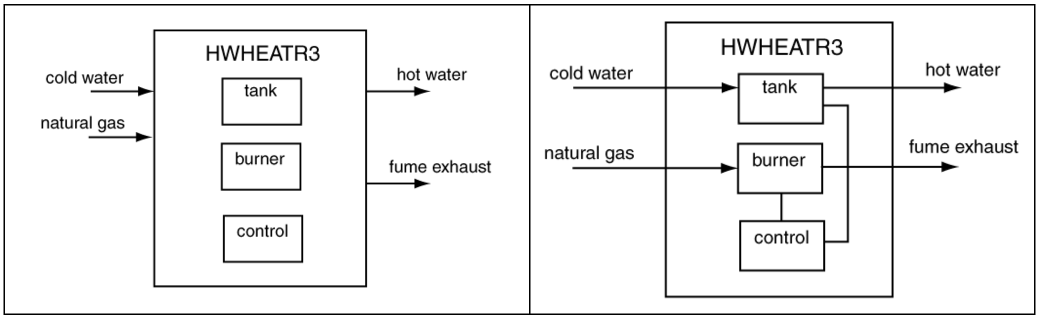
**Example 3.3: Black-Box Design Problem Version 3**

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

**Solution**: This example is a modification to the previous problem. Whereas in the previous example we had to know something about the heater's subsystems, we now must know something about how the subsystems interact with each other.

The problem doesn't state how much detail we needed to include, so we'll add a few details and call the problem done. The first step in the solution is to borrow from the previous example's solution. Figure 3.9(a) shows the result of this step with the top-level BBD now having a new label. The remainder of the solution includes adding internal connections between subsystems, which we describe below. Figure 3.9(b) shows the final result.

- We extend the external connections to the new subsystems. The cold-water input connects to the storage tank. The natural gas input connects to the burner. The hot water output connects to the tank, as that is where the unit stores the water. The fume exhaust connects to the burner, because the burner creates the fumes.
- The control unit is the brain 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 from the tank to the control unit) and tell the burner to turn on/off (another connection goes from the control unit to the burner).



**(a)** **(b)**

**Figure 3.9: A possible solution to this example.**

## 3.5 Black Box Modeling Redux

The previous set of examples highlights the power of black box modeling. Although this example had nothing to do with digital design, the hierarchical design approach in these examples is the mainstay of modern digital design. There are two major things to note about this problem:

- These examples only roughly stated the level of detail required by the problem solution. We did our best without worrying too much about the fact that the U.S. Patent office probably would not like our BBD. We provided what the problem asked for, and then moved on.
- While doing these examples, we started out with nothing and had little knowledge about hot water heaters. When we were done, we had an interesting model of a hot water heater, and we're probably a bit smarter. The black box modeling technique allowed us to take random bits of information and reassemble them in a viable model that satisfied 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 feel like you have no idea where to go.

Black box modeling is the mainstay of digital design. Accordingly, two of the most basic and important digital design principles deal directly with black box modeling:

**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:** Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.

**Mealy's Fourth Law of Digital Design**: The digital design process is circular, not linear. If you think you're going to generate the correct solution with the first pass, you're bound for disappointment. The digital design process is circuit; always make going backwards a few steps to fix issues part of the design process. Don't try to make your design perfect from the get-go, make it simple to understand so that you can fix issues as they arise.

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. The first step in every solution is to draw a top-level BBD that 1) lists what you do know (such as inputs/outputs and given signal name, and 2) labels everything (such as the names of the blocks). The purpose of Mealy's Second law of Digital Design is to prevent you from becoming stuck on a bad design path. If your design is not coming relatively easy, toss it out, rethink it, and start again. Digital design should never be overly complicated. Good digital designers are people who know they are going to make mistakes, but have the wherewithal to quickly correct their issues.

## 3.6 Chapter Summary

- 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, HDL 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 reader.
- Models in general promote an overall understanding of the entity being modeled 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 help 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.
- Three important digital design 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.
  - **Mealy’s Third Law of Digital Design:** Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.
  - **Mealy’s Fourth Law of Digital Design**: Digital design is circular, not linear. Plan on going backwards to correct issues in your design as they arise.

## 3.7 Chapter Exercises

**1)** Briefly explain the general purpose for a model.

**2)** Is there one correct model for anything? Briefly explain your answer.

**3)** Briefly describe the attributes of the “best” model for anything.

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

**5)** 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

## 3.8 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) a 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) refrigerator/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) internal combustion engine

b) soda-dispensing machine

# 4 Number Systems Basics

## 4.1 Introduction

The previous chapters gave you a small taste for the meaning of the terms “digital” and “model”. This chapter continues our move towards digital design by discussing some of the underlying details regarding number systems and their relation to digital design. This chapter introduces number systems.

**Main Chapter Topics**

**NUMBER SYSTEM INTRODUCTION**: Since number usage has become second nature, we probably forgot some of the underlying characteristics that make numbers “work”. This chapter provides a friendly reminder of common definitions associated with number systems.

**COMMON DIGITAL RADII**: We use the binary and hexadecimal number systems in digital design to support hardware implementations of problem involving mathematics. This chapter describes these number systems.

**SPECIAL ATTRIBUTES OF BINARY NUMBERS:** Binary numbers have several properties that we draw upon continuously in digital design. This chapter describes these attributes.

**ENGINEERING NOTATION**: Writing numbers in a clear and concise manner is important in digital design. This chapter describes the motivation behind engineering notation.

**Chapter Acquired Skills**

- Be able to describe the basic vernacular associated with number systems
- Be able to describe the following number systems:
  - Stoneage unary
  - Binary
  - Decimal
  - Hexadecimal
- Be able to describe the important attributes of binary numbers
  - Unsigned binary number ranges
  - Number of bits required to represent positive decimal number
  - Unique numbers representable by a given number of bits
- Be able to represent numbers using engineering notation

## 4.2 Number System Retrospective

Number systems became an integral part of human life, as humans required more viable approaches for quantifying their possessions. Human developed of numbers in order to correct a basic limitation of the human brain: the lack of ability to handle large quantities of “things”.

My eighth grade algebra teacher[1] once told the class a story about some primitive culture. I've forgotten why he told the story, but I never forgot the story itself, as it was the day I found out that I was not much better than a caveperson[2]. The teacher told the class about a number system used by a primitive culture, which 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" quantities of things. Although this caveperson number system seems limited compared to modern number systems, it underscores the limitations of the modern human brain.

Figure 4.1 demonstrates a basic limitation in the human brain. In Figure 4.1(a), it's obvious there is only one dot in the square; your brain both sees and processes this information instantaneously. Your brain probably has no problem "counting" the number of dots in the square of Figure 4.1(b) either. However, once you arrive at Figure 4.1(c), your brain can't instantaneously gather this information: the sheer number of dots in the square instantly overloads your brain. In essence, your brain sees the number of dots as "many"; thus your brain is no more sophisticated than that of a caveperson.

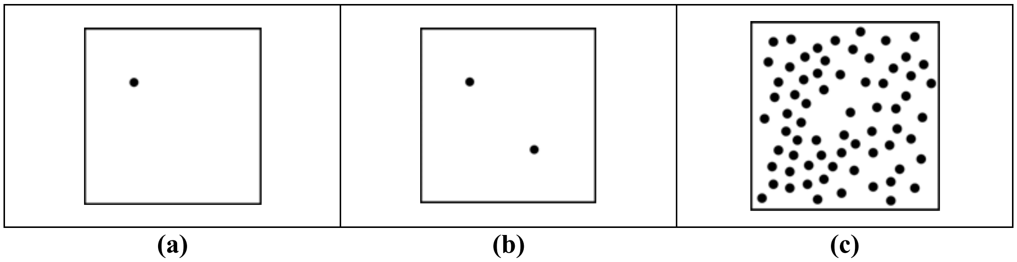


**Figure 4.1: An example showing a basic limitation of the human brain.**

We modern humans are able to both conceive of and process the dots in the square of Figure 4.1(c). We do this by representing the quantity of dots with a "number". We define this number by a mutually agreed upon set of rules to ensure that everyone who uses that number refers to the same quantity of dots. There is even an agreed upon set of squiggles that we use to represent the numbers.

### 4.2.1 Stoneage Unary

Stoneage unary is still a useful and relatively popular number system. When cavepeople realized they needed a more precisely way to track the quantity of things, they started saving a small stone for each thing they possessed. For example, if they had 12 cows, they would store 12 small stones in the pockets of their stone-age loincloths. We call this counting system stoneage unary in that each stone represents a count of one thing. We still often use stoneage unary today with the notion of tick-marks. For example, cowboys cut one groove in the handle of their six-shooters for each person they kill. Similarly, academic administrators carve a notch in their desks for each person they bully, harass, and/or fire.

It is still common to use tick marks to count various things; Figure 4.2 shows an example of such a counting system. This method of counting made it easy to perceive a total number of things by placing tic marks in groups of five things. For example, the number represented by the marks is Figure 4.2 is 23, which also happens to be the average IQ of academic administrators.

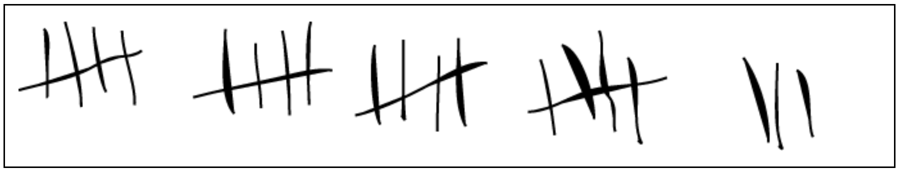


**Figure 4.2: An example of stoneage unary.**

[1] It was Mr. Fangman; the year was 1975. That was really his name.
[2] Mr. Fangman actually used a gender specific term; we'll opt for a gender-neutral term to protect the innocent.

## 4.3 Number Systems Basics

A quick review of the some of the underlying structure and definitions of number systems is in order. The concepts presented in this section should be nothing new to you, but you may have forgotten the actual definitions. Although you're probably able to tweak around with multi-variable calculus but you probably forgot what exactly a radix point is. Welcome to higher education.

- **Number System**: a language system consisting of an ordered set of symbols (digits) with rules defined for various mathematical operations
- **Digit**: a symbol in a number system
- **Radix**: the number of digits in the ordered set of symbols in a number system
- **Number**: a collection of digits, which can contain both a fractional and integral part
- **Radix Point**: a symbol that delineates the fractional and integral portions of a number

As an example, consider a decimal number (radix = ten). Since the number is a decimal number, we can use any one of ten different symbols to represent a decimal number (0, 1, 2, 3, 4, 5, 6, 7, 6, 8, or 9)[3]. If we were only limited to ten numbers, the number system would be of little use to us. However, by placing digits side-by-side and using special rules, we can represent any quantity of things.

Placing digits side-by-side to represent numbers is what we refer to 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 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 4.3 lists some other fun facts regarding numbers and juxtapositional notation. Figure 4.3 shows that we divide numbers into their integral and fractional parts, where the radix point delineates the integral and fractional portions of the number[4]. 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. Figure 4.4 provides an alternative and more formal definition of a number, which includes some of the typical lingo we use to describe numbers.

```
NUMBER = (N)R = (Integral Part) . (Fractional Part)
                                ↑
                           Radix Point
```

**Figure 4.3: The form of a typical number using juxtapositional notation.**

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

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

**NUMBER = $(N)_R = (A_{n-1}\ A_{n-2}\ \dots\ A_1\ A_0\ .\ A_{-1}\ A_{-2}\ \dots\ A_{-m})_R$**

**where:**

**$R \equiv$ radix**

**$A \equiv$ one digit in the number**

**$A_{n-1} \equiv$ the most significant digit (MSD)**

**$A_{-m} \equiv$ the least significant digit)**

**Figure 4.4: Another form of a typical number.**

**Example 4-1: Describing Parts of Decimal Number Representations**

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

**Solution**: “989” is the integral portion of the number; “45” is the fractional portion of the number; the radix point divides the integral and fractional portions of the number. Since there is no listed radix, the radix value of ten is implied and thus the number is a decimal number. We only include a radix if the number has a radix other than ten.

## 4.4 Juxtapositional Notation and Numbers

Juxtapositional notation allows a given number system to represent quantities larger than the “radix-1”. Juxtapositional notation places symbols side-by-side in order to represent quantities larger than the numbers in the given set by assigning a weight to every digit position in the number. By convention, the numbers are monotonically increasing (scanning right to left) powers before the radix point, and, and monotonically decreasing powers of the radix (scanning left to right) after the radix point. The weighting of the digit to the immediate left of the radix point is the radix raised to the zero power while the weighting of the digit immediately to the right of the radix point is raised to power of “-1”.

**Example 4.2: Weightings in Decimal Numbers**

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

**Solution**: Table 4.1 shows the solution to Example 4.2. The radix exponential row uses the radix to monotonically increasing/decreasing powers to designate the weightings. This convention follows the juxtapositional number conventions in Figure 4.4.

| | | | | | | |
|---|---|---|---|---|---|---|
| **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<br>(900) | 8 x 10<br>(80) | 7 x 1<br>(7) | . | 4 x 0.1<br>(0.4) | 5 x 0.01<br>(0.05) |
| | | | | ↑ **Radix Point** | | |

**Table 4.1: The solution to Example 4.2.**

## 4.5 Common Digital Radii

Digital design commonly uses three different radii: 10, 2, and 16. We generally refer to these number systems as “base 10”, “base 2”, and “base 16”, respectively, where the “base” is the radix value; we refer to these number systems as decimal, binary, and hexadecimal (hex), respectively. Table 4.2 lists the justification for using these number systems digital design and include the symbol sets for these three number systems.

| **Name** | **Radix** | **Justification** | **Symbol Set** |
|---|---|---|---|
| Decimal | 10 | It’s what humans understand and is thus comfortable to work with | 0,1,2,3,4,5,6,7,8,9 |
| Binary | 2 | It’s what digital hardware understands and we must be able to work with the hardware | 0,1 |
| Hexadecimal<br>(hex) | 16 | It is a substitute for binary as it helps humans understand long strings of 1’s and 0’s | 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F |

**Table 4.2: Justifications for using particular number systems in digital design.**

Here are a few important things to note about these number systems:

- We refer to binary digits as *bits*
- The number of symbols in each number system spans from ‘0’ to the “radix – 1”. The lowest value symbol is zero and the highest valued symbol is “radix – 1”.
- The hexadecimal system runs out of number symbols after ‘9’, and then arbitrarily switches to the alpha characters of A→F (case does not matter).
- We often list binary numbers in groups of four and “zero-extend”” (add extra zeros) to the values to make then four bits when necessary
- We often refer to a grouping of four bits as a “*nibble*”
- We refer to a grouping of eight bits are a “*byte*”

Table 4.3 shows a list of decimal values 0→15, along with their binary and hex equivalents. We list the binary numbers in groups of four bit as that helps humans quickly identify the numbers. Zero-extending the binary numbers does not change the value of the number.

| (base 10)<br>Decimal | (base 2)<br>Binary | (base 16)<br>Hexadecimal |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

**Table 4.3: Numbers that every digital designer should memorize.**

### 4.5.1 Binary Number System

The binary number system is basis of all transactions in digital hardware because digital circuit hardware (namely transistors) only operates in two states[5]. Therefore, while humans prefer decimal, digital circuits require binary. Using binary represents a new thang, so you need to invest some time in learning the basics of binary (and hexadecimal). The best way to start this is to memorize everything in Table 4.3. This is not a big deal, as you already know decimal, and hexadecimal is straightforward to learn as it only contains six new characters. You need to quickly translate between decimal↔binary↔hexadecimal; generating a table similar to Table 4.3 each time you need to do a conversion is a waste of time. To fluent with powers of two, you also need to memorize the values in Table 4.4.

[5] Transistors can operate in more than two states, but transistors in digital circuits only operate in two states.

| (base 2)<br>Binary | (base 10)<br>Decimal | (base 16)<br>Hexadecimal | (base 2)<br>Binary | (base 10)<br>Decimal | (base 16)<br>Hexadecimal |
|---|---|---|---|---|---|
| $2^0$ | 1 | 1 | $2^0 – 1$ | 0 | 0 |
| $2^1$ | 2 | 2 | $2^1 – 1$ | 1 | 1 |
| $2^2$ | 4 | 4 | $2^2 – 1$ | 3 | 3 |
| $2^3$ | 8 | 8 | $2^3 – 1$ | 7 | 7 |
| $2^4$ | 16 | 10 | $2^4 – 1$ | 15 | F |
| $2^5$ | 32 | 20 | $2^5 – 1$ | 31 | 1F |
| $2^6$ | 64 | 40 | $2^6 – 1$ | 63 | 3F |
| $2^7$ | 128 | 80 | $2^7 – 1$ | 127 | 7F |
| $2^8$ | 256 | 100 | $2^8 – 1$ | 255 | FF |
| $2^9$ | 512 | 200 | $2^9 – 1$ | 511 | 1FF |
| $2^{10}$ | 1024 | 400 | $2^{10}$ - 1 | 1023 | 3FF |

**Table 4.4: Important powers of two that you also need to memorize.**

**Example 4.3: Binary Number Weightings**

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

**Solution**: Table 4.5 shows the solution to Example 4.3.

| | | | | | | |
|---|---|---|---|---|---|---|
| **Decimal 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<br>(4) | 0 x 2<br>(0) | 1 x 1<br>(1) | . | 1 x 0.5<br>(0.5) | 1 x 0.25<br>(0.25) |
| | | | | ↑ **Radix Point** | | |

**Table 4.5: The solution to Example 4.3.**

### 4.5.2 Hexadecimal Number System

The hexadecimal number system contains sixteen digits in its ordered set of symbols. The first ten numbers are the same as decimal numbers, but we use alpha characters (A→F) to represent the numbers 10→15 because we run out of the arbitrary squiqqles we use for the ten decimal digits. Table 4.3 shows the hexadecimal numbers along with the associated decimal and binary numbers (in 4-bit format).

Hexadecimal numbers exist in digital design for one single purpose: they provide a shorthand notation to represent binary numbers. The general rule in digital design is to never use binary numbers greater than four bits because they are too hard for your brain to process. One of the accepted exceptions is when the binary number is all 0's or all 1's.

## 4.6 Important Attributes of Binary Numbers

In digital design, we find ourselves working with special properties of binary numbers. This section introduces and describes a few of these properties.

### 4.6.1 Unique Numbers vs. Number of Bits

Quite often in digital design land, there is an issue of how many unique numbers can you represent by "X number of bits". There's 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 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, etc. The equation in Figure 4.5 shows the relationship between the number of bits and the quantity of unique numbers those bits can represent. The equation in Figure 4.5 essentially describes the second column in Table 4.4.

**Number of unique combination of bits = 2 $^{\text{number of bits}}$**

**Figure 4.5: The relation between the number of bits and number of unique numbers.**

**Example 4.4: Binary Number Characteristics**

How many unique numbers can you represent with an 8-bit unsigned binary number?

**Solution**: The solution to this problem utilizes the formula in Figure 4.5. The quantity of unique numbers = $2^{\text{number of bits}} = 2^8 = 256$

**Example 4.5: Binary Number Characteristics**

How many unique numbers can you represent with a 12-bit unsigned binary number?

**Solution**: The solution to this problem utilizes the formula in Figure 4.5. The quantity of unique numbers = $2^{\text{number of bits}} = 2^{12} = 4096$.

### 4.6.2 Number Range vs. Number of Bits

A given number of bits in an unsigned binary number can represent a range of value, which runs from "all 0's" to "all 1's". For unsigned binary numbers, we interpret all zeros as the decimal value of zero, which represents the minimum value for the range; the maximum value is where all bits are a '1'. Figure 4.6 shows a formula for the number range as a function of the number of bits.

**number range (unsigned binary) for X Bits = [0,2$^{X-}$ 1]**

**Figure 4.6: The range for a given number of bits for unsigned binary numbers.**

**Example 4.6: Binary Number Characteristics**

What is the number range for an 8-bit unsigned binary number?

**Solution**: The solution to this problem utilizes the formula in Figure 4.6. The quantity of unique numbers = $[0,2^8 – 1] = [0,256 – 1] = [0,255]$.

**Example 4.7: Binary Number Characteristics**

What is the number range for a 12-bit unsigned binary number?

**Solution**: The solution to this problem utilizes the formula in Figure 4.6. The quantity of unique numbers = $[0,2^{12} – 1] = [0,4096 – 1] = [0,4095]$.

### 4.6.3 Number of Bits to Represent a Number

Quite often in digital design, we need to know the minimum number of bits we require to represent a given decimal number. For example, we have 100 different items that we need to assign a unique set of bits. In digital design, we most often want to represent that number in a minimum number of bits as well. Figure 4.7 shows the formula; this formula uses a ceiling function.

$$\textbf{Minimum Number of Bits Required to Represent a Decimal Number X} = \lceil \log_2 X \rceil$$

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

**Example 4.8: Binary/Decimal Number Relations**

What is the minimum number of bits required to represent 269 items?

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

Minimum number of bits = ceiling(log2[269]) = 9 bits.

**Example 4.9: Binary Number Characteristics**

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

We derive the two middle numbers from the fact that there is always an even quantity 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. These two numbers effectively divide the 6-bit number into the following two ranges: [0,31] and [32,63]. Note that each range has 32 unique numbers.

## 4.7 Engineering Notation

In order to reduce their workload and thought-load, we typically use engineering notation to represent numerical quantities. Problems can arise when attempting to express numbers without using a convention as Table 4.6 shows, which lists the same number in different but equivalent ways.

| | |
|---|---|
| 0.000034.7 x $10^{2}$ | 0.347 x $10^{-2}$ |
| 0.00034.7 x $10^{1}$ | 3.47 x $10^{-3}$ |
| 0.00347 | 34.7 x $10^{-4}$ |
| 0.0347 x $10^{-1}$ | 347 x $10^{-5}$ |

**Table 4.6: A few ways to represent 34.7 x 10-4.**

The problem is that it’s hard to gather an intuitive feel for numbers if they don’t conform to some standard. The solution is to use engineering notation to represent numbers in digital design. Engineering notation is a subset of scientific notation with some extra rules added. The motivation of using engineering notation is to enhance the intuitive feel of numbers by placing restrictions on their representations.

Engineering notation uses special suffixes to represent the exponential portion of the number. The advantage of engineering notation is that it allows you to obtain a quick feel for the magnitude of numbers based on its magnitude and prefix. Figure 4.8 shows the rules for using engineering notation.

- The magnitude portion of the number should be between 1 and 1000. We officially list this range as [1,1000)[6].
- The units portion of the number uses an appropriate prefix and does not use exponential notation. The valid prefixes are integral multiples of three.

**Figure 4.8: The rules for correctly using engineering notation.**

Table 4.7 lists the prefixes you need to know. There are many others, but how often do you have the unsatisfiable urge to use prefixes such as “yocto”[7]. You should be familiar with most of these prefixes already; but if not, now is your chance to learn some lingo that impresses your friends.

[6] This notation means that the number is greater than or equal to 1 but less than 1000.

| 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 4.7: Engineering Notation prefixes.**

**Example 4.10: Converting a Number to Engineering Notation**

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 need to use the "k" prefix. We then divide the given number by 1000 to obtain the proper magnitude portion of the number before we attach the k prefix. The final answer is 452.3 kHz.

**Example 4.11: Converting Exponential Notation to Engineering Notation**

Represent the value 84.3 x 10-8s in engineering notation.

**Solution**: First, convert the exponential portion of the 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. The resulting magnitude value is then 0.843. However, since this value is less than one, this is not 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 as compensation. The result is 843ns.

[7] Yep, it sounds more like a personal hygiene problem than a prefix.

## 4.8 Chapter Summary

- Engineering notation is a subset of scientific notation and we typically use it to represent numbers when we need to quickly get a feel for the size of the number. 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, which we represent with standard metric prefixes.
- The development of numbers resulted from the need to process larger "quantities" of things. Human brains can't process large quantities of things; "numbers" allows human brains to comprehend and process larger quantities of things
- We use hexadecimal numbers to make long strings of binary numbers more readable to humans.
- Numbers represent quantities that are too big for our brain to understand and process. We form numbers 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. We assign different weightings to digit positions for each position in a number. Numbers have both integral and fractional portions, which we delineate with 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 numbers are 1) the quantity of numbers you can represent by a given number of bits, and, 2) the range of unsigned numbers that you can represent by a given number of bits. These quantities can be represented by closed form formulas:

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

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

**Minimum Number of Bits Required to Represent a Decimal Number X = $\lceil \log_2 X \rceil$**

## 4.9 Chapter Exercises

**1)** Briefly describe why we use hexadecimal numbers in digital design?

**2)** Convert the following values to engineering notation.

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

**3)** Which of the following numbers are have a larger magnitude?

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

**4)** 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?

**5)** Represent the following numbers in two different styles of stoneage unary

- **a)** 3
- **b)** 16
- **c)** 10,456,638

**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)** Show the unsigned binary and hexadecimal equivalents of the following decimal numbers. Use four bits to represent the binary numbers.

- **a)** 7
- **b)** 9
- **c)** 14
- **d)** 2
- **e)** 15

**8)** Briefly described why binary numbers are associated with 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 range.

**10)** Consider a 4-bit unsigned binary number that uses the following weighting (listed from left-most to right-most bits): 5, 3, 2, and 1. (Don't laugh, people actually do things like this).

**a)** List the unique numbers that can be represented by this range.

**11)** How many bits (unsigned binary) does it require to represent the following decimal number?

**a)** 3

**b)** 32

**c)** 129

**d)** 193

**e)** 3999

**f)** 250

**12)** How many unique numbers can be represented by the following number of bits. Also, list the ranges considering the bits represent unsigned binary numbers.

**a)** 6

**b)** 10

**c)** 8

**d)** 9

## 4.10 Design Problems

**1)** Design your own personal number system. This system should have radix of eight. Make sure you define both the symbols and the weighting of numbers based on digit position. The symbols you use in your number system must be unique. Provide a few example numbers and at least one example conversion to decimal.

# 5 Number Systems: Codes and Conversions

## 5.1 Introduction

Digital design uses various codes to represent “things of interest”, such as numbers. There are a bajillion different codes out there, but digital design primarily uses only a few of those codes. Digital designers need to both understand those codes and be able to convert between them. This chapter describes some popular codes and the conversion between these codes.

**Main Chapter Topics**

**CONVERSIONS BETWEEN VARIOUS RADII:** This chapter describes basic algorithms to convert between various number systems.

**Chapter Acquired Skills**

- Be able to convert a number from any radix to decimal
- Be able to convert a decimal number to any radix
- Be able to convert binary numbers to hexadecimal and hexadecimal to binary
- Be able to convert BCD numbers to decimal and back
- Be able to describe and generate a one-hot code
- Be able to describe and generate a unit-distance code

## 5.2 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 need to be able to translate between the various number systems typically associated with digital design.

### 5.2.1 Any Radix to Decimal Conversions

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. An earlier chapter had a few binary to decimal conversions; some more examples follow.

**Example 5.1: Hexadecimal-to-decimal conversion**

Convert $1CE.A4_{16}$ (hexadecimal) to decimal.

**Solution**: Table 5.1 provides the solution to Example 5.1. The solution is similar to a previous example, but with a different radix.

| | | | | | | |
|---|---|---|---|---|---|---|
| **Decimal 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<br>(256) | 12 x 16<br>(192) | 14 x 1<br>(14) | . | 10 x 0.0625<br>(0.625) | 4 x 003906<br>(0.015625) |
| | | | | ↑ Radix Point | | |
| **Final answer:** 256 + 192 + 14 + 0.0625 + 0.003906 = 462.066409 | | | | | | |

**Table 5.1: The solution to Example 5.1.**

### 5.2.2 Decimal to Any Radix Conversion

You can use many different algorithms to convert numbers from decimal to a number system of any radix; this section examines the most straightforward algorithm for humans. This approach works for converting decimal to any base, but we only perform decimal to binary conversions. Note that the best approach to do these conversions is to use a calculator.

The decimal to binary conversion is the conversion you use most often in digital design. There are 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 convert a decimal number to a decimal number (don't worry, it proves a point). The approach we take is to divide the number multiple times by the radix value. Example 5.2 provides an overview of this division process.

**Example 5.2: Decimal-to-decimal conversion**

Convert 487 to decimal.

**Solution**: Table 5.2 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 5.2: Decomposing an integral decimal number into a decimal number.**

You can see in Example 5.2 repeated division by the radix value decomposes the original value into its individual weighted components. The first value that this algorithm generated was the least significant digit (LSD) which is 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!

This example proves that the algorithm is valid and it thus works when transferring from decimal to a number of any radix value. In both examples, we use the terms LSB and MSB, which refers to Least Significant Bit and Most Significant Bit, respectively. Not surprisingly, the technique we refer to this technique as repeated radix division (RRD).

**Example 5.3: Decimal-to-Binary Conversion**

Convert 12 to binary.

**Solution**: Table 5.3 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 5.3: The solution to Example 5.3: decomposing a decimal number into a binary number.**

**Example 5.4: Decimal-to-Binary Conversion (integral)**

Convert 147 to binary.

**Solution**: Table 5.4 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:
14710 =$10010011_2$

**Table 5.4: The solution to Example 5.4**

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. The approach we take is to multiply the number repeatedly by the radix value and examine the result. In each step, we'll peel off the newly created integral portion of the number and put it aside. Example 5.5 provides an overview of this algorithm. Note from the result in Example 5.5 that the first integral result is the MSD of the original number. The final value we obtain is the LSD of the original number. We refer to this algorithm to as repeated radix multiplication (RRM).

There are two key points about the example in Example 5.7. First, as opposed to the example in Example 5.6, the example in Example 5.7 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 is that the answer is no longer a proper equation. In reality, since our conversion never ended as nicely as in Example 5.6, 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.

**Example 5.5: Decimal-to-decimal conversion (fractional)**

Convert 0.243 to decimal.

**Solution**: Table 5.5 shows the solution 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 5.5: Solution to Example 5.5**

**Example 5.6: Decimal-to-Binary Conversion (fractional)**

Convert 0.375 to binary.

**Solution**: Table 5.6 shows the solution to this example in a few steps starting with the top row of the table being the first step.

| | | |
|---|---|---|
| 0.375 × 2 = 0.75 | remove the 0 | MSB = 0 |
| 0.75 × 2 = 1.50 | remove the 1 | |
| 0.5 × 2 = 1.0 | remove the 1 | LSB = 1 |

0.375 = 0.0112

**Table 5.6: Solution to Example 5.6**

**Example 5.7: Decimal-to-Binary Conversion (fractional)**

Convert 0.879 to binary.

**Solution**: Table 5.7 shows the solution to this example in a few steps starting with the top row of the table being the first step.

| | | |
|---|---|---|
| 0.879 × 2 = 1.758 | remove the 1 | MSB = 1 |
| 0.758 × 2 = 1.516 | remove the 1 | |
| 0.516 × 2 = 1.032 | remove the 1 | |
| 0.032 × 2 = 0.064 | remove the 0 | LSB = 0 (?) |

0.879 ≈ 0.11102

**Table 5.7: Solution to Example 5.7**

### 5.2.3 Binary ↔ Hex Conversions

The key to converting between binary and hex numbers is to note that a single hex number represents 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 of Example 5.8 and Example 5.9 highlight the relationship between the group of fours in the context of a binary-to-hexadecimal conversion and a hexadecimal-to-binary conversion, respectively. In addition, there are a few special items to note in these examples.

**Example 5.8: Binary-to-hexadecimal conversion**

Convert $1100110.10101_2$ to hexadecimal.

**Solution**: Figure 5.1 shows the solution to Example 5.8; here is some cool stuff to note:

- We omit the leading zeros in the number as they have no value
- We add zeros 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', but the number has the weight associated with the MSB of a 4-bit binary number. The final bit is associated with a hexadecimal '8' and not '1'.

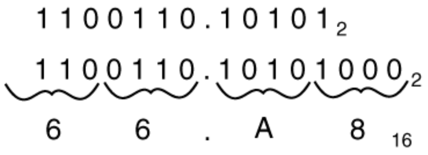
| | |
|---|---|
|  | $1100110.10101_2 = 66.A8_{16}$ |

**Figure 5.1: The solution to Example 5.8.**

**Example 5.9: Hexadecimal-to-binary conversion**

Convert $D37.AC_{16}$ to binary.

**Solution**: Figure 5.2 shows the solution to Example 5.9.

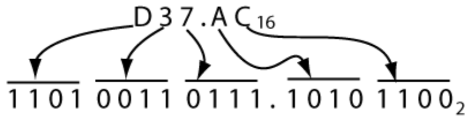
| | |
|---|---|
|  | D37.AC16 = $110100110111.10101100_2$ |

**Figure 5.2: The solution to Example 5.9.**

## 5.3 Fast Radix-Based Division & Multiplication

Division and multiplication are usually complex operations in any radix. When the divisor or multiplicand is the radix raised to an integral power, these multiplication and division are trivial. We're all familiar with the notion of dividing or multiplying decimal numbers by powers of ten, where all we do is move the decimal point around and added extra zeros where necessary. This ease of operation is no different for other radii, namely binary and hexadecimal. The best way to show this is with a few examples.

**Example 5.10: Binary Division**

Divide the following value by 8: $1100110.101_2$

**Solution**: First, 8 is an integral power of 2 ($2^3=8$). This means that we need to move the radix point three digits to the left, which is the same operation as dividing by 8. The final answer is $1100.110101_2$.

**Example 5.11: Binary Multiplication**

Multiply the following value by 32 $111011.001_2$

**Solution**: First, 32 is an integral power of 2 ($2^5=32$). This means that we need to move the radix point five digits to the right, which is the same operation as multiplying by 32. The final answer is $11101100100_2$.

**Example 5.12: Hexadecimal Division**

Divide the following value by 256: $3AD7.B_{16}$

**Solution**: First, 256 is an integral power of 16 ($16^2=256$). This means that we need to move the radix point two digits to the left, which is the same operation as dividing by 256. The final answer is: $3A.D7B_{16}$.

**Example 5.13: Hexadecimal Multiplication**

Multiply the following value by 256: $CDF.8_{16}$

**Solution**: Note that 256 is an integral power of 16 ($16^2=256$). This means that we need to move the radix point two digits to the right, which is the same operation as multiplying by 256. The final answer is: $CDF80_{16}$.

## 5.4 Other Useful Codes

Using binary patterns to represent numbers is a major field of study in modern engineering.. In that we currently live in the information age, there are an endless number of binary codes in use.

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). As you'll soon find out, this is only true for unsigned binary numbers in one particular format; we need other number representations to be fluent in digital-land.

### 5.4.1 Binary Coded Decimal Numbers (BCD)

Binary coded decimal (BCD) numbers are similar to the group of fours. 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 need at least four bits to uniquely represent each of the decimal digits. We could not represent the set of decimal numbers with three bits because that only provides eight different unique bit patterns. On the other hand, there is nothing stopping us from using more than four bits to represent the digits but that would end up having lots of unassigned codewords. 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[1].

Table 5.8 shows the four-bit code words and the decimal digits they represent. The primary role of BCD numbers is to represent decimal numbers in devices that display numbers.

| 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 |

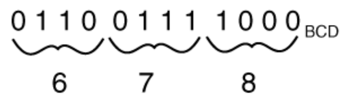
**Table 5.8: The decimal digits and their associated BCD codes.**

**Example 5.14: BCD-to-decimal conversion**

Convert $011001111000_{BCD}$ to decimal.

**Solution**: Figure 5.3 shows the solution to Example 5.14.

[1] Although these six combinations are often used to represent "numbers" 10-15 hexadecimal.



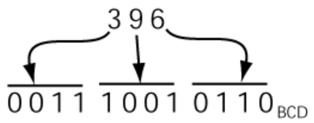
$011001111000_{BCD} = 678$

Figure 5.3: The solution to Example 5.14.

**Example 5.15: Decimal-to-BCD conversion**

Convert 396 to BCD.

**Solution**: Figure 5.4 shows the solution to Example 5.15.



$396 = 001110010110_{BCD}$

Figure 5.4: The solution to Example 5.15.

### 5.4.2 One-Hot Codes

The idea of a one-hot code is simple: for a codeword of n-bits in length, only one of the bits is '1' at any given time; the other bits are zero. Table 5.9 shows examples of 3, 4, 5, and 6-bit one-hot codes; creating one-hot codes is relatively simple.

Two areas in digital design use one-hot codes. First, they are the outputs of a "standard decoder", which is a device we discuss in an upcoming chapter. Second, we typically use one-hot codes in the low-level design of finite state machines (FSMs). In this text, we do a majority of FSM design at a high level, so we won't visit the topic in a significant manner in the remainder of this text. There are also "one-cold" codes; these codes share the same properties as one-hot codes, except the codeword's bits are inverted (changed to '1' if '0', or changed to '0' if '1').

| 3-bit<br>One-Hot Code | 4-bit<br>One-Hot Code | 5-bit<br>One-Hot Code | 6-bit<br>One-Hot Code |
|---|---|---|---|
| 001 | 0001 | 00001 | 000001 |
| 010 | 0010 | 00010 | 000010 |
| 100 | 0100 | 00100 | 000100 |
| - | 1000 | 01000 | 001000 |
| - | | 10000 | 010000 |
| - | | | 100000 |

**Table 5.9: Examples of 3, 4, 5, and 6-bit one-hot codes.**

### 5.4.3 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 were given set of binary code words of equal length; the set of codes also has a specified sequence (such as a binary count). In this context, each of the code words is different from all of the other code words in the set. This set of code words now has order, uniqueness, and a constant bit-length, so we can discuss 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 5.10 shows an example of a 5-bit binary code. Table 5.10 shows that we define the distance between two code words as the number of bits that you must toggle (invert) to form one code word out of another contiguous code word in the set.

| Code<br>Word A | Code<br>Word B | Distance from<br>Word A to Word B | Comment |
|---|---|---|---|
| 00000 | 11111 | 5 | Toggle all bits |
| 01110 | 00110 | 1 | Toggle second bit from right |
| 00110 | 00110 | 0 | Toggle no bits |
| 00111 | 11100 | 4 | Toggle outer two bits |

**Table 5.10: 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 contiguous 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.

There is a science to creating UDCs but we'll not go into that. Know when you hear the words "unit distance", that it's describing a relationship between two binary numbers. There is also a special form of UDCs that we refer to as *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. Table 5.11 lists a few UDC examples.

| 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 5.11: Examples of 2, 3, and 8-bit UDC codes.**

## 5.5 Chapter Summary

- Hexadecimal (base 16) and binary (base 2) are two of the primary number systems commonly used and associated with digital design. We use hexadecimal to make long strings of 1's and 0's more readable.
- We often require conversion between various types of numbers associated with digital design. The important most common conversions are decimal-to-binary, binary-to-decimal, hexadecimal-to-binary, and binary-to-hexadecimal. We perform these conversions using special algorithms.
- Binary coded decimal (BCD), unit distance codes (UDCs), and one-hot codes are three codes we commonly use in digital logic design.

## 5.6 Chapter Exercises

**1)** Explain briefly but fully why the group of four approach works for converting number between hexadecimal and binary representations.

**2)** Complete the following number systems conversions:

- **a)** $011110010001_{BCD}$ to decimal
- **b)** $0001000000110110_{BCD}$ to decimal
- **c)** 4377 to BCD
- **d)** 70023 to BCD
- **e)** $4AC_{16}$ to decimal
- **f)** $782B_{16}$ to decimal
- **g)** $10110_2$ to decimal
- **h)** $101111_2$ to decimal

**3)** Complete the following mathematical operations

- **a)** 110110112 * 8
- **b)** 10110110 ÷ 16
- **c)** 3AB16 * 8
- **d)** 4A7F ÷ 32

**4)** What is the minimum radix value of the following number?: 145.801

**5)** What is the minimum radix value of the following number?: BA.12

**6)** Which of these two positive numbers is greater? $100110110.1100_2$ or $15B.B_{16}$

**7)** Assemble these numbers into a gray code sequence: 111, 000, 110, 011, 001, 100

**8)** Can the following set of number be made to form a gray code?

0011, 0110, 1100, 0111, 1111, 1110, 0001

**9)** What is the maximum distance between any two of the following numbers?

0011, 0110, 1100, 0111, 1111, 1110, 0001.

**10)** In the table below, cross out one code word from each column to make the code 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 |

**11)** In the table below, add one code word to each column to make the code 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 | | |

**12)** The table below shows five binary codes. Circle the codes that are unit distance codes.

| | | | | |
|---|---|---|---|---|
| 000 | 0000 | 0000 | 01000 | 00000 |
| 001 | 1000 | 0001 | 01001 | 00100 |
| 011 | 0100 | 0011 | 01011 | 01100 |
| 111 | 0010 | 0010 | 01111 | 01110 |
| 110 | 0001 | 0110 | 11111 | 11111 |
| 100 | | 0100 | 01111 | 11110 |
| | | 1100 | 01110 | 11100 |
| | | 1000 | 01100 | 11000 |
| | | | 00100 | 10000 |
| | | | 00000 | |

**13)** Show the one-hot codes for the following number of bits:

**a)** 3

**b)** 6

**c)** 8

**14)** Show a 16-bit one-hot code in hexadecimal.

**15)** Divide the following number by 256: 3 5 F D $1_{16}$.

**16)** Divide the following number by 32: 1 0 1 1 0 1 0 0 0 1 0 0 1 1 $1_2$.

**17)** Multiply the following number by 256: A473.$1_{16}$.

**18)** Multiply the following number by 2048: $B321.A2_{16}$.

**19)** Multiply the following number by 64: $110110.10_{2}$.

**20)** Multiply the following number by 256: $110.1001_{2}$.

## 5.7 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 unit distance code that you can use to represent a re-design of a BCD code. Your new code should be a four-bit code and represent all numbers from 0→9.

**3)** Design a unit distance code that you can used to represent a re-design of a standard binary code. Your new code should be a four-bit code and represent all numbers from 0→15.

**4)** Design 8-bit two-hot code. For this code, each code word has only two bits set. Any given bit is only set in one of the codewords. In what applications would this code be potentially useful?

**5)** Design an 8-bit two-hot code that contains six different codewords. For this code, each code word has only two bits set.

# 6 Brute Force Digital Design

## 6.1 Introduction

This chapter is the first chapter covering true "digital design". This chapter presents a single approach to digital design, but it is by no means the only approach. This chapter presents a model for solving digital design problems.

**Main Chapter Topics**

**DIGITAL DESIGN OVERVIEW**: This chapter uses a design example to introduce a simple digital design process: the "iterative", or "brute force" approach to digital design.

**BOOLEAN ALGEBRA**: This chapter introduces Boolean algebra including its basic axioms and associated theorems.

**Chapter Acquired Skills**

- Be able to describe the purpose of a logic gates and inverters
- Be able to describe truth tables for both AND gates, OR gates, an inverters
- Be able to model solutions to digital design problems by using specifying input/output relationships in equation form and circuit forms.

## 6.2 Digital Design

Being the average smart person, you've solved many problems during your life. However, have you ever analyzed your approach to solving problems? The following verbage lists the approach that I generally take to solving a problem. This approach is generic enough to be applicable to any problem. Here is my basic algorithm for solving problems.

**a)** Define the problem: understand the starting point and requirements

**b)** Describe your solution to the problem: propose a path to the solution

**c)** Implement your solution to the problem: embodiment of the solution

The following verbage represents an introduction to digital design that we present in the context of an actual problem. We're designing a digital circuit; you would take a different approach if you were designing a stick in the mud. The basic concept of all digital design is simple: you're creating a circuit that provides the correct output(s) to a given set of input(s)[1]. There are many approaches to performing digital design; this section presents only one of them. You'll find that you eventually develop your own style and approach to digital design as you gain more experience. You'll initially be on a mission to collect tools and experience with digital design.

[1] What you see later in this test is that the "correct" 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.

### 6.2.1 Step 1: Defining the Problem

The basis of any design problem is a relatively clear statement of the problem. In digital design, you typically face the notion of designing a digital circuit that processes some set of inputs and generates the desired output. Example 6.1 provides the problem statement for this painfully long design example.

**Example 6.1: The First Design Problem**

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

**Solution**: The first step in defining the problem is to translate what the problem is asking into another form. The best place to start with all digital design problems is to draw a BBD that shows the circuit's inputs and outputs. You can see from the problem statement that the digital circuit has three inputs (the 3-bit binary number) and one output (indicates a quality of the inputs). A model in this context is a description of a digital circuit, which we loosely define in Figure 6.1(a).

The diagram of Figure 6.1(a) shows that our final circuit has three inputs and one output. Figure 6.1(b) shows another model of our final circuit. The main difference between these two models is the fact that the model in Figure 6.1(b) has given specific names to the inputs and outputs. The circuit models of Figure 6.1(a) and Figure 6.1(b) shows the same thing but the Figure 6.1(b) provides a greater amount of detail.

The model of Figure 6.1(b) is better because we need to use the signal names to solve this problem. The signal names applied to the model in Figure 6.1(b) are nothing special: the "**B**" could mean binary; the numbers following the B's are probably 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.

There is some important information missing from the model of Figure 6.1(b): since the three inputs represent a binary number, we need to know the weights associated with each bit. The solution needs to state this this information in order for the model of Figure 6.1(b) solution to have meaning. Let's consider the **B2** input to be the most significant bit (MSB) and the **B0** input to be the least significant bit (LSB). You must always state this extra information in your digital design solutions. A good model of anything prevents the reader of that model from assuming anything, so always state any assumptions as part of the model.

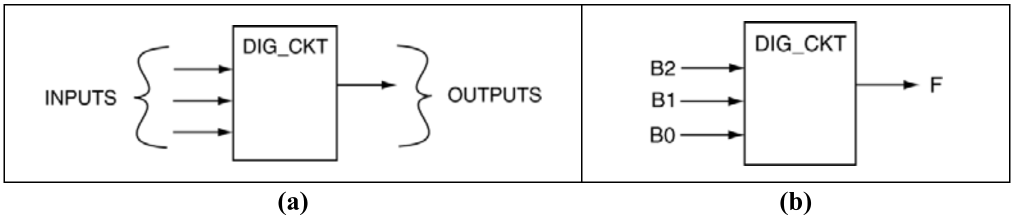


**Figure 6.1: Two different models of the proposed digital circuit.**

The next step is to establish a relationship between the circuit's inputs and outputs. The approach we take is to show an input/output relationship in such a way as that we are essentially solving the given problem. The way we 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 problem. We refer to the table that displays this input/output relationship as a *truth table*. Figure 6.2(a) shows the empty truth table while Figure 6.2(b) shows the truth table with every possible combination of the three binary inputs; the output indicates when the input combination solves the 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 6.2: The empty and completed truth table for Example 6.1.**

The following describes some of the important things to note about the truth tables in Figure 6.2.

- Figure 6.2(a) shows an empty truth table while Figure 6.2(b) shows a truth table containing many 1's and 0's. Digital circuitry and digital models typically use 1's and 0's to model the voltages that drive the underlying hardware, so this allows us to abstract past the need to deal with voltages. We model 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 associated truth table. Since there are three inputs, there are $2^3$ unique combinations of the three inputs. 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 a function , which 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. The value of **F** is dependent upon the values of the **B2**, **B1**, and **B0** inputs. The output **F** has only one value for each possible input combination, which preserves the functional relationship.
- 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 appears on the inputs. For this example, we entered 0's for the cases where the inputs bits represent a number less than five. We enter 1's for the cases where the input combination is greater than four.
- 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. We typically divide truth tables into rows of four.

The problem is now 100% defined using the truth table in Figure 6.2(b). In case you're thinking that this problem is somewhat straightforward in the way that we specified the outputs, you're correct. This particular style of digital design is an exhaustive approach in that the truth table lists every possible input combination. We refer to this approach as the iterative approach to digital design, but we refer to it as BFD (brute force design). Would an iterative approach be possible if the circuit had 24 inputs? No! Therein lays the basic limitation of the iterative approach.

### 6.2.2 Step 2: Describing the Solution

Although the truth table has completely defined the solution to this problem, it is somewhat klunky to work with, especially as the number of inputs increase. What we need to do is develop a "science" of sorts in order to more efficiently describe the problem's solution. Lucky for us that someone a long time ago already developed the "science" we're looking for. Here's the shortened version of the story.

About a bajillion years ago, George Boole developed some methods to deal with a two-valued algebra[2]. Although his original intent was to model logical reasoning in a mathematical context, his work currently forms the basis for all digital design. We refer to this two-valued algebra as Boolean algebra. Boolean algebra uses a basic 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 (a binary thang).

Table 6.1 lists the basic axioms of Boolean algebra. The axioms completely define the basic operators in Boolean algebra: the dot (•), the cross (+), and the overbar ( ¯ ). Table 6.2 and Table 6.3 list the Boolean algebra theorems; we can prove these theorems using the axioms in Table 6.1[3].

| | | | |
|---|---|---|---|
| **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** | $\overline{0}=1$ | **4b** | $\overline{1}=0$ |

**Table 6.1: 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** | $\overline{\overline{x}}=x$ | | | **Double Complement** |
| **9a** | $x\cdot\overline{x}=0$ | **9b** | $x+\overline{x}=1$ | **Inverse** |

**Table 6.2: 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** | $(x\cdot y)+(x\cdot\overline{y})=x$ | **14b** | $(x+y)\cdot(x+\overline{y})=x$ | **Combining** |
| **15a** | $\overline{(x\cdot y)}=\overline{x}+\overline{y}$ | **15b** | $\overline{(x+y)}=\overline{x}\cdot\overline{y}$ | **DeMorgan’s** |

**Table 6.3: Two and three-variable theorems.**

The most important result gathered from the basic axioms of Table 6.1 is the definition of the three operators. Although the axioms completely define these operators, the definition of these operators is clearer when represented in a truth table. The three operators have names: we refer to the dot operator (•) as the AND operator as it defines an AND operation (to as *logical multiplication*). We refer to the cross operator (+) as the OR operator as it defines an OR operation (*logical addition*). We refer to the overbar as the NOT operator as it defines a NOT operation (usually referred to as *inversion* or *complementation*). Table 6.4 shows the truth tables associated with these three operator definitions; we generate these truth tables from the basic axioms.

[2] 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. The world refers to these basic assumptions 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 proven true from axioms.

[3] 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.

**AND (logical multiplication)**

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

**OR (logical addition)**

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

**NOT (inversion)**

| x | $F = \overline{x}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Table 6.4: Truth tables for the three basic logical operators.**

The goal of this section is 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 6.3 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 6.3: The truth table for the original problem.**

We now have several different ways of describing the function that solves the problem at hand. The first representation is the truth table, which we know as being rather klunky. A second solution is sort of a verbal and thus non-scientific solution. Figure 6.4 shows the long and drawn out text of this verbal solution. Notice that Figure 6.4 extensively uses of the words "and" and "or" in the solution. However, since we went to all the trouble to describe Boolean algebra, Figure 6.5 shows a better (more efficient and scientific) way to describe the function using Boolean algebra. Note the similarities in the solutions of Figure 6.4 and Figure 6.5.

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 6.4: One approach to describing the solution to Example 6.1.**

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

**Figure 6.5: A better approach to describing the solution to Example 6.1.**

There are several important things to note about the equation in Figure 6.5.

- This is truly an equation (note the presence of the equal sign). We refer to this equation as a Boolean equation or sometimes as a Boolean expression. We wrote the expression in functional form where we list the complete set of independent variables on the left side of the equals sign and we list the dependent value on the right of equals sign.
- The expression implies some form of precedence of the AND, OR, and NOT operators. The NOT operator has highest precedence followed by the AND, and then the OR operator. We write these

Boolean expressions using parenthesis around the individual terms that are ANDed together[4]. Figure 6.6 shows an example of the equation of Figure 6.5 with a refreshing use of parentheses.

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

**Figure 6.6: An arguably better approach to describing the solution to Example 6.1.**

### 6.2.3 Step 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 word implement has many connotations; what we mean in this context is that we need some way to implement this function in actual hardware[5]. All you currently know are the basic functions associated with Boolean algebra: AND, OR, and NOT.

There are entities out there we refer to as "logic gates" that implement the individual logic functions. Just as there are AND, OR, and NOT functions, there are also physical circuits (AND, OR, and NOT gates) that implement these functions. A logic gate is a physical device that implements a logic function. Figure 6.7 shows model for these three basic gates. In other words, the gates represent the associated logic functions but without providing details as to the function's implementation on the transistor level.

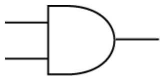
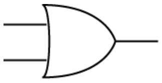
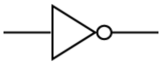
| AND gate | OR gate | Inverter |
|---|---|---|
|  |  |  |

**Figure 6.7: The basic gate symbols used to model AND, OR, and NOT functions.**

AND gates and OR gates must have at least two inputs but don't have a maximum number of inputs. In the cases of more than two inputs, the functions remain consistent. Figure 6.8 lists a more generic definition of AND & OR gates; these definitions completely describe the functionality of these gates when they have more than two inputs[6]. AND & OR gates can have as many inputs as they need while still exhibiting the basic AND & OR functionality. Inverters can only have one input and one output.

- ***AND gates and OR gates can have only one output.***
- ***Inverters can only have one input and one output.***

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

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

**Figure 6.8: A more generic and intuitive definition for AND & OR functions.**

These gates give us the ability to implement the solution in 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 that solves this problem. Figure 6.9 shows a model of the final circuit implementation. Make sure you understand the relationship between the circuit model of Figure 6.9 and the Boolean equation in Figure 6.5. To test your understanding of this relationship, you should be able to generate the associated Boolean equation in Figure 6.5 that describes the circuit from the circuit model in Figure 6.9.

[4] Use of parenthesis reduces the need to memorize operator precedence. So, if in doubt, use parenthesis.

[5] A digital design synonym for implementing a function in hardware is to "realize" the function or "function realization".

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

Non-complemented signals in the Boolean equation connect directly to the gates, while signals with overbars (complemented signals) pass through inverters; the output of the inverter connects to the gates. The equation contains three terms where the signals are ANDed together. The output of the three associated AND gates form the three inputs to the OR gate. The output of the OR gate is the circuit's final output.

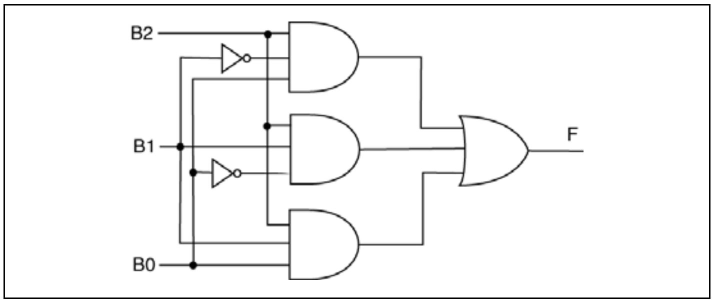


**Figure 6.9: The circuit model that solves Example 6.1.**

You should be able to go back and forth between the various representations of a Boolean function. In this example, we worked with four different representations of a Boolean function: 1) truth table, 2) written description, 3) Boolean equation, and 4) a circuit model. There are many more ways to represent a Boolean function. Each of these representations is a model of a digital circuit. Given any one of these models, you can 1) generate any of the other models, and 2) implement the circuit.

An important issue to realize about this circuit is that is has no control feature. Most of the circuit we study in digital design have one of four types of control: 1) no control, 2) internal, 3) external, or 4) by a controller circuit. The circuits we've study so far have no control: the outputs simply react to the inputs.

**Example 6-2: Generic Design #2**

Design a circuit that has four inputs (**A**, **B**, **C**, **D**) and two outputs (**F**, **EVEN**). All inputs and outputs are single bits. The four inputs represent a binary number where **A** is the MSB and **D** is the LSB. The F output indicates when the 4-bit input value is odd and has two and only two bits set. The EVEN output indicates when the input value is even. Provide a top-level BBD and a lower-level circuit diagram for your circuit solution. Also, state what type of control the circuit uses.

**Solution:** The first step is to generate a BBD for the solution, which we show in Figure 6.10. The circuit has four single-bit inputs and two single-bit outputs.

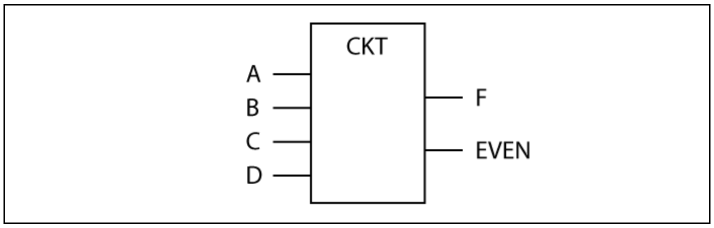


**Figure 6.10: The top-level BBD for this example.**

The next step is to define the solution using a truth table. This problem has four single-bit inputs, which requires that the truth table have $2^4$ or 16 rows. This problem has two outputs, which we represent with two separate columns in the truth table. Table 6.5 shows the final truth table with completed **F** and **EVEN** columns.

| A | B | C | D | F | EVEN |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

**Table 6.5: The truth table for the solution.**

Figure 6.11 shows the final equations for the solution. We took a straightforward approach for the **F** equation; we list the inputs associated with the given row in the truth table where the **F** output is a '1'. We could have done the same thing for the **EVEN** output, but that would have created an equation containing eight sum terms, which is gruntwork we try to avoid. While it's comfortable to follow rules when solving digital design problems, you must always use some horse sense, which is what we did for the **EVEN** output. We noted that the **EVEN** column is an inversion of the **D** input column; we can easily represent this by writing an equation that equates an inverted **D** input to the **EVEN** output. This is a shortcut, but it represents something you should always look for when solving problems. Figure 6.12 shows the final circuit model for this example.

$$EVEN = \overline{D}$$

$$F = \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + A\bar{B}\bar{C}D$$

**Figure 6.11: The final equations for this example.**

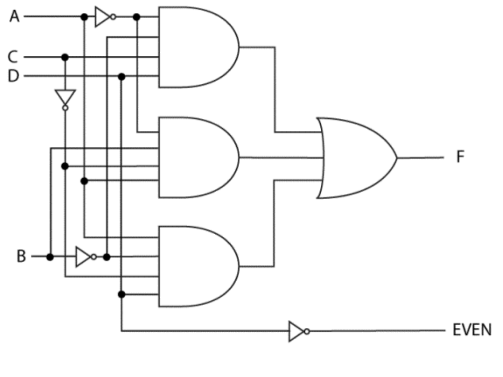


**Figure 6.12: The final circuit model for this example.**

## 6.3 Chapter Summary

- The need to solve a problem drives the creation of a digital circuit. We can describe the basic process of digital design in three steps: 1) define the problem, 2) describe the solution, and 3) implement the solution. We can describe solutions to digital design problems Boolean equations, which have their basis in Boolean algebra.
- There are many possible ways to represent solutions to digital design problems. We consider these many solutions to be 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.
- Four axioms define the basic operation of Boolean algebra. Those axioms define the basic logic operators of AND, OR, and INVERSION.
- There is relatively long list of Boolean algebra theorems associated with Boolean algebra. Some of these theorems are quite useful in digital logic while we rarely apply others.
- Digital design uses logic gates to implement basic Boolean operators in hardware.

## 6.4 Chapter Exercises

**1)** What entity forms the basis of iterative design? Briefly explain

**2)** Why is it that you have to learn something as inefficient as iterative design? Briefly explain

**3)** Why is the term “brute force” associated with iterative design? Briefly explain.

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

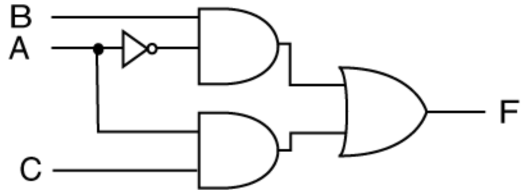
c) $F(X,Y,Z)=(\overline{X}+\overline{Y}+Z)\cdot(\overline{X}+Y+\overline{Z})\cdot(X+\overline{Y}+Z)\cdot(\overline{X}+\overline{Y}+\overline{Z})$

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

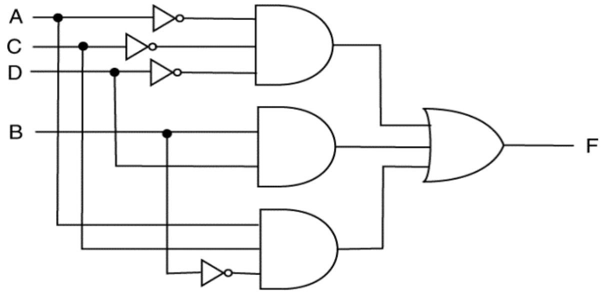
a) $F(X,Y,Z)=\overline{X}\cdot\overline{Y}\cdot Z+X\cdot\overline{Y}\cdot\overline{Z}+X\cdot Y\cdot\overline{Z}+X\cdot Y\cdot Z$

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

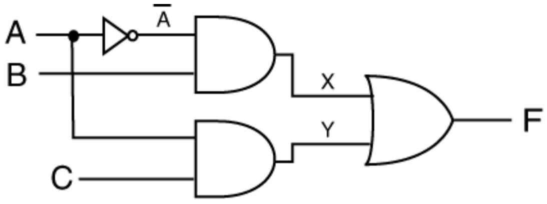
**10)** Write a Boolean equation that describes the following circuit:



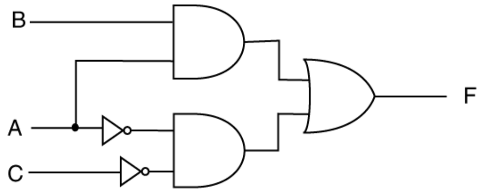
**11)** Write a Boolean equation that describes the following circuit:



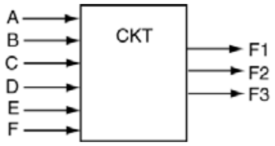
**12)** Write a Boolean equation that describes the following circuit:



**13)** Write a Boolean equation that describes the following circuit:



**14)** If a truth table were constructed in order to define the input/output relationship of the circuit represented by the following schematic diagram, how many rows would the truth table have? Briefly explain your answer.



## 6.5 Design Problems

In addition to solving each of the problems below, state whether the circuit has “no control”, “internal control”, or “external control”. Model the final circuit using AND gates, OR gates, and inverters.

**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.

**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.

**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’.

**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.

**6)** Design a circuit that translates a 4-bit stoneage unary code to an unsigned binary code.

**7)** Design a circuit that translates a 4-bit one-hot code to an unsigned binary code. Consider the unsigned binary number on the output to indicate the bit position of the set bit in the one-hot code, where the right-most bit in the one-hot code is the “zero” position.

**8)** 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’.

**9)** You’re the owner of a clothing store that has three dressing rooms. Each dressing room has a sensor that indicates (with a ‘1’) when a dressing room is occupied (a ‘0’ indicates the dressing room is empty). Provide a block diagram, truth table, and Boolean equations that model the solution to this problem. Design a circuit that indicates the following:

- When all three dressing rooms are empty
- When only one or only two dressing rooms are occupied
- When two or three dressing rooms are occupied

**10)** Your four friends are total whack jobs so you’ve decided to design a circuit that will help you decide how you will spend time with them. Provide a block diagram, truth table, Boolean equation and circuit diagram that models a solution for this problem. Be sure to state any assumptions you make for this problem. Design a circuit that specifies when it is safe to go out with your friends according to the following criteria:

- You’re a total whack job too, so you must go out with at least two friends
- At no time will all four of your friends want to go out together
- Friend A will only go out if Friend B goes out too

**11)** 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, an input value of “0000” will never occur (be sure to note this fact where appropriate). Provide a block diagram, truth table, and a Boolean equation that models a solution for this problem.

**12)** Design a circuit with an output that indicates when the 4-bit unsigned binary input is greater than two and less than twelve (2 < input_val < 12). For this problem, the binary equivalent of 15 will never appear on the circuit inputs. Provide a block diagram, truth table, and Boolean equation for the final circuit.

**13)** Design a circuit that has inputs consisting of a single switch and a 3-bit unsigned binary number. If the switch is off (off = ‘0’), the output indicates when the 3-bit binary input is less than four. If the switch is off (on = ‘1’), the output indicates when the 3-bit binary input is less than three. The value of “000” will never appear when the switch is in the off position. Provide a block diagram, truth table, and a Boolean equation for the final circuit.

**14)** Design a circuit that has four inputs and one output. The output is used to indicate the following conditions regarding four people (Person A, B, C, and D) in a room. Provide a block diagram, truth table, and a Boolean equation that models a solution for this problem.

- When person A is in the room and at least two other people are in the room, and
- When person B is in the room and only one other person is in the room.

# 7 Timing Diagram Introduction

## 7.1 Introduction

The previous chapters provided a foundation of digital design. Half the battle in implementing of any design is the notion that your design will need modifications in order to ensure the design successfully completes the task it set out to do. This 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 timing diagrams, a mechanism to facilitate both of these objectives. Timing diagrams are going to help limit the number of mistakes you make and help you and/or anyone understand your design.

Timing diagrams represent both a design tool and a test tool, which means that you can use timing diagrams to both specify designs and test designs. Timing diagrams provide a visual representation of what the various signals in your circuit should be doing (design) or what your circuit is actually doing (test). Whoever who coined the phrase "a picture is worth a thousand words" was definitely referring to timing diagrams.

**Main Chapter Topics**

**TIMING DIAGRAMS:** Digital designers use timing diagrams in order to specify, explain, and/or model digital circuits. Timing diagrams provide both a design tool as well as a method to verify the proper operation of circuits. This chapter introduces timing diagrams and describes their relation to digital circuits.

**Chapter Acquired Skills**

- Be able to understand the terminology and symbology associated with timing diagrams.
- Be able to use and interpret different timing diagram styles
- Be able to use timing diagram to specify functional relationships in digital circuits
- Be able to analyze timing diagrams to generate Boolean equations describing digital circuits.

## 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 100% accurate descriptions, they are "timeless" in nature. This "timelessness" forms somewhat of an artificial representation of a circuit because digital circuits operate over given periods of time[1]. As digital circuits become more complex, it becomes harder to imagine how exactly 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. We generally expect the circuit's inputs to change; when these changes occur, the

[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.

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.

Timing diagrams detail a digital circuit's operation over an arbitrary time span. Because of this, timing diagrams are important in digital design for two main reasons. Firstly, timing diagrams are able to specify and/or model digital circuit operation[3]. Secondly, digital designers use timing diagrams to verify that digital circuits are operating as specified either by using some type of simulator or by examining the waveform output from the actual circuit. In a written text such as this one, we only deal with the first item. When you're designing and implementing circuits, you'll be living with the second item when you work with simulators.

We typically use special terminology and symbology in timing diagrams; we go over 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 overly complicated.

### 7.2.1 Timing Diagrams: The Gory Details

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 numbered notes below Figure 7.1 provide an extended description and comments regarding each of the timing diagrams in Figure 7.1. The horizontal axis is the time axis in each of these timing diagrams; we only use the term "time" but we don't include metrics such as "seconds" or "milliseconds". The timing diagram shows a "functional" relationship; at any given time, a given signal is either high or low, but never both at the same time.

1) This timing diagram shows a line that represents the value of digital signal in question. The signal typically has 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.
2) This timing diagram explicitly shows the two values of the signals. The vertical axis lists these two values as 'H' and 'L', which represent the high and low values of signals, respectively. This timing diagram also includes horizontal dotted lines, which support the notion that the digital signal is either high or low[4]. Timing diagrams often omit these dotted lines; we often include them in "busy" timing diagrams in order to increase readability.
3) This timing diagram is similar to the timing diagram of (b) but we replace the 'H' and 'L' 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.
4) This is another common style of modeling digital signals. While the previous timing diagrams use vertical lines to represent signal transitions, this timing diagram uses slanted lines. The lines always slant 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 previous timing diagrams. In other words, if you look close enough[5], every signal appears slanted.
5) The final timing diagram is nothing new, but 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 low state to a high state, or the signal *toggles*. At (c), the signal switches from a high state to a

[3] More often, digital designers only specify the important parts of the circuit. In this context, "important" could have many meanings. As we travel deeper into digital design land, these meanings start to surface.

[4] 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 obfuscate due to brain overload.

[5] This means if you lower the time scale to smaller and smaller values.

low state, toggles. 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.

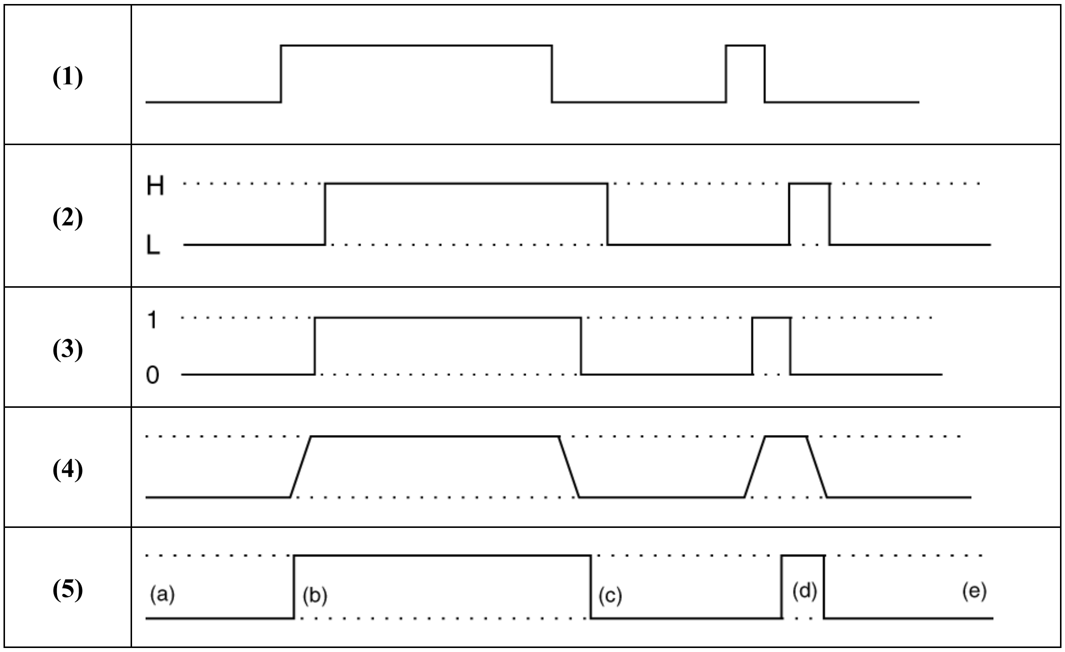


**Figure 7.1: Example timing diagrams.**

### 7.2.2 Timing Diagrams: The Initial Details

We use timing diagrams to model the operation of digital circuits. Figure 7.2 shows an inverter and an associated timing diagram. The signal names **x** and **F** represent the input and output to the inverter, respectively (the top of Figure 7.2). The upper signal in the timing diagram is labeled **x**; the timing diagram shows the **x** signal as a function of time. The signal activity 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**[6]. Figure 7.2 shows the complementary relationship between the input and output for the inverter.

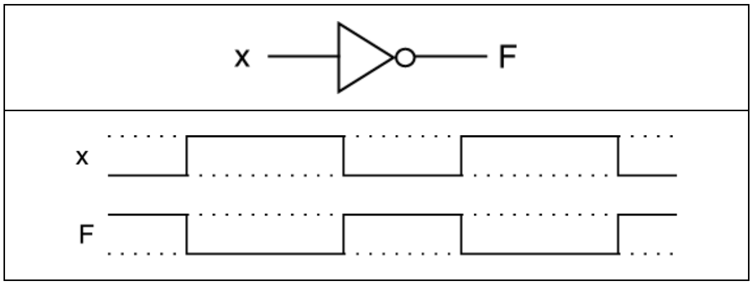
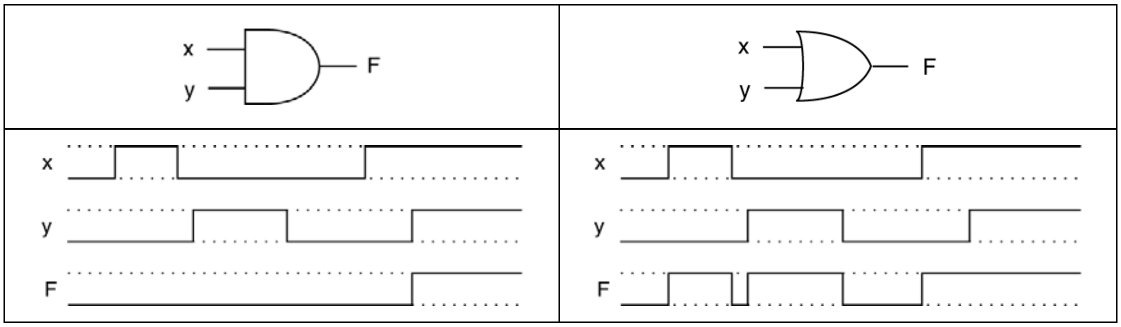


**Figure 7.2: Example timing diagram for inverter.**

Figure 7.3 shows example timing diagrams for AND & OR gates. 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 timing diagrams in Figure 7.3 completely describe the

[6] Keep in mind that timing diagrams show the true functional relationship between the input (the independent variable) and the output (the dependent variable). For any one given instance of time, the output is necessarily high or low, but never both.

operation of AND & OR gates by showing the same information as the truth table but in a different form. Keep in mind that for both timing diagrams in Figure 7.3, the value of the input variables is arbitrary.



**(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. The timing diagram includes the three inputs and one output in the truth table.

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 dotted line, 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 input 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 design problem.**

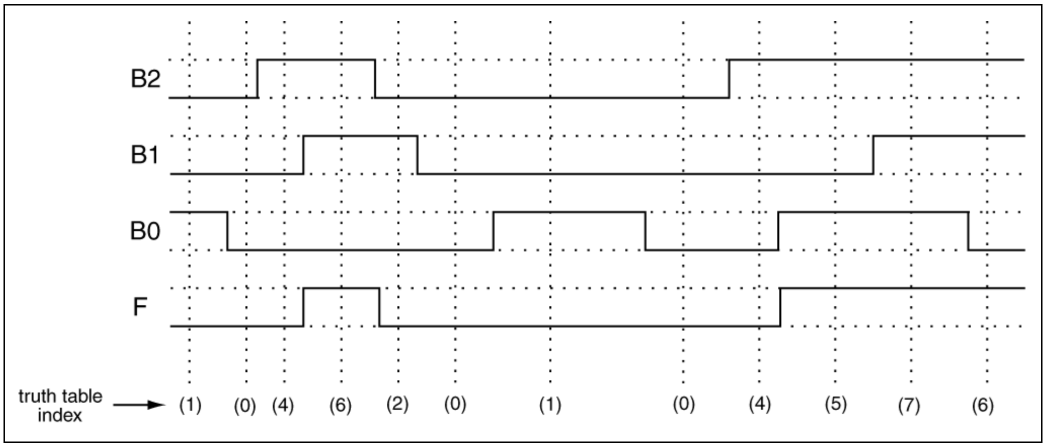


**Figure 7.5: Timing diagram for main problem specified in this chapter.**

Here are a few more comments regarding timing diagram of Figure 7.5.

- The vertical dotted lines in Figure 7.5 do not overlap any input signal transitions. The “vertical” transitions in the signals indicate a discontinuity[7] in the signal.
- The input signals **B0**, **B1**, and **B2** are arbitrary. In this particular timing diagram, one of the eight possible input combinations is missing from the timing diagram. Therefore, the timing diagram in Figure 7.5 does not completely describe a function as it would if it had output for every possible combination of inputs.

## 7.3 Timing Diagrams: Bundle Notation

Every digital designer knows that the underlying goal is to transform things from one form, to an equivalent but simpler form. This is particularly true with timing diagrams because they tend 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 group. The resulting grouping of signals makes designs easier to understand; the associated timing diagram is also easier to analyze.

In digital design, the term “bus” sometimes refers to a group of signals, but the term bus has multiple definitions[8]. The more appropriate term for what we’re describing here is a “bundle”. You need to get used to the terms “bundle notation” and “bus notation” as digital design uses these terms quite often.

### 7.3.1 Bundle Notation in Schematic Diagrams

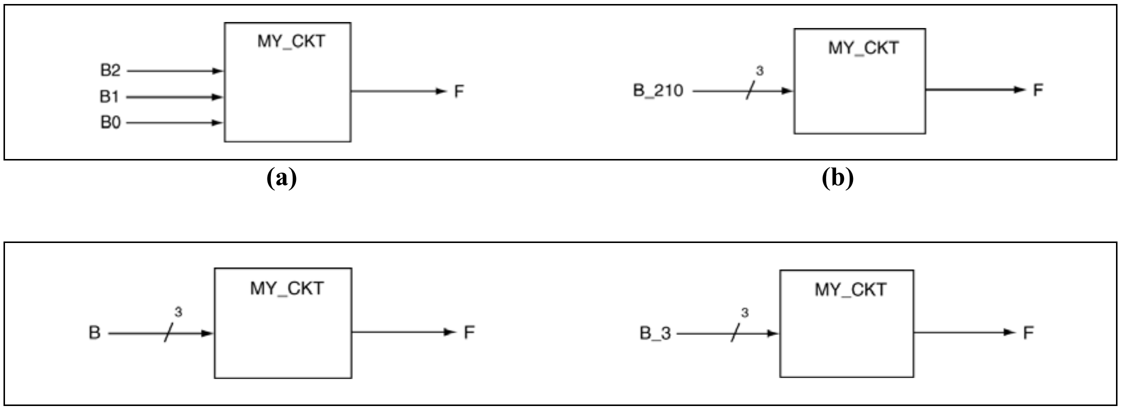
We can simplify block diagrams by “bundling” signals; using *slash notation*” allows us to do this quite easily; Figure 7.6 shows a few examples. We use a forward slash indicates a bundled signal and a number to indicate the number of signals in the bundle. Figure 7.6(a) shows the original diagram while 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. In each of the subsequent bundles, some information is lost (the names of the individual signals) in an effort to simplify the diagram less busy.

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

[8] 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.

- Figure 7.6(b) shows one approach to bundling. This diagram attempts to preserve the names of the underlying signals from Figure 7.6(a). The slash on the "B_210" line indicates that the B_210 signal is now a bundle and that it contains three signals as the tiny "3" near the slash mark indicates.
- 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 clear what the "_3" is attempting to indicate. As a result, it is questionable how much the "_3" helps.



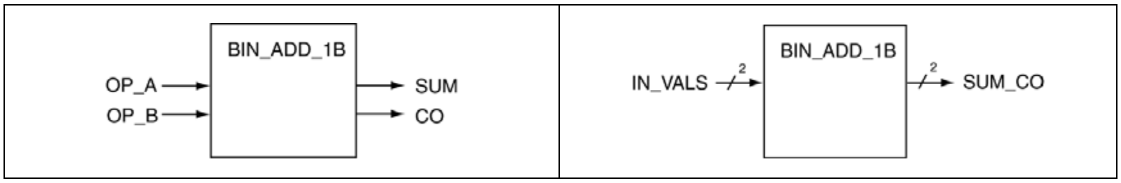
(a) (b)

(c) (d)

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

The general idea is to use bundling to make diagrams more readable. However, you need to be careful, 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 one-bit adder circuit[9] 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 is distinct; thus placing them into a bundle has made the diagram more confusing. We know this circuit is a 1-bit adder, but from the Figure 7.7(b) it appears to be some flavor of two bits. The idea behind bundling is to make the resulting diagrams more readable to humans; the example in Figure 7.7 has failed in this mission. Always make sure whatever you're doing makes things easier to read and understand; "looking better" does not necessarily support "being better" is all about making things more understandable.



(a) (b)

**Figure 7.7: A1-bit adder BBD (a), and a bad attempt to simplify (a) by using bundle notation (b).**

[9] This circuit adds two one-bit values and outputs a sum and a carry-out.

### 7.3.2 Bundle Notation in Timing Diagrams

There are many ways to model bundles in timing diagrams; this section shows a few of them. Figure 7.8(a) shows that same tired block diagram we've been using for way too long now. What we're interested in is a timing diagram associated with Figure 7.8(b). The block diagram in Figure 7.8(b) represents an equivalent version of Figure 7.8(a), which we simplify using bundle notation; the 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 are now different, 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 in the timing diagram.

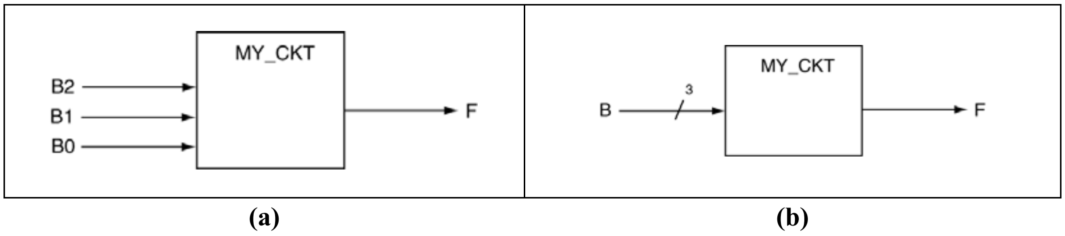
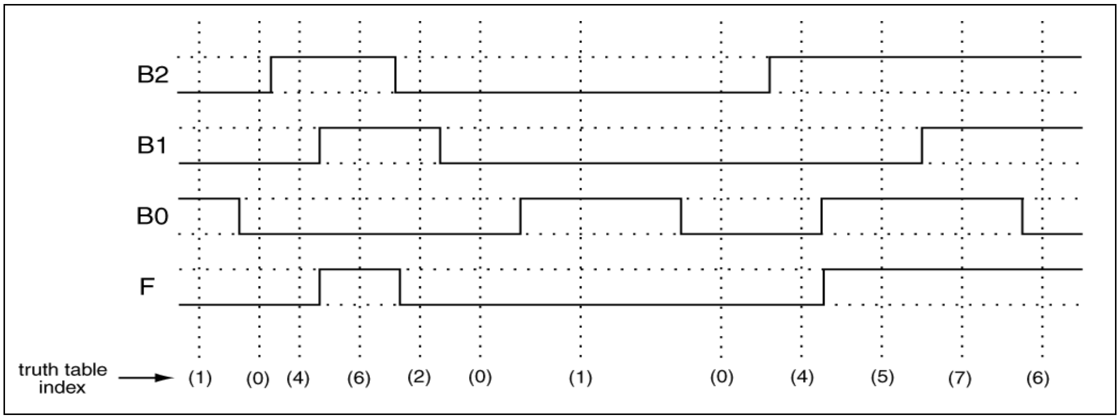
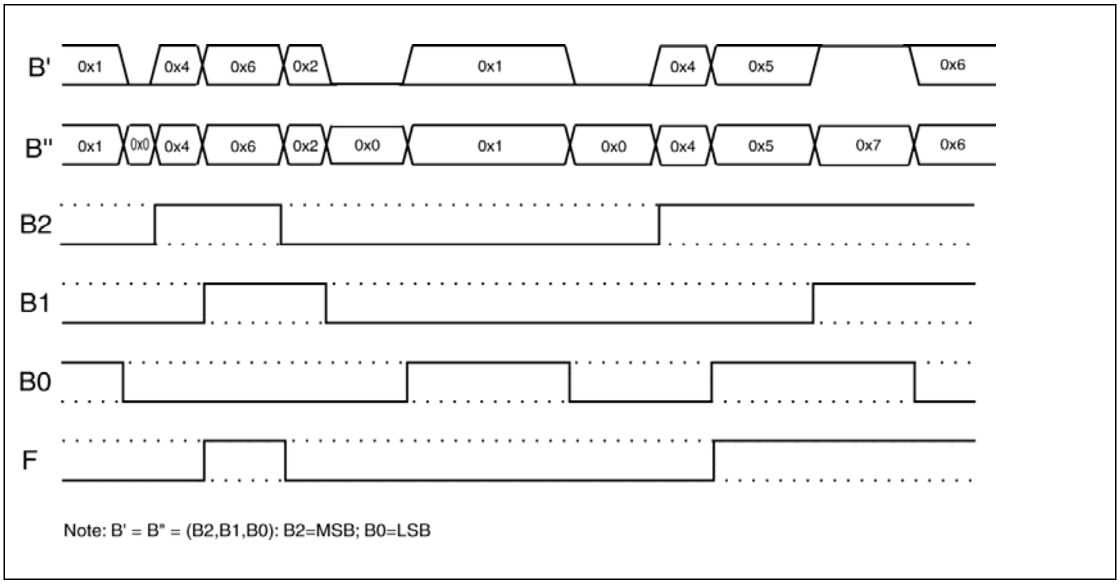


**Figure 7.8: Example block diagrams for use by Figure 7.9.**

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'
0x1 0x4 0x6 0x2 0x1 0x4 0x5 0x6
B"
0x1 0x0 0x4 0x6 0x2 0x0 0x1 0x0 0x4 0x5 0x7 0x6
B2
B1
B0
F
Note: B' = B" = (B2,B1,B0): B2=MSB; B0=LSB

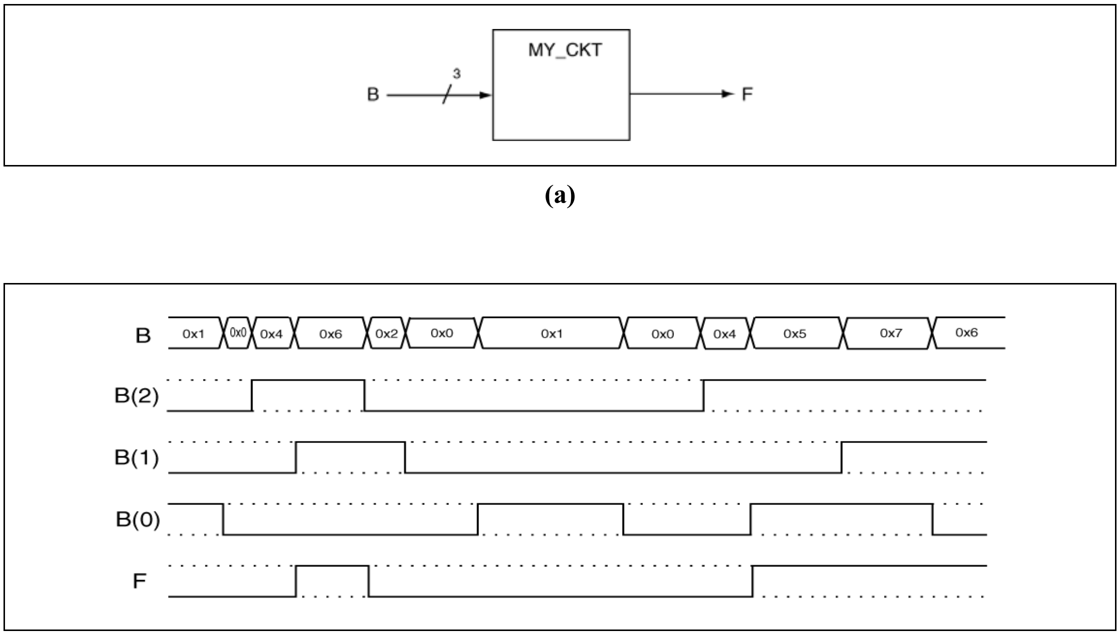
**(b)**

**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 "X's" 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 indicate the value of the signals in the bundle. You'll see many different ways of representing these numbers; we opt to use a C programming language-type notation used to represent hexadecimal numbers to indicate the individual signals in the bundle. Specifically, the "0x" prefix on a number indicates that you should interpret the number as a hexadecimal number.
- There are only three signals associated with the bundle while hex notation can specify four bits per hex number. We assume the missing signal(s) is always the most significant bit or bits; we also assume these missing bits are zero.

- The diagram should explicitly state that in the hex number n Figure 7.9(b), the most significant bit represented is **B2** while the least significant bit is **B0**. If you did not state this, the reader may make an incorrect assumption regarding your timing diagram.
- Figure 7.9(b) show the **B**' and **B**" signals. These are equivalent signals but we use two different styles to represent their values. Often times the timing diagram drops the "parallel bar" notation when all the signals in the bundle are all high (all 1's) or all low (all 0's). Ether approach is fine; the timing diagram in **B**" is clearer and more consistent (one man's opinion).

Another common 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 **B** signal. The diagram indirectly states that bundle **B** comprises of three signals (**B(2)**, **B(1)**, and **B(0)**, with **B(2)** being the MSB and **B(0)** being the LSB[10]. Simulators typically use this notation.



**(a)**

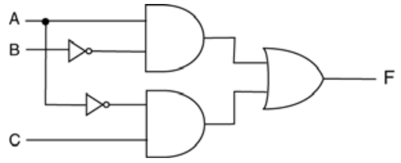
**(b)**

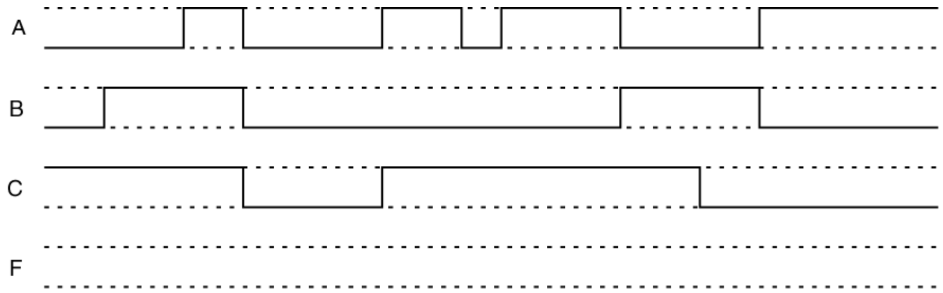
**Figure 7.10: Bundle expansion showing parenthetical indexing on the expanded bundle.**

[10] This notation assumes that the signal with the highest index is the most significant bit. This notation is quite common and diagrams rarely state 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.

**Example 7.1: Timing Diagram Based on Circuit**

Use the following circuit to complete the accompanying timing diagram.





**Solution**: There are many ways to approach this problem; the approach we take 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. When you gain more experience in digital design, you'll see other ways to solve the problem.

Step 1) Write out the Boolean equation implemented by the circuit in a form we recognize. While we're at it, we may as well expand the equation into standard SOP form, which helps us complete the truth table. We do this by multiplying each product term by something that ensures each product term includes one instance of the each independent variable. Multiplying a term by a variable ORed with its complement does not change the product term because we are multiplying the term by '1' (a Boolean algebra theorem). Figure 7.11 shows the result of this step.

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

**Figure 7.11: Expanding the original equation.**

Step 2) Generate a truth table and fill in a '1' for the output associated with each of the product terms. We include the index values here as it may help us out later. One important point in this problem is that the problem never stated which of in the inputs the most significant bit. In this problem, not stating this information does not change the answer. However, since we decided to list the problem using a numeric index, we must state that input **A** is the MSB while input **C** is the LSB. Figure 7.12 shows the result of this step.

| 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 |

**Figure 7.12: Completing the associated truth table.**

Step 3) Figure 7.13 shows that you can use the state of the inputs signals to generate numeric indexes on the original timing diagram. The timing diagram includes vertical dotted lines for every notable span of time on the timing diagram.

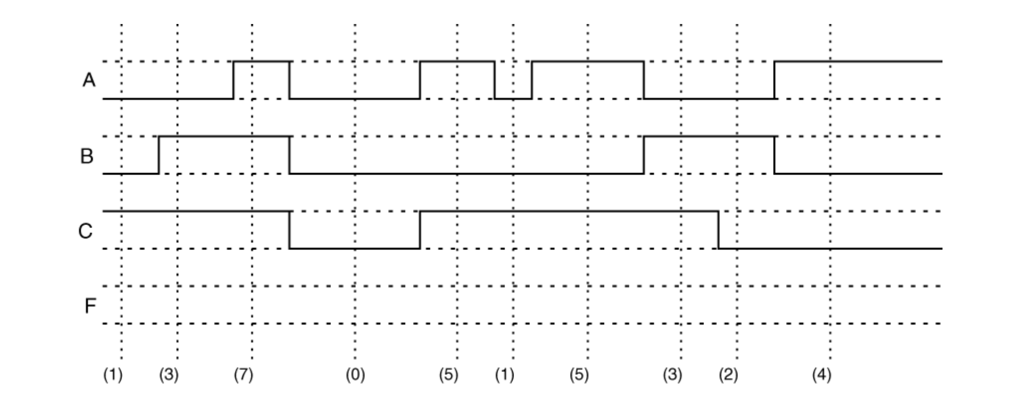


**Figure 7.13: Entering the truth table inputs to a timing diagram.**

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. Figure 7.14 shows the timing diagram representing the final solution for this example.

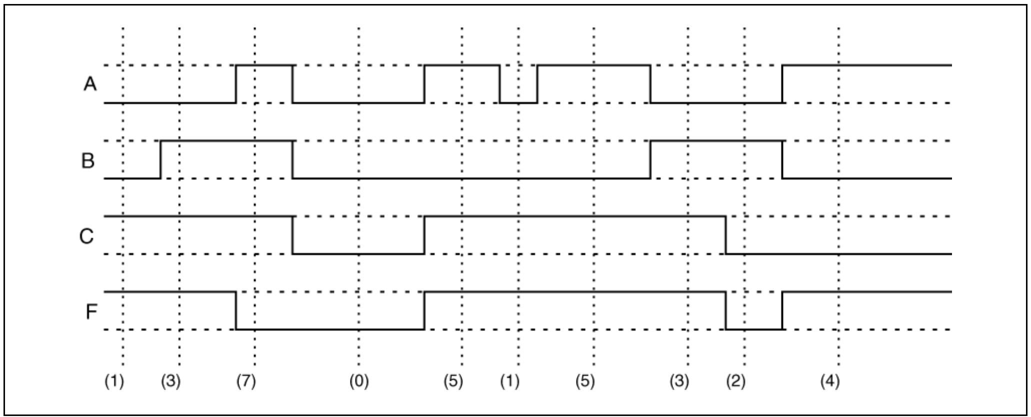
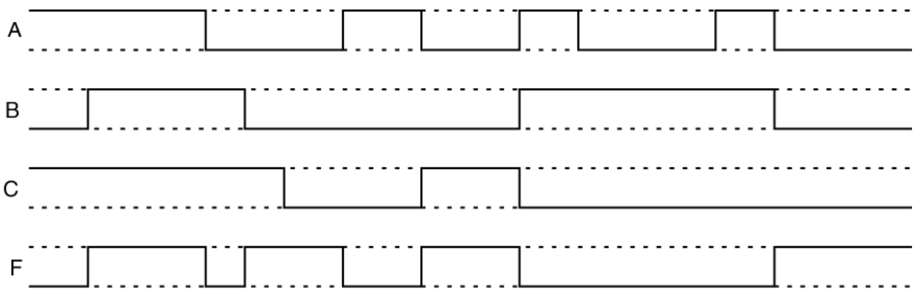


**Figure 7.14: The completed timing diagram for Example 4.8.**

**Example 7.2: Timing Diagram Modeling a Circuit**

If possible, use the timing diagram listed below to generate a Boolean equation 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 describes a function. For the timing diagram to describe a function, the timing diagram must possess two characteristics. 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.

Step 1) Find and mark all the input combinations represented in the given timing diagram. Figure 7.15 shows the result of this step.

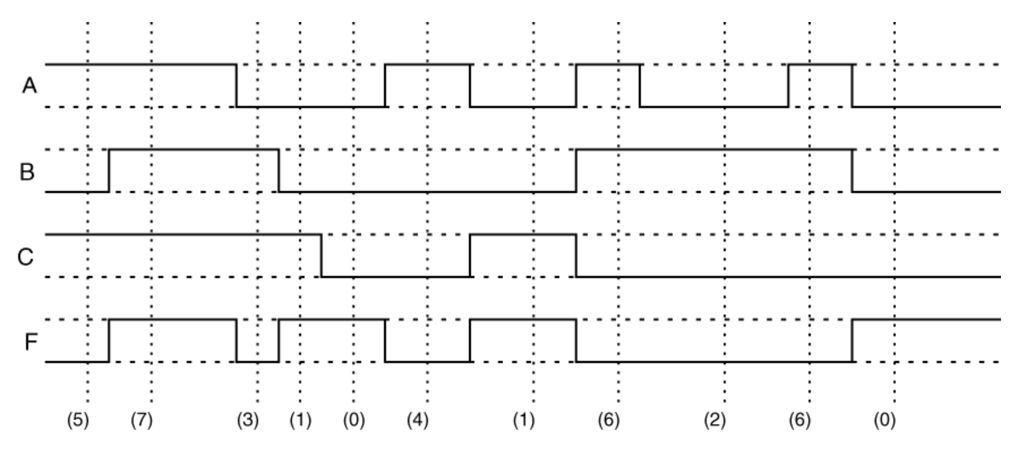


**Figure 7.15: Inserting useful annotations into the timing diagram.**

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. Figure 7.16 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 |

**Figure 7.16: Completing a truth table for the problem.**

Step 3) From the previous truth table, we can generate the following Boolean equations. Figure 7.17 shows the equation that solves this problem.

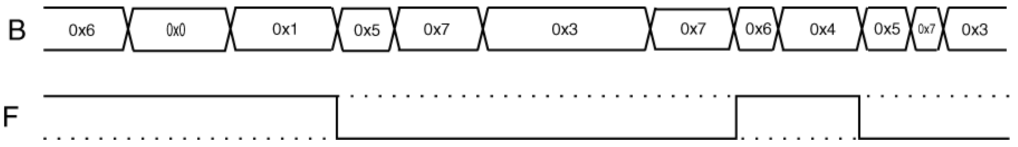
$$F = \overline{A}\ \overline{B}\ \overline{C} + \overline{A}\,\overline{B}\,C + ABC$$

**Figure 7.17: The final equation for Example 7.2.**

Post Problem Commentary: This problem could be categorized as an "analysis" problem, or maybe even better as a "timing analysis" problem. We "analyzed" the original timing diagram in order to arrive at our solution. In addition, if we could also draw the circuit associated with the final equation.

**Example 7.3: Timing Diagram with Bundle Notation**

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 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**". Figure 7.18 shows the final solution. Are you ready for the final solution[11]?

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

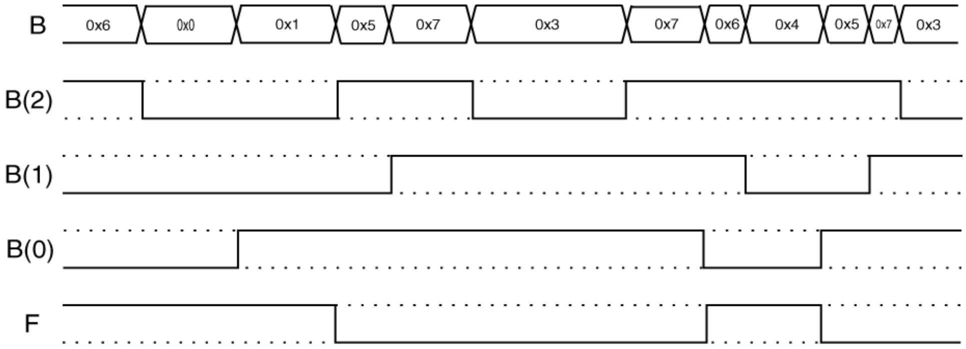


**Figure 7.18: The final solution for Example 7.3.**

## 7.4 Timing Diagram Annotations

Including notes on timing diagram is something you should always do. Digital designers refer to this practice as “annotating” their timing diagrams. Nothing looks crappier than a page filled with timing diagrams that include no notes helping the reader extract pertinent information from the timing diagram. The unstated rule for all timing diagrams is that they should include notes to describe what is important in that specific timing diagram in order to draw the reader’s eye to those items. Stated differently, either timing diagrams are trying to show you something, or you use them to show something to other people. 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 your annotations make the timing diagram clearer, 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.
- 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 signals in timing diagrams.
- 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 diagrams, for that matter) should include a title that quickly describes what the timing diagram is trying to show.

We tout timing diagrams as being incredibly useful, but that usefulness has two constraints. First, timing diagrams are only useful if you understand what you’re looking at. Second, the only way you can understand what you’re looking at is by having a working knowledge of the underlying circuit. If you don’t meet these two constraints, timing diagrams look like a bunch of random squiggles.

When you’re looking at a timing diagram, chances are good that it came from one of three sources. Here are those sources with some brief explanation.

**Logic Analyzers:** A logic analyzer is a device that shows the output of a circuit over a course of time. The key here is that the circuit is implemented and you’re testing an actual device. Logic analyzers output plain timing diagrams associated with the signals they are monitoring. What you get in the end is a plain timing diagram. While the timing diagram is great in itself, you must have an idea of what you're looking at and understand the circuit that generated the timing diagram for make the timing diagram useful.

**Simulators**: Simulators typically outputs timing diagrams. Simulators generate their timing diagrams based on a model of your circuit, which means you don't necessarily need to implement your circuit before simulating your circuit. Additionally, while a logic analyzer output can only display signals in the timing diagram that you've physically connected a test lead to, the simulator can generally provide an output of any signal in the circuit. Once again, the timing diagram is great only if you have an idea of what you're look at and understand the underlying circuit.

**Humans:** Yes, humans can generate timing diagrams too. The problem here is that they are timing consuming to generate, particularly if you plan to make them legible[12]. Yes, humans have their issues, but unlike simulators or logic analyzers, humans have the ability to make their timing diagrams more understandable. Humans increase the understandability and usefulness of their timing diagrams by annotating them, logic analyzers and simulators don't have this ability.

There is a common problem with courses that require the generation of timing diagrams. People tend to use a tool to generate the timing diagram as requested by the assignment, and then simply submit that timing diagram with the assignment deliverables. In this case, there is no evidence that people know what they submitted. The solution is to fully annotate any timing diagram you submit; the assignment is meaningless otherwise.

### 7.4.1 Timing Diagram Usage

No matter what you find yourself doing in digital design, you'll be working with timing diagrams. Digital design uses timing diagrams for three main purposes:

**Design Description**: If someone wants you to implement a circuit, they may provide you with a timing diagram that models the desired operation of a circuit. There are other ways to model circuits, but in some instances, the timing diagram provides the best model[13]. In this case, the timing diagram specifies how the circuit should operate.

**Design Verification**: Once the circuit has been modeled and/or implemented, we can simulate it or connect it to the logic analyzer. We use the timing diagram output from these devices to determine if the design is working as we expect it to. The simulator tells you if your design has a good chance of working if you were to implement it, while the logic analyzer tells you if design is working after you implement the circuit.

**Design Documentation**: Once you establish that the circuit actually works, you should use a timing diagram to document the circuit's operation. It's not enough to print out the timing diagram; you must also provide annotations on the timing diagram to make it meaningful[14]. Additionally, if you're submitting the timing diagram as part of a report, it requires annotations to make it meaningful.

### 7.4.2 Understanding Timing Diagrams

Timing diagrams present a ton of information regarding the operation of your circuit. The key to quickly understanding timing diagrams is to annotate them. This generally means annotation by hand, even if a simulator or logic analyzer generated that timing diagram. You created the timing diagram for a reason; you won't obtain your objective if the person reading your timing diagram does not understand what they're looking at, which is a potential issue if that person is your instructor or supervisor.

[12] I admit it…this text is lacking in timing diagrams because they take so long to generate.

[13] Recall that good models transfer the most information at the fastest rate.

[14] It does not matter if you're the expert on it at the moment; six months from now you'll have no idea how the circuit works.

Even the simplest timing diagram presents a lot of information very quickly. Your mission as the timing diagram annotator is two-fold: 1) to draw the reader's eyes to the portion of the timing diagram that contains the information you feel is important, and, 2) to tell the reader what it is you're trying to show (what they're looking at).

Students always ask me how to annotate timing diagrams. The truth is, I don't really know; I'm actually hoping my students give me some ideas, or at least some ideas that are better than the ideas that I currently use. The key here is to add annotations that make the timing diagram clean. I don't feel there is only one way to make timing diagrams clear, and I don't feel that the approaches I use are the best ways. However, for lack of any better direction, I present them here.

Generally speaking, timing diagrams are attempting to show you something. You'll initially spend most of your time designing your annotations to show one of the following three items:

1) **Temporality of Events**: This is a fancy way of stating the time required for "something", which is generally bounded on both ends. Examples include the duration of a pulse or the duration of a propagation delay.
2) **Causality of Events**: Events in timing diagrams typically are value changes for a given signal or set of signals. The annotations the condition(s) that caused that particular event.
3) **Correctness of Operations**: Digital circuits typically perform logic and mathematical operations. In this case, we want to show that a given operation is correct. This is slightly different from causality of an event in that we show the circuit performed the operation as it should have.

Figure 7.19 shows an example of a timing diagram that indicates the temporality of events. These annotations show various time durations associated with events such as the rising and falling edges of signals. Add whatever notes you feel necessary to tell the reader the significance of events in the timing diagram.

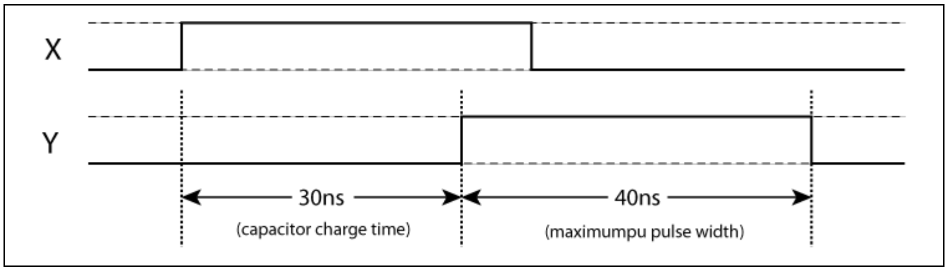


**Figure 7.19: Examples of annotations showing the temporality of events.**

Figure 7.20 shows an example timing diagram annotating the causality of events. This diagram is from an up/down counter (a circuit that counts up or down); the diagram is showing the required control signals to either allow an up or down count. The event of interest is always what the arrow is pointing at; the other annotations are the conditions that allowed the event to occur. Here are a few notes to support the circled values in the diagram.

1) The event of interest is what the arrow is pointing at. The conditions causing this event are the rising edge of the **CLK** signal (because the arrow emanates from that edge) and the fact that the **UP** signal is low (because we put a heavy dot there). Note the **CNT** value decrements.
2) The conditions causing this event are the rising edge of the **CLK** signal (because the arrow emanates from that edge) and the fact that the **UP** signal is high (because we put a heavy dot there). Note the **CNT** value increments.

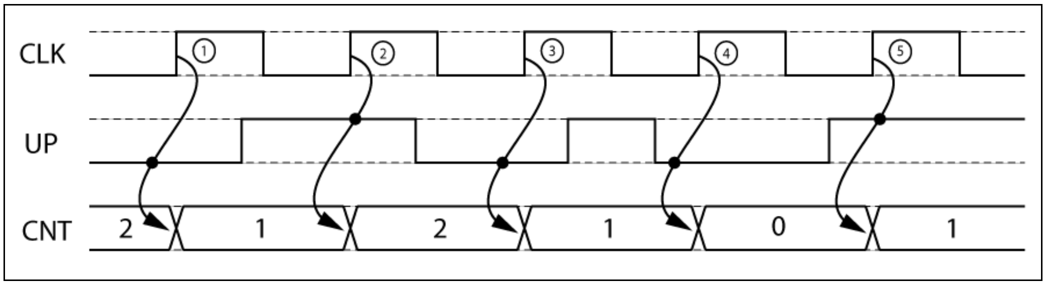


**Figure 7.20: Examples of annotations showing the causality of events.**

Figure 7.21 shows annotation indicating correctness of operations. This timing diagram is from some type of adder circuit, which adds the values of **A** & **B**. The timing diagram shows the two values being added and the resulting sum. The timing diagram shows that the values being added generate the correct results. Here are a few other things of interest to note about the timing diagram.

- The diagram uses two different types of annotating styles. The first style uses arrows that point to two versions of the equation. The second form uses fewer arrows and equations. You could argue that the first version is better because it has more information, but it also introduces more clutter to the timing diagram.
- We did not bother to annotate every line in the diagram. We stopped at some point for a reason such as to save time as we felt the given annotations did the job for us.
- This timing diagram could have also shown the causality of events, but we opted to omit that style of annotation.

These styles of annotations are guidelines; they're not carved in stone. If you feel you have a better approach to annotating, then you should do it. Remember that you're trying to transfer information; you know you've done a good job if your annotations quickly and easily transfer that information to the reader of the timing diagram.

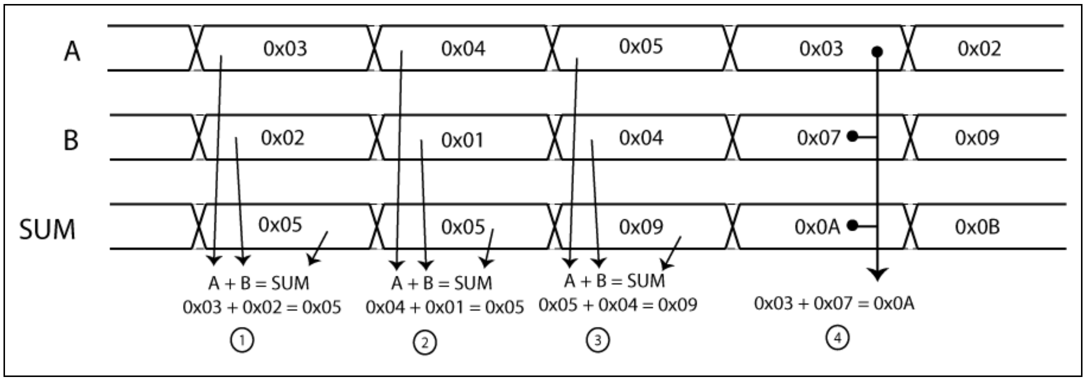


**Figure 7.21: Inserting useful annotations into the timing diagram.**

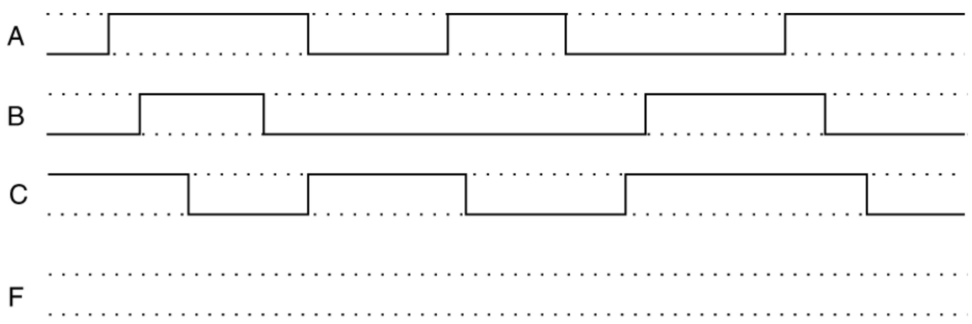
## 7.5 Chapter Summary

- 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 completely specify a digital circuit's correct operation.
- Timing Diagrams for Design: We often use timing diagrams 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: We often use timing diagrams 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 trying to figure out is whether your circuit may 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 simplify 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.
- Timing diagrams generally have three main purposes:
  1) Design Description
  2) Design Verification
  3) Design Documentation
- We generally use timing diagrams to show three types of information
  1) Temporality of Events
  2) Causality of Events
  3) Correctness of Operations

## 7.6 Chapter Exercises

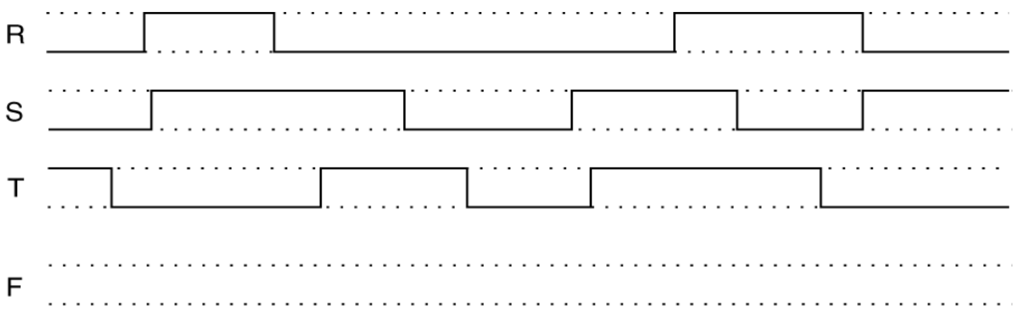
**1)** Using the following Boolean equation to complete the accompanying timing diagram.

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

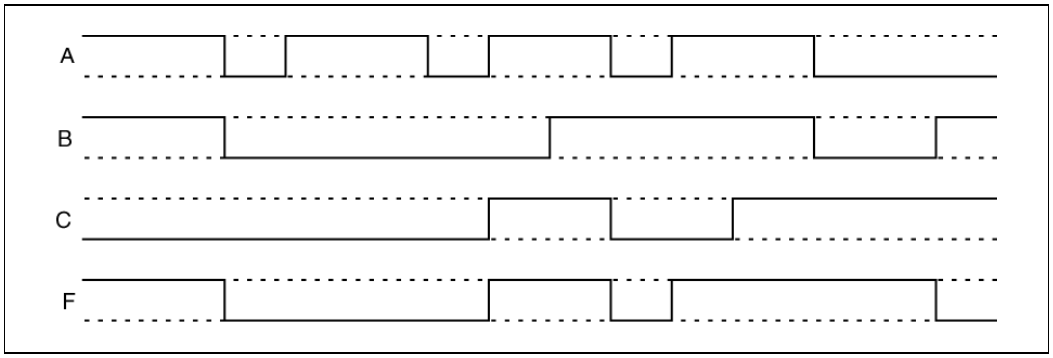


**2)** Using the following Boolean equation to complete the accompanying timing diagram.

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

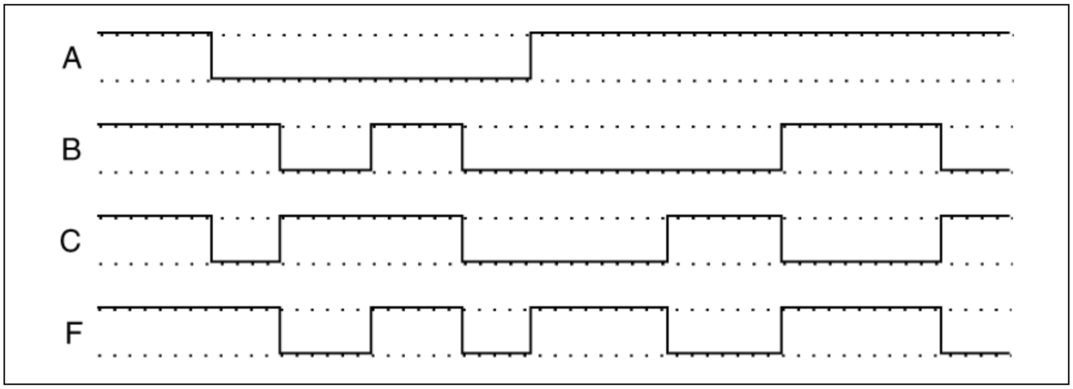


**3)** Does the timing diagram listed below completely define a function? Why or why not? If it does, write both SOP and POS equations that describes the function and provide a circuit diagram in both SOP and POS form that you could use to implement the circuit.



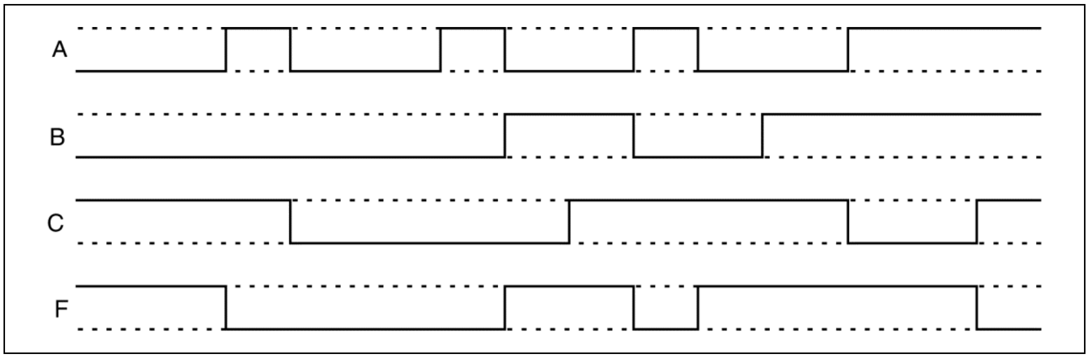
**4)** The following timing diagram may completely model a function.

- 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.



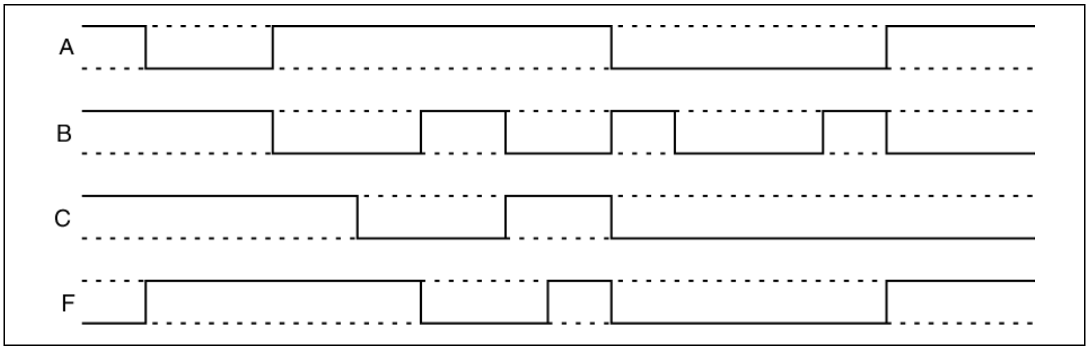
**5)** Consider the previous problem… can you safely state which of the inputs variables is the MSB or LSB? Be sure to provide a complete explanation.

**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 you could use to implement the circuit.

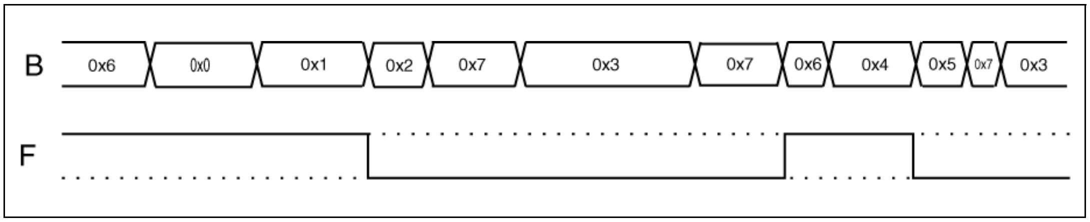


**7)** Consider the previous problem… how does the ordering of the labels of A, B, and C change the outcome of the problem? Be sure to provide a complete explanation.

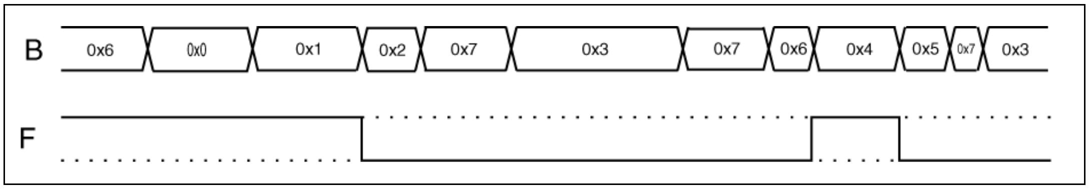
**8)** Does the timing diagram listed below completely define a function? Why or why not? If it does, write both SOP and POS equations that describes the function and provide a circuit diagram in both SOP and POS form that you could use to implement the circuit.



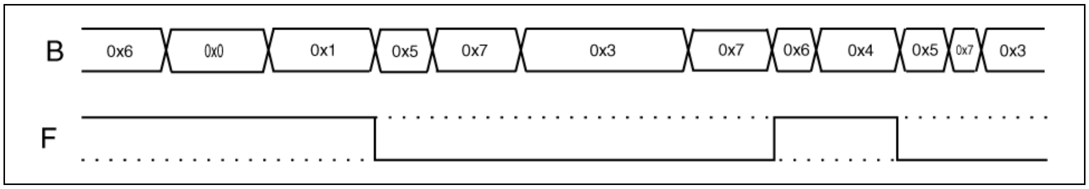
**9)** If the following timing diagram completely specifies a function, write a Boolean expression for that function.



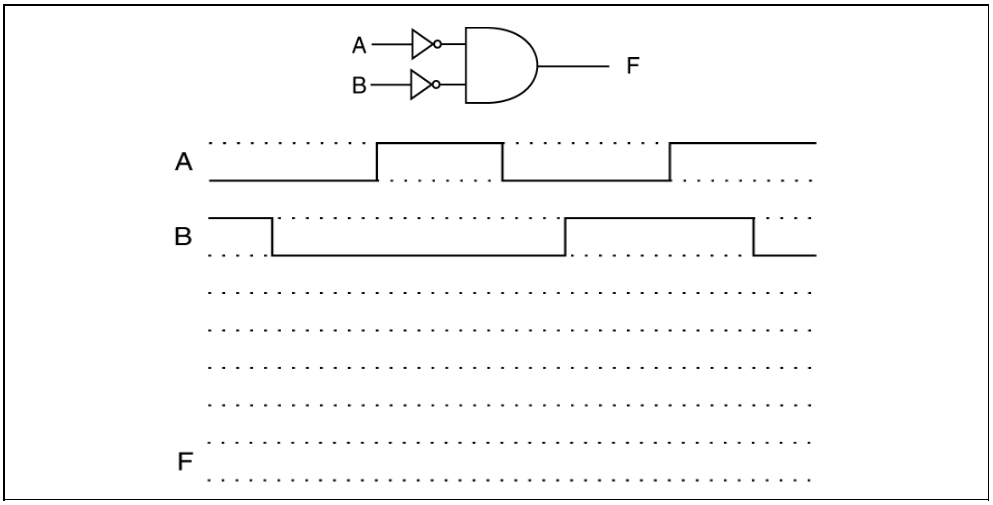
**10)** If the following timing diagram completely specifies a function, write a Boolean expression for that function.



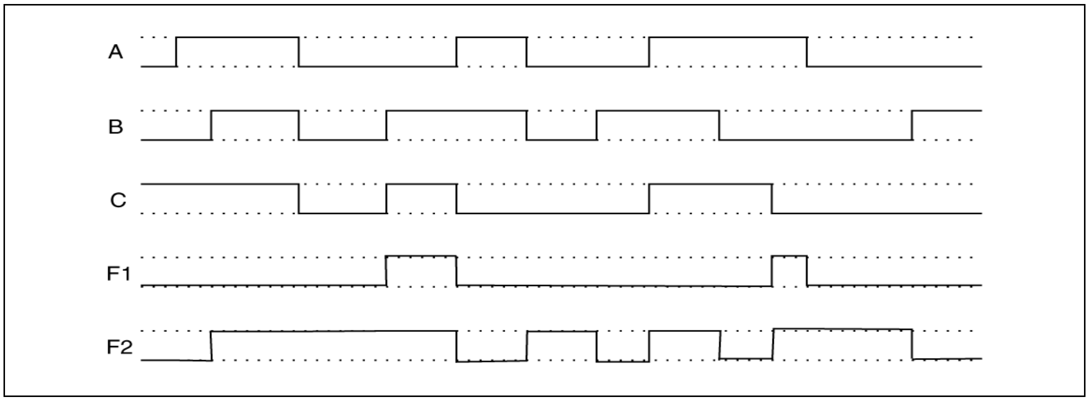
**11)** Does the following signal completely specify a Boolean function? Briefly explain why or why not.



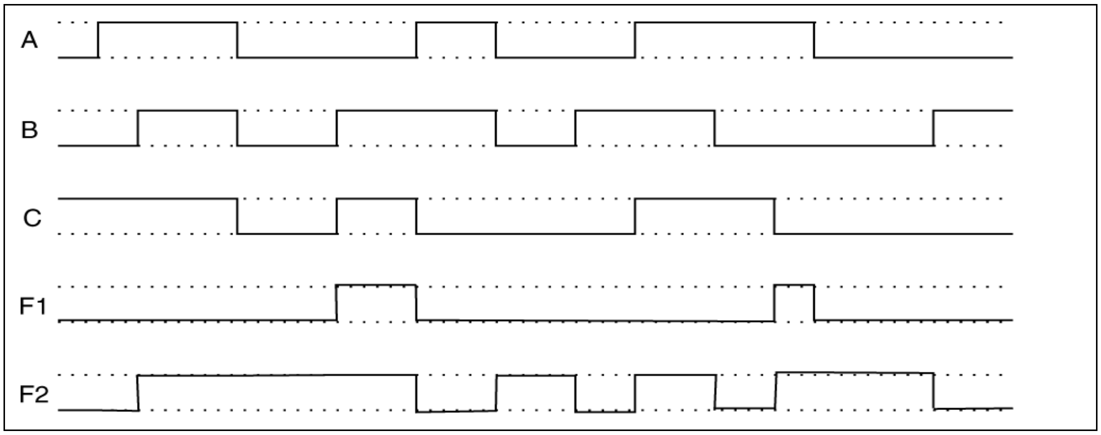
**12)** Complete the following timing diagram for the F output based on the given circuit.



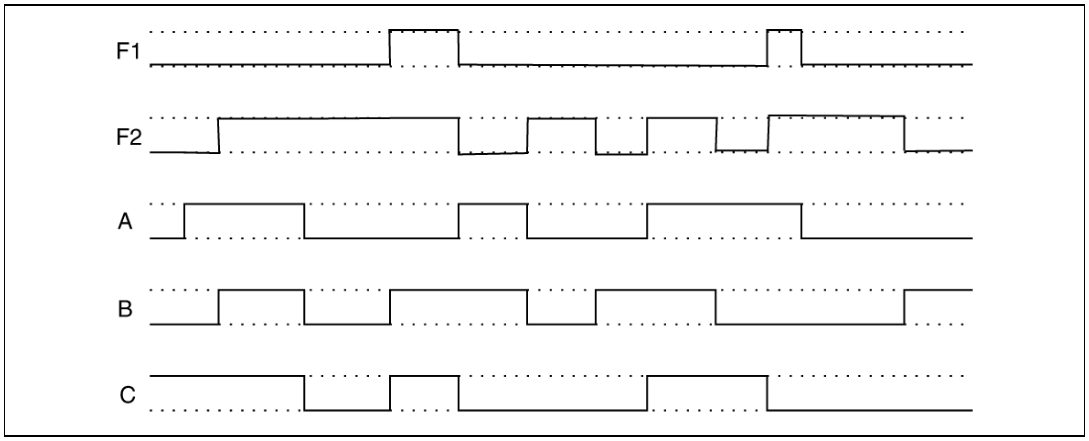
**13)** For this problem, consider the input variables to be A, B, and C and the outputs to be F1 and F2. The timing diagram below completely described functions F1 and F2. Write a Boolean expressions that describe F1 and F2



**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 expression that describe F1 and F2.

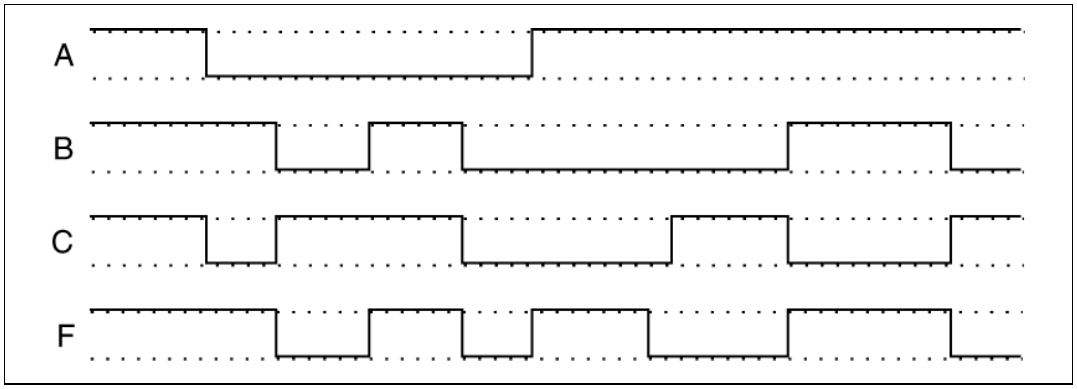


**15)** For this problem, consider the input variables to be A, B, and C and the outputs to be F1 and F2. The timing diagram below completely described functions F1 and F2. Write a Boolean expressions that describe F1 and F2.

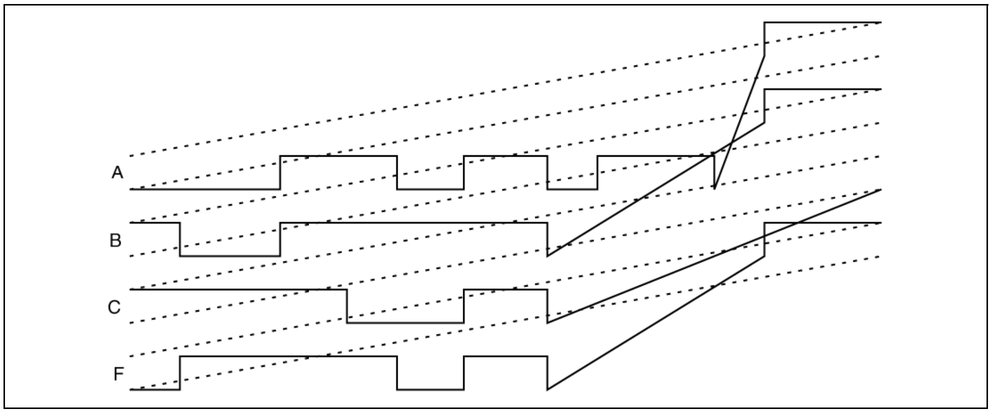


**16)** The following timing diagram may completely model a function.

- 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.

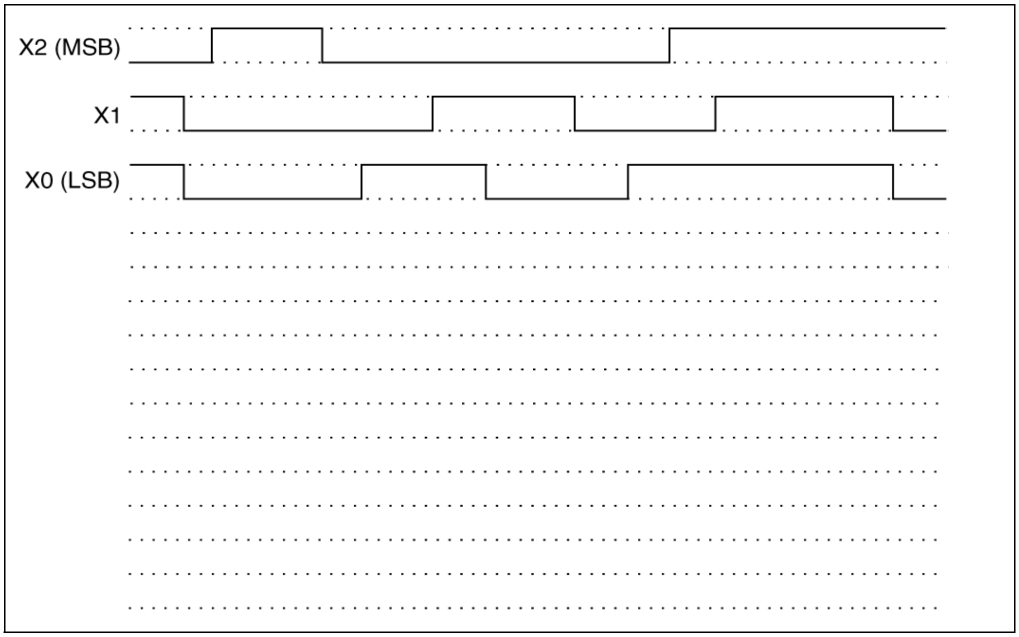


**17)** For those aspiring digital designers on drugs, state whether the timing diagram listed below completely defines a function. Why or why not? Does anyone really freaking care?

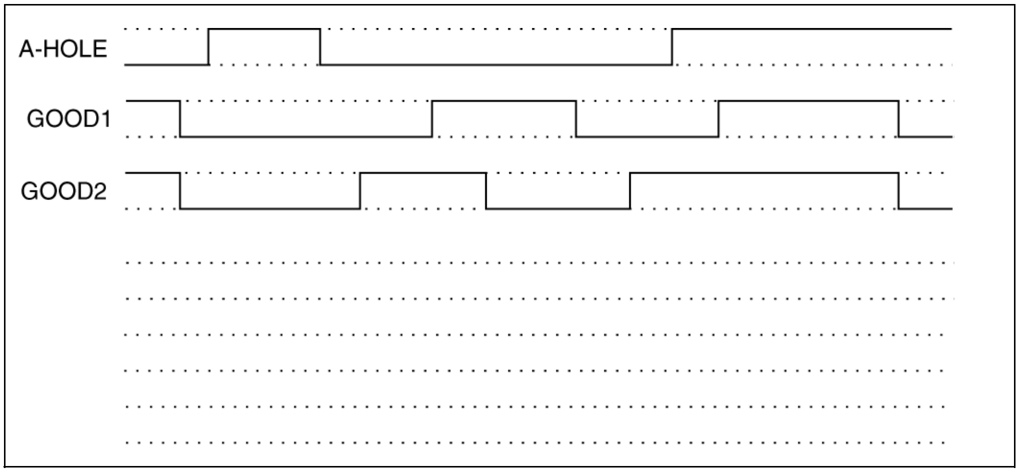


## 7.7 Design Problems

**1)** Design a circuit whose output represents a square of the input. For this problem, describe your design using SOP or POS equations. In addition, show the output of your circuit in the timing diagram 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 "KISS_ASS1" and "KISS_ASS2". Being a typical head of a committee, the chair of the committee has commissioned you to build this circuit in order to better serve him. The committee has a set of switches that they use for "secret" voting. 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.



# 8 Ripple Carry Adders

## 8.1 Introduction

There are three different approaches to performing digital design; up until now, we've only worked with one of these approaches: BFD, or *iterative design*. In an effort to increase our efficiency as digital designers, we need other design approaches. This chapter introduces our second design approach: IMD, or "iterative modular design". This approach is somewhat limited also, but it's useful in some situations. Probably the best part about IMD is that it provides a great vehicle for presenting our first digital design foundation module: the ripple carry adder (RCA).

**Main Chapter Topics**

**ITERATIVE MODULAR DESIGN (IMD):** This chapter introduces the notion of iterative modular design in the context of a standard digital circuit.

**HALF ADDERS:** One type of circuit that performs one-bit addition

**FULL ADDERS:** Another type of circuit that performs one-bit addition

**RIPPLE CARRY ADDERS:** A standard digital circuit that adds two digital values of arbitrary length.

**Chapter Acquired Skills**

- Be able to design a Half Adder (HA) and produce a gate-level model of it
- Be able to design a Full Adder (FA) and produce a gate-level model of it
- Be able to describe the differences between BFD and IMD
- Be able to design a Ripple Carry Adder (RCA) using the iterative modular design approach
- Be able to design specialty circuits using RCAs

## 8.2 Iterative Modular Design Overview

The main push behind IMD is the notion that we want to move away from the limits inherent to BFD, meaning primarily truth tables and their associated Boolean equations. IMD is the first step in decoupling the digital designer from generating Boolean equations as part of designing digital circuits.

There are two separate aspects to IMD as the name implies. The first aspect is the "modular" part of IMD; this means we use previously designed modules as part of the design. In this context, a module is a black box model of something that was previously modeled. The second aspect of IMD is the "iterative" part, which means that we do some aspect of a design repeatedly. In IMD, the thing that is going to be iterated is the modular part of IMD, or the modules. IMD involves using pre-designed modules in an iterative manner in order to create circuits that do not require Boolean equations to model. Lastly, IMD introduces hierarchical digital design.

## 8.3 The Half Adder (HA)

The HA is one of the most basic digital circuits and is the first mathematical circuit we develop in digital design. We know the HA as a “1-bit adder”, which means it adds two 1-bit values and outputs their sum and a carry out.

**Example 8.1: The 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. We refer to this circuit as a *half adder,* and it is one of the most basic circuits in digital design and the digital design world refers to it as a Half Adder, or HA. Also, state what controls the circuit.

**Solution**: 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 add two, single-digit, decimal numbers, your result are either a single digit number (less than ten) or a two-digit number (greater than nine). We represent the results of this addition that are greater than or equal to the radix with two digits while we represent the results that are less than the radix 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-digit addition. The same is true for binary addition. Table 8.1 shows the four possible results for binary addition of single bit as well as the SUM and Carry-out results.

One item of particular interest in Table 8.1 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 result is “10” which is the binary equivalent of 2 (two) in decimal[1].

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

**Table 8.1: All possible single-bit addition operations with sum and carry results.**

Step 1) **Define the Problem**: The first step is to draw a high-level BBD of the circuit. From the problem statement, this circuit contains two inputs and two outputs. Figure 8.1 shows the two inputs (arbitrarily named **OP_A** and **OB_B**)[2] and two outputs: **SUM** and **CO**. Table 8.2 and Table 8.3 show the empty truth the completed truth table for this design, respectively. The circuit has two inputs, which means it there are four ($2^2$) rows in the truth table.

[1] 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 me.

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

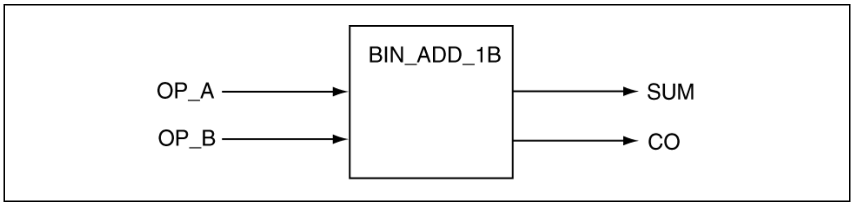


**Figure 8.1: The black-box diagram for this problem.**

| OP_A | OP_B | SUM | CO |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

**Table 8.2: The empty truth table.**

| OP_A | OP_B | SUM | CO |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

**Table 8.3: The completed truth table.**

Step 2) **Describe the Solution**: For this problem, you'll need to generate two Boolean expressions: one for the SUM and the other for the CO. We write the final equations 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 8-1: The final equations for Example 8.1.**

Step 3) **Implement the Solution**: The final step is to translate the Boolean expressions of Equation 8-1 into circuit form. Figure 8.2 shows the final gate-level implementation. This circuit has not control features.

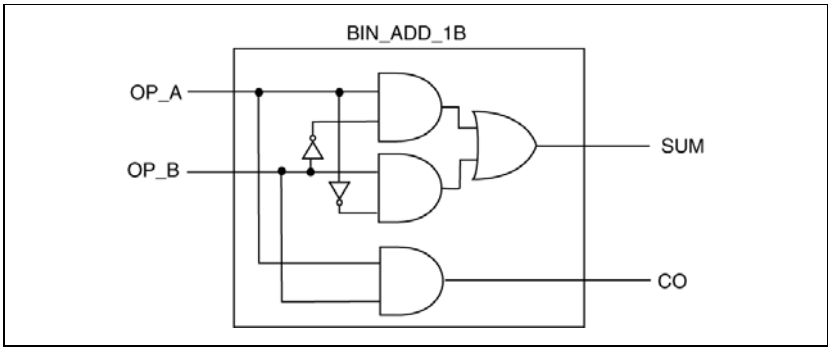


**Figure 8.2: The circuit model for the solution.**

## 8.4 The Full Adder (FA)

While adding single bits is interesting, we want to be able to add values larger than one bit. While the HA outputs both a sum and carry, the HA circuit can't do anything meaningful with the carry. HAs can never generate a result greater than one bit (or two bits if you include the carry as part of the sum). The solution is to redesign the HA so that it provides a provision for the carry from one HA as in input to another HA. The circuit that handles this is the full adder.

**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 a "carry-in" bit. In other words, this circuit completes the following operation: (**a** + **b** + **ci**) where **a** & **b** are the standard additive operands and ci represents the carry-in bit. The outputs of the circuit are identical to the half adder: **SUM** and Carry-out. Also, state what controls the circuit.

**Solution**: This design is similar to the half adder (HA), but the difference is that the FA contains an extra input, the carry-in bit. While the HA added two single bits to each other, the FA adds three single bits together. We know both the HA and FA as 1-bit adders. Figure 8.3 shows the BBD for the full adder.

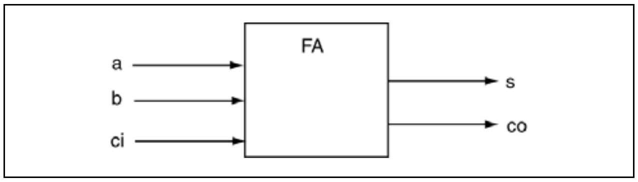


**Figure 8.3: Black box diagram of the full adder.**

The next step in the design is to specify the input/output relationship of the design, which means we must specify the outputs we want for a given set of inputs. This is the BFD design approach, so we start with a truth table that lists every combination of the input variables. Figure 8.4 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 8.4: The truth tables associated with the FA design specifications.**

The next step is to translate the information in the output columns of the truth table of Figure 8.4(b) into equation form. Equation 8.2 shows the final equations for the two output variables. From these output equations, you could easily draw the final circuit model. Finally, this circuit has no control features.

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

**Equation 8.2: Boolean equations describing sum and carry-out outputs of the FA.**

## 8.5 Ripple Carry Adders (RCA)

People get PhDs or get the big bucks for designing new circuits that perform some calculations "better" than other digital circuits. This is important because computers generally spend a significant portion of time performing mathematical operations. If you can perform math operations more efficiently or with a smaller circuit, you've saved time (so you can do more operations), and/or you've saved space (so you can include other circuitry to do more stuff), and you've probably saved power (so you can play games on your phone longer before the battery dies).

This section examines one type of mathematical circuit: the ripple carry adder (RCA). The RCA is versatile in that we can also easily configure it do subtraction (a topic for another chapter). The RCA is also a great vehicle to introduce iterative modular design (IMD).

The RCA is our first Digital Design Foundation module. The RCA is an "n-bit adder", which is a circuit that adds two n-bit numbers and provides an n-bit result and a carry-out. We construct the RCA in an iterative manner using a series 1-bit adders.

**Example 8-3: The Ripple Carry Adder**

Design a 4-bit Ripple Carry Adder (RCA). Represent each bit of this RCA using either a HA or FA. Also, state what controls the circuit.

**Solution:** First, we must specify the inputs and outputs of this design using a black box diagram (BBD). The inputs include two 4-bit values; the output includes a 4-bit result and a 1-bit "carry" output. We refer to the carry output as the "carry-out", or "co". Figure 8.5 shows a black box diagram for this example.

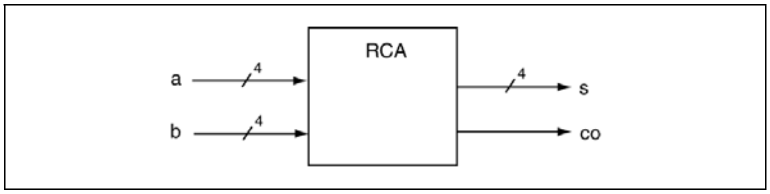


**Figure 8.5: Top-level BBD for the ripple carry adder.**

If we had used BFD, we would start with a truth table. The problem with BFD is that circuit has eight inputs; the associated truth table would require $2^8$, or 256 rows[3]. We instead solve this problem using IMD, which leverages the fact that we already designed two different one-bit adders (the HA and the FA). This problem requires that we design a 4-bit adder, so we assemble the 1-bit adders in such a way as to create a 4-bit adder. Figure 8.6 shows the final solution for this problem.

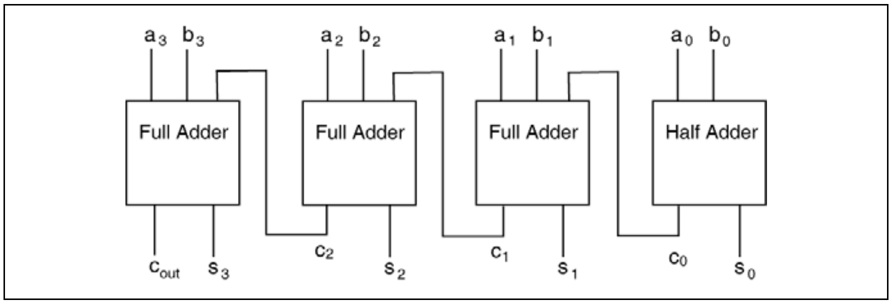


**Figure 8.6: Lower-level BBD for a 4-bit Ripple Carry Adder.**

[3] While this would be possible, such work is more suited to an academic administrator rather than a digital designer.

The circuit in Figure 8.6 has four specially connected 1-bit adders. To ensure the correct answer on the circuit's outputs, each 1-bit adder must generate the "correct" values for both the sum and carry-out. 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 locations moving from right to left (except for the HA). For example, generating the correct second-from-right sum bit is dependent upon the carry-out from the HA. We refer to this circuit as a ripple carry adder is because the carry must "ripple" from the lower-order adders to the higher-order adders (right-to-left in Figure 8.6). Here are some other useful things to note about this circuit.

- Figure 8.6 uses weightings associated with each bit location as the given numbering implies. Higher the number "indexes", have higher weightings. We refer to the "s3" output bit as the most significant bit (MSB) of the sum while we refer to the s0 output as the least significant bit (LSB) of the sum. This RCA uses the weightings associated with binary numbers for the individual bit locations.

- We completed this design without using truth tables or Boolean equations. We completed this design on a higher level than previous designs, which means the design uses only previously designed modules (HAs & FAs). The design is modular in that we use previously designed modules; the design is iterative in that we place the modules in a repetitive manner. We've thus abstract the RCA design to a higher level.

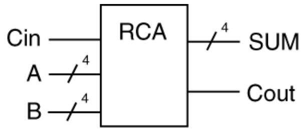
- The notion of the "carry out" ($c_{out}$) in serves as the "fifth bit" and MSB for the addition. In essence, though we added two 4-bit unsigned numbers, we obtained a 5-bit result.

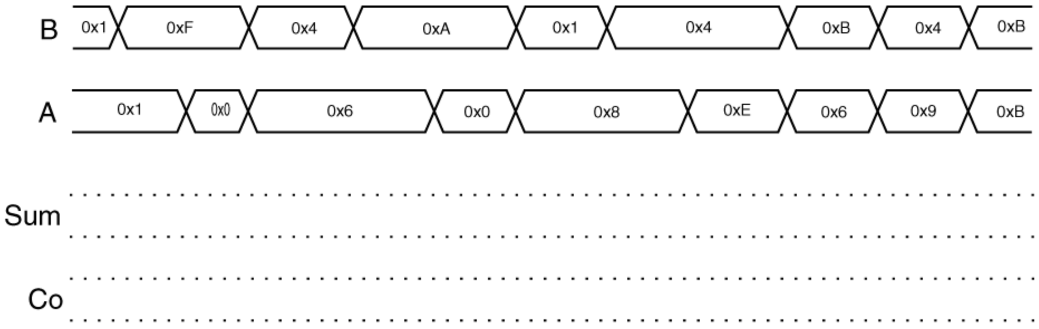
- We refer to the RCA as an "n-bit adder" because if we wanted an 8-bit adder, we simply add four more FAs to the 4-bit RCA design. The act of "adding four more FAs" to the design is simple, but powerful.

None of the modules in the RCA has control inputs, so this device uses no control.

**Example 8-4: Timing Diagram for an 8-bit RCA**

Use the block diagram of the 4-bit RCA below to complete the accompanying timing diagram. For this problem, assume the Cin input is always '0'.





**Solution**: Convince yourself what is going on in this problem by examining the timing diagram in Figure 8.7:. 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 bit (MSB). The possible range for a 4-bit binary number is 0x0 to 0xF (equating to 0 to 15 in decimal). Figure 8.7: shows the solution.

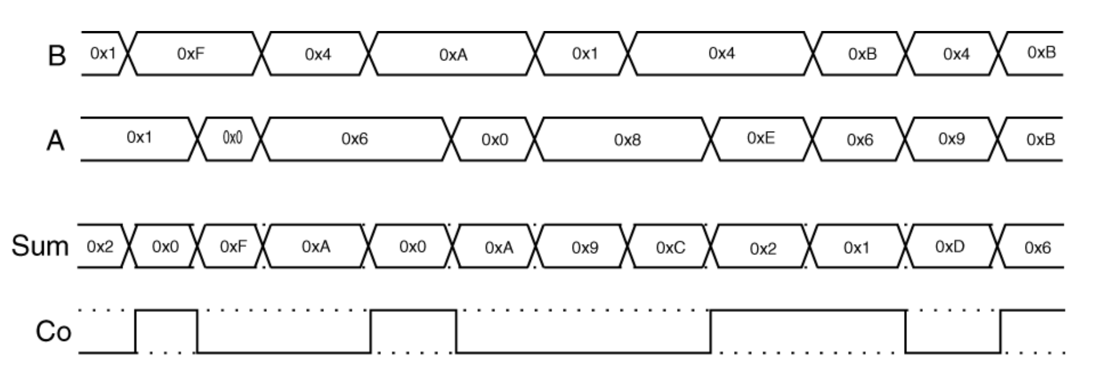


**Figure 8.7: The solution to** Example 8-4**.**

**Example 8.5: A Component-based 8-Bit RCA**

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. Also, state what controls the circuit.

**Solution**: The first step in this solution is to draw the BBD; Figure 8.8 shows the BBD for this problem. We add a carry-in input (Cin) to the BBD, which is arbitrary, but we include it so that we can cascade this circuit if we need to. If we choose not to cascade this circuit, we connect the carry-in input to ‘0’, which prevents the Cin input from affecting the SUM output.

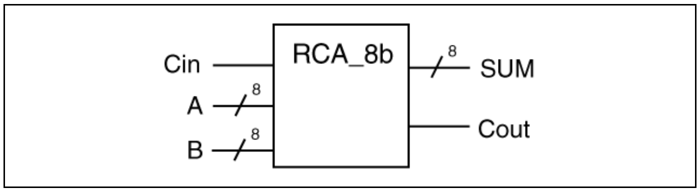


**Figure 8.8: The high-level black box diagram for this problem.**

Although you may not have noticed it in the previous example, the “thing” that allowed you to increase the width of your 1-bit adder was the fact that the FA was a 1-bit adder that added three different bits together to generate 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 (a few gates), but it left the circuit with the inability to be “cascaded” with other RCAs. This cascading of RCAs allows us to effortlessly build RCA of greater widths. The solution to this example would be to substitute a FA for the HA of Figure 8.6; Figure 8.9 shows the result of this step, which is a 4-bit RCA with a carry-in (Cin) input.

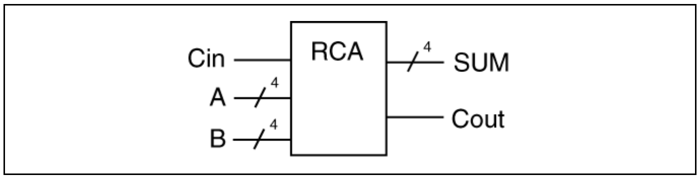


**Figure 8.9: Black box diagram for the 4-bit RCA we use in the solution.**

The next step in this solution is to draw a lower-level BBD that shows the all the modules we need for the solution. The problem states that we need two 4-bit RCAs; this means we must divide the circuit’s 8-bit inputs and output between the two RCAs. Figure 8.10 shows the result of this step.

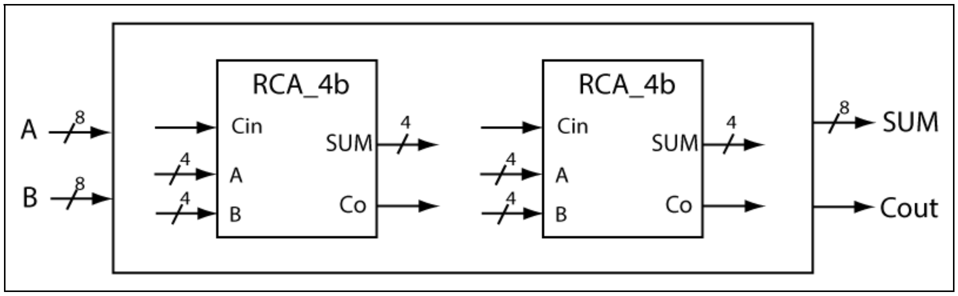


**Figure 8.10: The internal modules for this solution.**

The final part of the solution is to connect the internal modules of Figure 8.10. Figure 8.11: shows the result of this step. This circuit has no control features. There are a few things to notice in Figure 8.11:.

- There is no notion of HAs and FAs because it is a relatively high-level model
- It is a common assumption in digital-land to include a “carry-in” in RCAs. In this case, we are assuming that the lowest-order bit uses an FA instead of a HA.

- Parenthetical notation shows that the total number of input bits for the two operands; we subsequently divide these between the two individual 4-bit RCAs. You always need to provide notation to indicate the routing of the associated signals. We use the same style of routing for the "sum" output.
- The "Cin" input to the lower-order RCA is "tied to ground". A requirement of circuit diagrams is that we must account for every input in the schematic diagram by connecting them to a signal, or assigning it a known and constant.
- This circuit works as an 8-bit adder in this cascade formation because the "carry-out" from the lower-order RCA connects to the "carry-in" of the higher-order RCA. This is common in digital-land also as many digital ICs allow you to connect many of the same ICs together to increase the overall width (or length in some cases) of signals. We refer to the act of connecting things together in this manner as *cascading*.

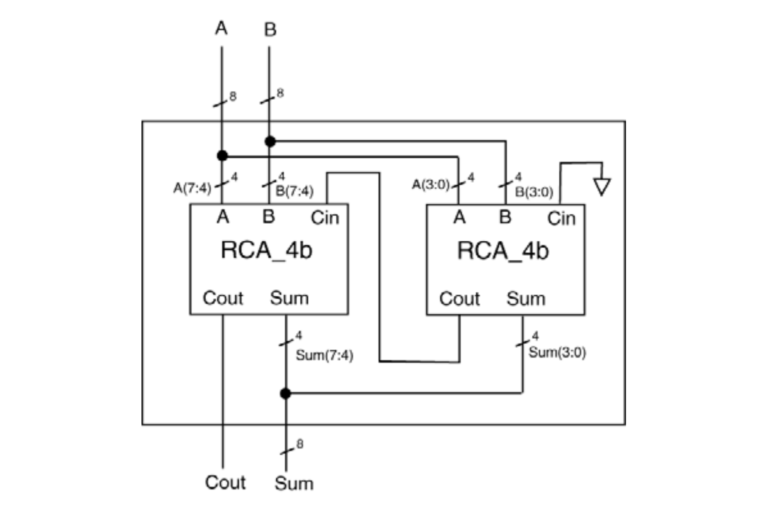


**Figure 8.11: The final solution for this example.**

The RCA has no control input, so this circuit has no control ability; the outputs always respond to the inputs in the same way.

**Example 8-6: Signal Changing Circuit**

Design a circuit that increases the value on an 8-bit unsigned binary input signal by 30. The output of this circuit is always valid. Also, state what controls the circuit.

**Solution**: The first step in this solution is to draw a BBD, means we must figure out the width of both the data inputs and data. The problem states the output data width should always be correct, which means that when we add the value to the number, the sum output is always correct. Thus, the width of the output data must be one bit greater than the width of the input data, which makes the output data width nine bits. Figure 8.12 shows the associated BBD.

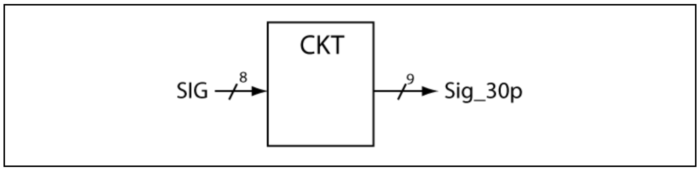


**Figure 8.12: Black box diagram for this problem.**

The next two steps in this problem are to establish the lower-level modules and connect them in such a way as to solve the problem. Figure 8.13 shows the final solution to this problem; here is the thought process that leads to that solution.

- The circuit increases the input value by 30, so there must be an RCA in the circuit.
- Because RCAs have two inputs to account for, we discern that one input must be the external input value while the other input must be "hardcoded" to a value of 30.
- The width of the output is one bit greater than the width of the input. The RCA's carry-out becomes the MSB. This fact is not obvious, so we clearly note it in Figure 8.13.

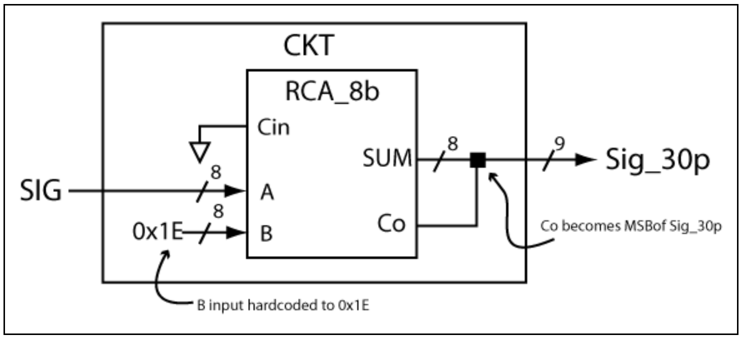


**Figure 8.13: The final solution for this problem.**

The next thing we must do is establish what controls the circuit's operation. The RCA in this circuit has not control inputs, so this circuit has no control features.

## 8.6 Digital Design Foundation Notation: The RCA

We consider the RCA to be a Digital Design Foundation module. The RCA is a controlled circuit; Figure 8.14 shows the RCA in appropriate digital design foundation notation. As you would expect from an adder-type circuit, the RCA adds the two input operands (A & B) and the carry to generate the SUM output. Note the RCA has no control inputs, which means the device always performs the same operation on the three data inputs. The RCA's CO output provides status for the RCA's addition operation. Table 8.4 provides a description of all the inputs and outputs to the RCA.

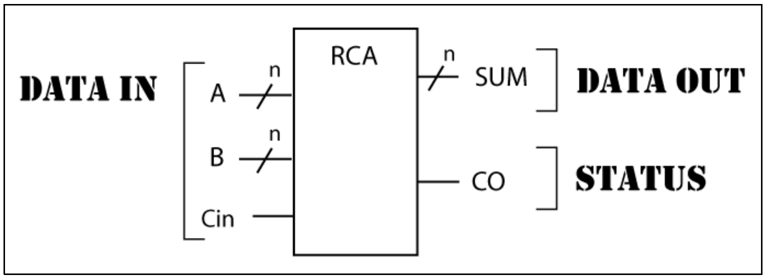


**Figure 8.14: Data, control and status signals for a RCA.**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **A** | One of two multi-bit addends (or operands). The data width of the two addends is equivalent. |
| | **B** | One of two multi-bit operands. The data width of the two addends is equivalent. |
| | **Cin** | A “carry in” input. |
| OUTPUT DATA | **SUM** | The result of summing the three inputs: two addends and the Cin input. |
| CONTROL | **n/a** | - |
| STATUS | **Co** | A “carry-out” signal; this signal shows when the summation operation has generated a carry. The carry is effectively the “n+1” bit of an n-bit RCA. |

**Table 8.4: The foundation matrix for a RCA.**

## 8.7 Chapter Summary

- Iterative Modular Design (IMD) is a more powerful design method than brute force design (BFD) because it bypasses the constraints presented by the truth tables and the entire BFD approach. There are several standard digital circuits that we design using IMD, the RCA is one of them.
- The half-adder (HA) is a single-bit adder with two inputs (addends) and a result (sum) and carry output.
- The full-adder (FA) is a single-bit adder with three inputs (two addends and a carry-in) and a result (sum) and carry output.
- The ripple carry adder (RCA) is an arithmetic circuit comprised of FA, and sometimes a HA for the LSB. We define RCAs as "n-bit" adders, where n is both the width of the two non-carry-in operands and the width of the sum output.
- We can easily "cascade" two n-bit RCAs to form a RCAs of width 2n. This modification requires that the higher-order RCA have a carry-in input.
- The notion of what controls a circuit is always of importance to the digital designer; the options are 1) no control, 2) internal control, 3) external control, or 4) circuit controlled. A given circuit can have more than one form of these controls.

## 8.8 Chapter Exercises

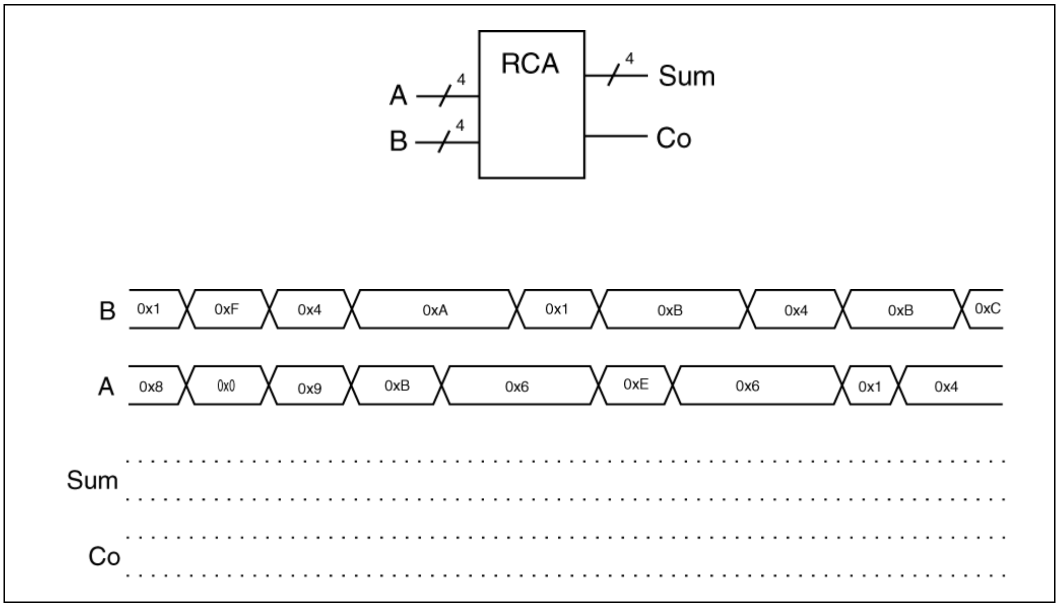
**1)** Briefly describe why we should always connect all unused input signals to either power or ground in all digital designs. In other words, why do we not what to "leave inputs hanging" or "leave inputs floating".

**2)** If you were to design a 10-bit RCA using the BFD approach, briefly explain how many rows with the associated truth table have?

**3)** There are adders out there that fall into the category of "look ahead carry" adders. Briefly explain why these would output a result faster than a RCA.

**4)** In your own words, briefly explain how the RCA got its name.

**5)** Complete the timing diagram below considering the given schematic symbol.



## 8.9 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**1)** Design a circuit that always increases the value of a 4-bit signal by two. The output of this circuit is always a valid summation.

**2)** Design a circuit that always increases the value of an 8-bit signal by seven. This circuit has an output VALID that indicates when the 8-bit output sum value is valid.

**3)** Design a circuit that always increases the value of an 8-bit signal by six. The width of the output data should always reflect the result of the addition.

**4)** Design a 10-bit RCA using only two 4-bit RCAs, an HA, and a FA.

**5)** Design a circuit that adds four 8-bit values and always returns the proper summation on the circuit’s output. The circuit’s output should not have a carry-out-type signal.

**6)** Design a circuit that always doubles the value of the carry in before summing it with the circuits two 10-bit input values. The output summation of this circuit is always valid.

**7)** Design a circuit that multiplies a single 8-bit input by three. The resulting output is always correct.

**8)** Design a circuit that multiplies a single 10-bit input by five. The resulting output is always correct.

**9)** Use an RCA to design a circuit that blinks a single LED output at the highest rate possible using that RCA.

**10)** Use an RCA to design a circuit that blinks a single LED output at the 1/16th the highest rate possible using that RCA.

**11)** Design a circuit that adds two 5-bit digital values. If an external button is being pressed, the circuit outputs the correct result of the summation; otherwise, the circuit outputs all zeros. Assume the pressed button outputs a logical ‘1’ and the unpressed button outputs a logical ‘0’. Assume the circuit output is 6-bits wide.

**12)** Design a circuit that adds two 4-bit digital values. If an external button is being pressed, the circuit outputs the correct result of the summation; otherwise, the circuit outputs all ones. Assume the pressed button outputs a logical ‘1’ and the unpressed button outputs a logical ‘0’. Assume the circuit's output is 5-bits wide.

**13)** 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 that indicates when the 10-bit output is a valid represents the actual sum of the five input values. You can only use 10-bit RCAs for this circuit.

**14)** Use two HAs and a minimal amount of additional logic to create a FA.

# 9 Boolean Functions and DeMorgan's Theorem

## 9.1 Introduction

Digital design depends on the use of various model types to represent digital circuits. Digital designers need to be adept at modeling circuits in a way most appropriate for a given situation. Using the various digital theorems allows us to represent circuits in different but functionally equivalent ways. One more use of DeMorgan's theorem is to help us transform Boolean equations into functionally equivalent forms.

**Main Chapter Topics**

**DEMORGAN'S THEOREMS:** Probably the most widely used theorem in digital design, DeMorgan's theorems can transform equations into functionally equivalent forms.

**REPRESENTING BOOLEAN FUNCTIONS:** There are many ways to represent Boolean functions; this chapter describes some of the more common approaches.

**Chapter Acquired Skills**

- Be able to represent functions using standard SOP form
- Be able to represent functions using standard POS form
- Be able to describe standard sum and standard product terms
- Be able to form minterm and maxterm expansions from reduced Boolean equations
- Be able to represent functions using SOP and POS forms
- Be able to represent functions using compact minterm and compact maxterm forms.
- Be able to transfer back and forth from any one function form to any other function form.

## 9.2 Representing Boolean Functions

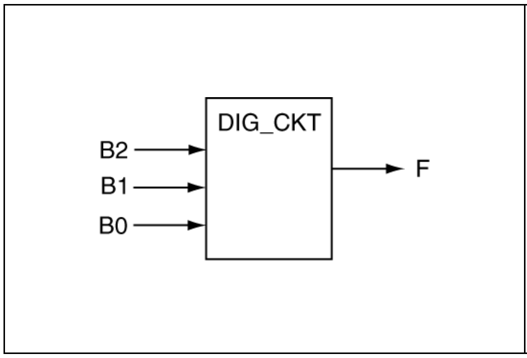
A Boolean function, or "function", is an equation that describes an input/output relationship of a module in terms of digital logic. There are many different ways of modeling this input/output relationship; you've seen three main approaches: truth tables, Boolean functions, and circuit models.

There are a few important things to notice about input/out relationships. First, these three representations are functionally equivalent; so they say the same thing but say it in three different ways. Secondly, you'll see that some function representations are more appropriate than others.

**Example 9.1: The First BFD Problem Revisited**

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

**Solution**: The solution to Example 9.1 included a black box diagram (Figure 9.1(a)), a truth table (Figure 9.1 (b)), a Boolean expression (Figure 9.2), and the final circuit diagram (Figure 9.3).



| 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 |

**(a)** **(b)**

**Figure 9.1: The black box model and completed truth table for Example 9.1.**

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

**Figure 9.2: A Boolean expression describing the solution to Example 9.1.**

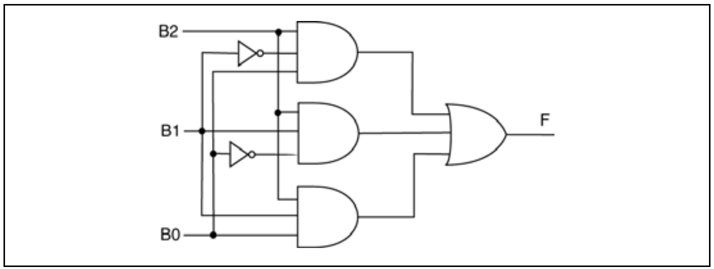


**Figure 9.3: The circuit model that solves Example 9.1.**

## 9.3 DeMorgan's Theorems

The list of theorems provided in a previous chapter is relatively long. Modern digital design rarely directly uses all of these theorems, but they use some of them quite often; DeMorgan's is one of those theorems. We can generate different representations of a Boolean equations from an application (or multiple applications) of DeMorgan's theorem. Figure 9.4 shows once again the Boolean equation that describes a solution to our first design problem. We refer to the form of this equation as the sum of products (SOP) form. This name makes sense in that there are three terms in the equation that are logically multiplied together; the equation then logically adds the product terms.

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

**Figure 9.4: The solution to the previous example listed again here.**

Another widely used Boolean equation form is 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. In the SOP form, you wrote the Boolean equation based on the rows of the truth table that contained a '1'. You found which rows contained a '1' in the output and you included the product term for that row in the final Boolean equation.

Inverting the **F** output officially describes the same function (the right-most column in Figure 9.5). Note the right-most column in Figure 9.5 is the same as the **F** column except we invert the associated values, so the two right-most columns of Figure 9.5 have a complementary relationship. Generating an equivalent POS form for the truth table in Figure 9.5 is similar to the approach for generating the SOP form. The only difference is that we need to apply DeMorgan's theorems multiple times to translate the equation to POS form.

| B2 | B1 | B0 | F | !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 9.5: 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. DeMorgan's theorem is also useful in other fields such as discrete mathematics, computer programming, and other various flavors of engineering. Table 9.1 shows DeMorgan's theorems in both two variable and generalized forms. The final form in Table 9.1 emphasizes the fact the "variables" in the original listing of DeMorgan's theorem are not necessarily Boolean variables. The symbols 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 9.1: 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 with a '1' on the output of the complemented output[1]. This first step is similar to generating SOP form but you're actually generating an equation in SOP form for the complement of the output[2]. Table 9.2 shows the set of equations generated by seeking a POS expression for the given function. An explanation of each row in Table 9.2 follows the table.

[1] Looking for 1's in the inverted output column is the same as looking for 0's in the non-inverted output column.

[2] 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 9.2: Generating a POS form from multiple applications of DeMorgan's theorem.**

**a)** This equation is SOP form, which we generate by listing the product terms for the 0's of the **F** column or the 1's of the **!F** column. This is a valid SOP form for the complemented output, but we're looking for a POS form for the uncomplemented output.

**b)** We complement both sides of the equation in Table 9.2(a), which preserves the equality.

**c)** Since the double complement of a variable equals that variable, the double-complemented **F** on left side of the equals sign becomes uncomplemented. Our ultimate goal is to generate an equation for **F** in POS form so we still need to massage this equation. The expression on the right side of the equals sign shows the results after the first application of DeMorgan's theorem. The product terms are now complemented and are ANDed together, thus 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. The overbar is distributed to the individual components of the product terms and we switch the logic operators from AND to OR.

**e)** A Boolean algebra axiom allows us to remove the double complements from the variables. The 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 common forms so note that 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[3]. The SOP and POS forms are functionally equivalent, which means they describe the same input/output relationship, but in different ways.

## 9.4 Minterm & Maxterm Representations

Without you knowing it, we previously exposed you to *minterm representations* and *maxterm representations* of functions. For this section, let's return to the design from a previous chapter. Figure 9.6 shows the equation we were previously working with. From the truth table of Figure 9.6, you generated the Boolean function in Equation 9.1 to describe 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, we're actually listing this equation in what we refer to as "standard SOP form". You know that the equation is in SOP form because you can see that there is a summation of many product terms. So what makes it a standard SOP form?

[3] 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.

| 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 9.6: The generic function from 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 one instance of each of the function's independent variables in either complemented or uncomplemented form. We consider the product terms in Equation 9.1 something special in that they are *standard product terms*[4]. When we're describing a function using standard product terms, we 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 9.3 shows the product terms for three-variable (A, B, C) function.

Table 9.3 shows that we also label the product terms as "minterms"s which is simply another name for a standard product term. Digital design also refers to an equation in standard SOP form as a *minterm expansion* of the function. Equation 9.2 shows the standard SOP form of the function from the previous example (we switched from B2, B1, and B0 to A, B, and C to make them easier to write).

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

**Equation 9.2**

There is also a standard product of sums (POS) form, which contains a logical multiplication of standard sum terms. We refer to a standard sum term 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[5]. Alternatively, equivalently, maxterms describe the 1's in the output of the complemented function. Equation 9.3 shows the standard POS form of the function; we sometimes refer to this form 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**

[4] Later in this set of notes you'll see that listing all the terms as standard product terms not generally done.

[5] More specifically, maxterms describe the location of the 0's in the rows containing 0's for the uncomplemented output.

| 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 9.3: A listing minterms and maxterms for the each combination of circuit inputs.**

There is a special relationship between the minterms and maxterms. For a given row in the truth table, the minterms and maxterms are complements of each other; Figure 9.7 shows this property. To generate a minterm from a maxterm (or vice versa), you first complement it and then tweak it using DeMorgan’s theorem. Figure 9.8 shows an example of this relationship for the fourth row in Table 9.3. In Figure 9.8(a), we complement the equation for the given minterm and then DeMorganized to generate the associated maxterm.

$$Maxterm = \overline{Minterm}$$
$$Minterm = \overline{Maxterm}$$

**Figure 9.7: The secret relationship between minterms and maxterms.**

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

**(a)**

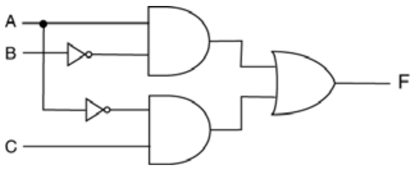
$$F(A,B,C) = F(1,0,0) = \overline{A}+B+C$$
$$\overline{F(A,B,C)} = \overline{F(1,0,0)} = \overline{\overline{A}+B+C}$$
$$\overline{F} = \overline{\overline{A}}\cdot\overline{B}\cdot\overline{C}$$
$$\overline{F} = A\cdot\overline{B}\cdot\overline{C}$$

**(b)**

**Figure 9.8: The complimentary relationship between minterms and maxterms.**

**Example 9.2: Circuit Form to Equation Transformation**

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



**Solution**: There are many ways to represent functional relationships in digital-land; you've seen several equation forms (SOP & POS), truth tables, and timing diagrams. There are functionally equivalent ways to represent any given function, so 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 in SOP form to a circuit model in POS form. There are many ways to solve this problem; we take the most straightforward approach. Here are the steps to solve this problem:

1) Write out the equation implemented by the circuit
2) Expand the equation into standard SOP form
3) Use the SOP equation to generate a truth table
4) Write an equation for the complemented output
5) Complement the equation and DeMorganize[6] the result until the equation is in POS form
6) Use the derived POS equation to re-implement the circuit

1) Write the equation implemented by the circuit. The circuit is in SOP form; from the circuit, you can see that there are two product terms (two AND gates) that are logically added together (one OR gate). Figure 9.9 shows the initial equation.

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

**Figure 9.9: The initial equation from this example.**

2) Although this equation is officially in SOP form, we need to transform it into 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 add back by logically multiply the equation by '1'. Thinking back to the original Boolean algebra theorems, you'll find that: (x + !x = 1). Note the first product term is missing the **C** variable. We add it by multiplying the first product term by (**C**+**!C** = 1) which does not alter the value of the product term. Figure 9.10 shows the derivation of the product terms; Figure 9.11 shows the final expanded equation.

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

| | |
|---|---|
| $A\cdot\overline{B} = A\cdot\overline{B}\cdot(C+\overline{C})$<br>$A\cdot\overline{B} = A\cdot\overline{B}\cdot C + A\cdot\overline{B}\cdot\overline{C}$ | $\overline{A}\cdot C = \overline{A}+C\cdot(B+\overline{B})$<br>$\overline{A}\cdot C = \overline{A}\cdot C\cdot B + \overline{A}\cdot C\cdot\overline{B}$<br>$\overline{A}\cdot C = \overline{A}\cdot B\cdot C + \overline{A}\cdot\overline{B}\cdot C$ |

**Figure 9.10: Expanding the product terms from the original equation.**

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

**Figure 9.11: The final expanded equation.**

**3)** Now that the terms look familiar, we enter them into a truth table. Figure 9.12 shows that we place a '1' in the **F** column for the corresponding product terms in the equation derived in the previous step.

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1· (!A·!B·C) |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1· (!A·B·C) |
| 1 | 0 | 0 | 1 · (A·!B·!C) |
| 1 | 0 | 1 | 1· (A·!B·C) |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

**Figure 9.12: The truth table showing the implicated product terms.**

**4)** The next step is to write an equation for the complemented output. Figure 9.13 shows that we do this by adding a complemented **F** column to the previous truth table. Using the table in Figure 9.13, we can write an SOP equation for the complemented output; this result equation appears Figure 9.14.

| A | B | C | F | !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 |

**Figure 9.13: The truth table expanded to show the complemented output.**

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

**Figure 9.14: The final expanded equation.**

**5)** The final equation is an expression for **!F** (another way of saying a complemented **F**). We want an expression for **F** (as opposed to **!F**), so we complement both sides of the equation and DeMorganize the result a bunch of times. Figure 9.15 shows these steps.

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

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

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

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

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

**Figure 9.15: The final solution this example.**

**6)** Finally, the last step is to draw a circuit model for the final equation of the previous step; Figure 9.16 shows the result of this step. This example turned out to be a long problem as it shows many of the useful and versatile properties associated with Boolean algebra. N note in the diagram below that the AND gate has some extended wings to handle the larger number of inputs.

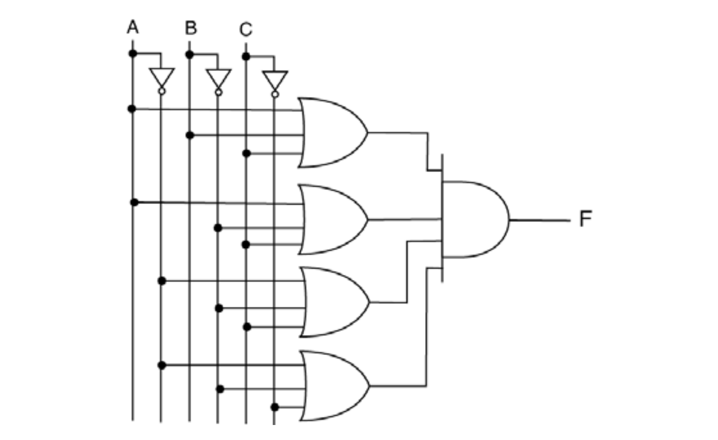


**Figure 9.16: The final solution to this example.**

## 9.5 Compact Minterm & Maxterm Function Forms

Representing functions in standard SOP or POS forms is klunky, so we use compact minterm forms or compact maxterm forms instead. The compact minterm and maxterm forms list the decimal index (see the right-most column of Table 9.3) 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: we use the summation symbol for listing minterms (since it is a "summing" of product terms) and the capital Pi symbol for listing maxterms[7]. Figure 9.17 shows the compact minterm and compact maxterm forms for the example we're working with. These compact forms always need listing as a function of the independent variables. If you did not include all

[7] If you consult the right source, you'll find that the Pi symbol is associated with multiplication.

of the independent variables, you would not be able to expand the list into standard sum or standard product terms.

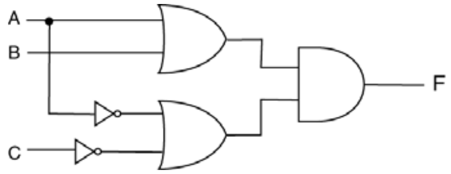
| | |
|---|---|
| $F(A,B,C) = \sum(5,6,7)$ | $F(A,B,C) = \prod(0,1,2,3,4)$ |
| **(a)** | **(b)** |

**Figure 9.17: Compact minterm and maxterm forms for the current example.**

You now know 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, so all of these different forms are functionally equivalent. This means that you should be able to change from any one of the forms to any other one of the forms. However, while switching from one form of a function to another is painfully exciting, it is not represent digital design, as most digital design textbooks lead you to believe.

**Example 9.3: Circuit to Equations Transformation Again**

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 a previous example; the steps are the same but you need to apply them in a strange reverse order.

First, write the equation implemented by the circuit. The circuit is in POS form; the circuit has two sum terms (two OR gates) that are logically multiplied together (one AND gate). Figure 9.18 shows the resulting equation.

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

**Figure 9.18: The initial equation derived from the problem description.**

We need to put the above equation into SOP form so we can easily enter it into the truth table. If we complement both sides of the equation and then DeMorganize it, we get an expression for **!F** in SOP form.

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

**Figure 9.19: DeMorganizing the original equation.**

From here, we need to expand each of the product terms to include each of the independent variables. Figure 9.20 shows that we use the same technique as in the previous problem.

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

**Figure 9.20: Expanding the derived equation.**

The equation in Figure 9.20 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 (that's what we need to give us an equation for this function in SOP form). Figure 9.21 shows the results of this description. Figure 9.22 shows that we can now write an equation for F.

| A | B | C | F | !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 |

**Figure 9.21: Including the complemented output in the truth table.**

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

**Figure 9.22: Writing the equation for F.**

Figure 9.23 shows the final step in this problem, which is drawing a model for the final circuit implementation.

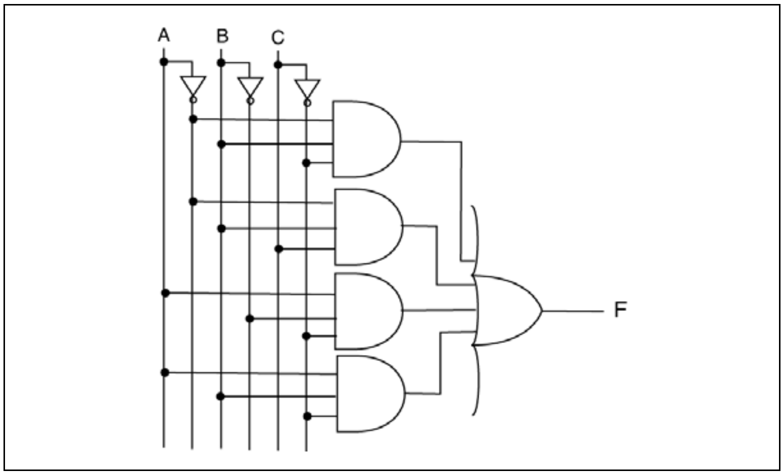


**Figure 9.23: The final circuit solution for Example 9.3.**

**Example 9.4: Half Adder in Standard POS Form**

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

**Solution**: For this solution, we assume you still remember the half adder.

Step 1) Define the Problem: Draw a black box diagram of the final circuit; Figure 9.24 shows this result. Table 9.4 and Table 9.5 show the original truth table and the truth table including the complemented outputs, respectively. In Table 9.5, we use an "!" (the exclamation mark) prefix to variables to indicate a complement of the variable. Nerdy people know this symbol as the bang character.

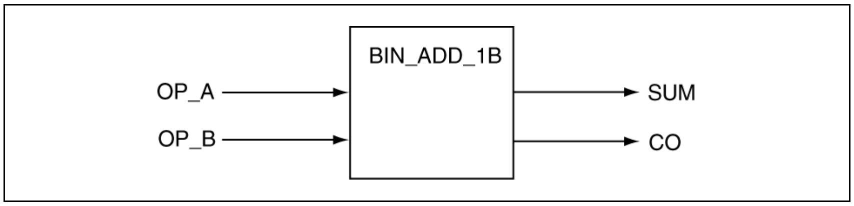


**Figure 9.24: The black-box diagram for the example problem.**

| OP_A | OP_B | SUM | CO |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

**Table 9.4: 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 9.5: The truth table including complemented outputs.**

Step 2) Describe the Solution: For this problem, you'll need to generate two Boolean expressions: one for the SUM and the other for the CO.

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

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

**Equation 9.4: The starting equations for Example 9.4.**

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

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

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

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

**Equation 9.5: The SUM path from SOP to POS for Example 9.4.**

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

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

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

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

**Equation 9.6: The CO path from SOP to POS for Example 9.4.**

Step 3) Implement the Solution: The final step involves translating the Boolean expressions in Equation 9.5 and Equation 9.6 into circuit form. Figure 9.25 shows the final gate-level implementation.

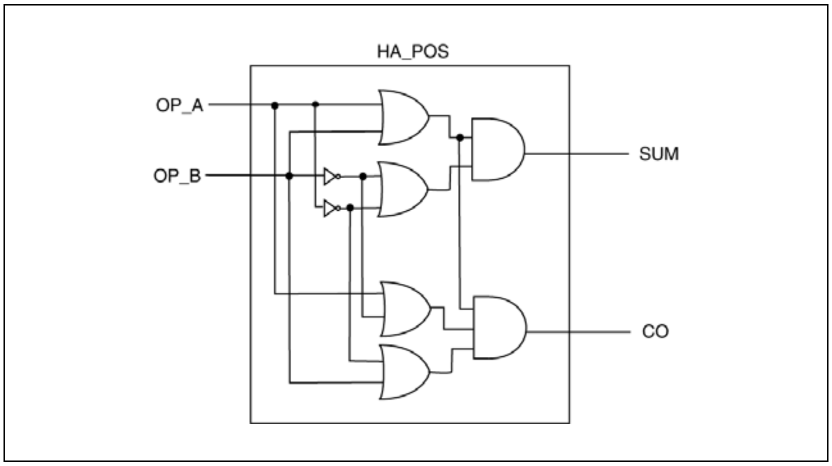


**Figure 9.25: The circuit representation of the final solution for Example 9.4.**

Equation 9.7 lists both the SOP and POS forms for the CO output while Equation 9.8 lists the SOP and POS forms for the SUM output. The SOP and POS forms for a given output are functionally equivalent. Finally, Figure 9.26 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 9.7: The CO path from SOP to POS for Example 9.4.**

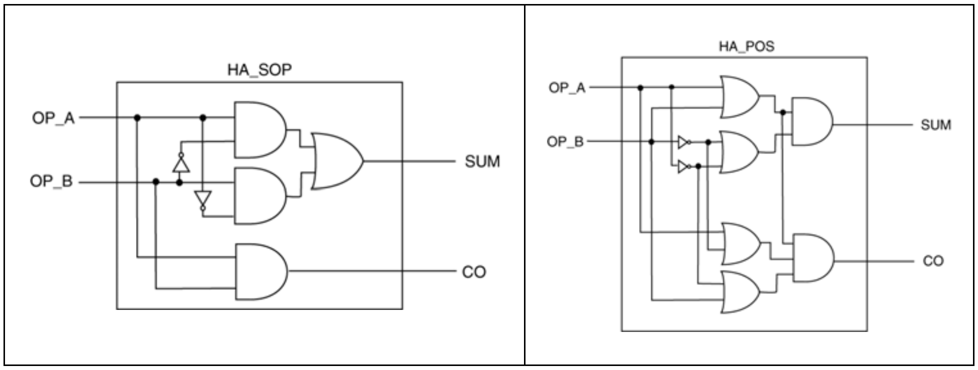
$$SUM = \overline{OP\_A} \cdot OP\_B + OP\_A \cdot \overline{OP\_B}$$

is functionally equivalent to:

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

**Equation 9.8: The CO path from SOP to POS for Example 9.4.**

Figure 9.26 provides some interesting and important information as it relates to functional equivalency. There are now two functionally equivalent ways to model a HA using Boolean equations. Because the circuit in Figure 9.26(a) uses less hardware than the circuit in Figure 9.26(b), you can conclude that Figure 9.26(a) is the better approach. The difference in two gates does not seem like enough in the context of this example, but it’s more meaningful if you circuit required thousands (or millions) of HAs. This is a brief introduction to minimum cost concept, a topic we cover in a later chapter.



**(a)** **(b)**

**Figure 9.26: A comparison of the SOP (a) and POS (b) circuit diagrams for the half adder.**

## 9.6 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. We can simplify Boolean expressions 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. We typically use DeMorgan’s theorem to generate a POS equation from a truth table. The SOP form has by multiple product terms that we logically sum together while the POS form has sum terms that we logically multiply together.
- We can represent Boolean functions in many different forms including standard and reduced SOP, standard and reduces POS, and compact minterm and maxterm forms.
- We can represent a function with a truth table in two ways; either we present the positive version output (a representation of a non-complemented output variable) or the negative version (a representation of the complemented output variable). These two outputs are complements, or inversions, of each other.

## 9.7 Chapter Exercises

For all of the following problems, SOP and POS refer to standard SOP and standard POS.

**1)** Briefly explain why it is proper to list all the independent variables in the compact minterm and maxterm forms.

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

**3)** 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)**

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

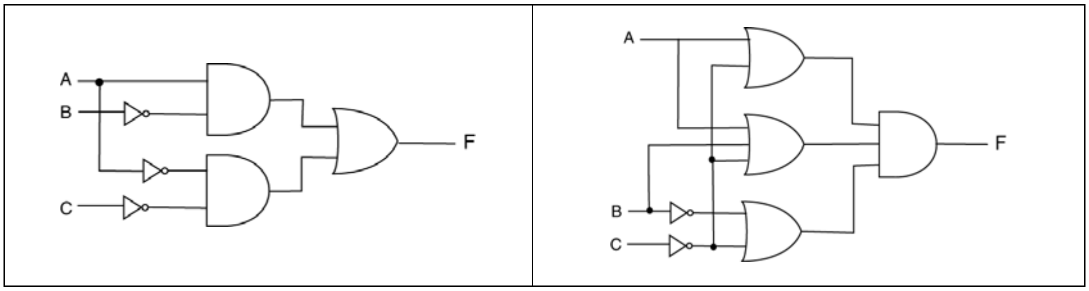
**5)** Convert the following functions to SOP form.

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

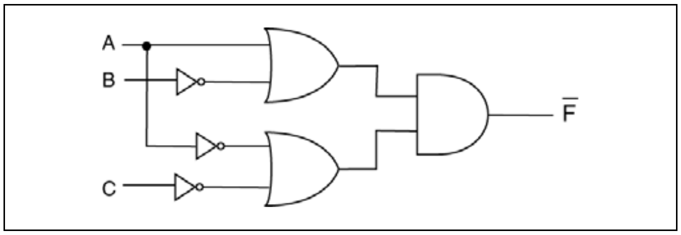
c) $F(X,Y,Z)=(\overline{X}+\overline{Y}+Z)\cdot(\overline{X}+Y+\overline{Z})\cdot(X+\overline{Y}+Z)\cdot(\overline{X}+\overline{Y}+\overline{Z})$

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



**(a)** **(b)**

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



**8)** Represent the following equation in compact minterm and maxterm forms.

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

**9)** Convert the following Boolean functions to both compact minterm and maxterm forms.

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

**10)** Write a reduced Boolean equation in SOP and POS forms for each of the following functions.

$\mathrm{F3(A,B,C)} = \sum(3,4,5,6)$ $\mathrm{F4(A,B,C)} = \prod(2,4,6,7)$

$\mathrm{F5(A,B,C,D)} = \sum(0,2,4,6,8,9,12,13)$ $\mathrm{F6(A,B,C,D)} = \sum(0,2,5,8,10,13,15)$

$\mathrm{F7(A,B,C,D)} = \prod(4,5,6,12,13,14,15)$ $\mathrm{F8(A,B,C,D)} = \prod(2,3,6,7,9,11,13,15)$

## 9.8 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**1)** Design a circuit that has four inputs and three outputs. The four inputs are considered 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.

**2)** 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 opens.
- If someone 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 opens.
- The person at door C can never be in the room alone but can be in the room with anyone else.

# 10 More Standard Logic Gates

## 10.1 Introduction

This chapter continues up the digital design learning curve by introducing four new logic gates. Though you've been using AND & OR gates (and inverters) in your designs, these are not the most common gates in digital design.

**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.

**LOGIC GATE ABSTRACTIONS:** The chapter introduces the notion that we can configure various logic gates as inverters, switches, or buffers.

**Chapter Acquired Skills**

- Be able to describe the notion of *functionally complete* as it applies to logic gates
- Be able to use NAND, NOR, XOR, and XNOR gates in digital circuits
- Be able to configure standard logic gates as inverters, switches, and buffers

## 10.2 NAND Gates and NOR Gates

We form the NAND and NOR gates by complementing the output of AND & OR gates, respectively[1]. The names NAND and NOR are a shortened version of NOT-AND (for NAND) and NOT-OR (for NOR). Figure 10.1 shows that we can model the NAND & and NOR gates by adding an inverter on the output of the AND & OR gates. Figure 10.2 shows the two new gate symbols for the NAND and NOR gates. Figure 10.3 shows the truth tables associated with the NAND and NOR functions. The truth tables in Figure 10.3, show that the outputs of the NAND and NOR gates are in fact complemented versions of AND & OR gates, respectively.

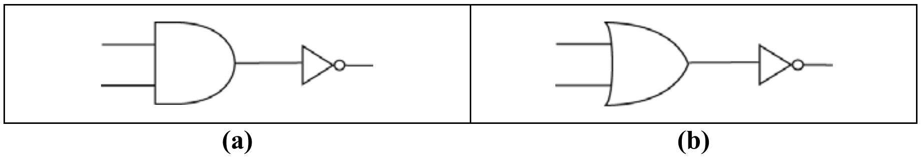
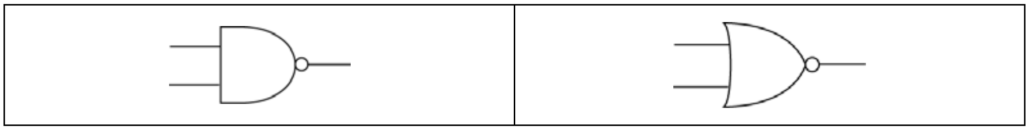


**Figure 10.1: Functional equivalent models for the NAND (a) and NOR (b) logic gates.**

[1] 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.



**(a) (b)**

**Figure 10.2: 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 |

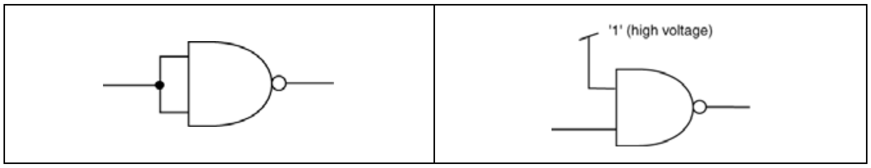
| A | B | $F = \overline{A + B}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**(a) (b)**

**Figure 10.3: Truth tables for the NAND (a) and NOR (b) logic functions.**

One of the advantages that NAND and NOR gates have over AND & OR gates is that they are functionally complete. This means that a NAND gate (or a series of NAND gates) can implement any Boolean function[2]. In other words, we can use a NAND gate to generate an AND function, an OR function, or a complement function (INVERTER). Note from the truth table for the NAND gate in Figure 10.3(a) that there are two ways to create an inverter from a NAND gate:

1) The first and fourth rows of the NAND gate's truth table indicate that if the two inputs to the NAND gate are equivalent, the output is an inversion of the input. We implement this in hardware by connecting the same signal to both inputs[3] of ta two-input NAND gate; Figure 10.4(a) shows this result.
2) The third and fourth rows 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. We implement this in hardware by connecting one NAND gate input to the high voltage; Figure 10.4(b) shows this result.



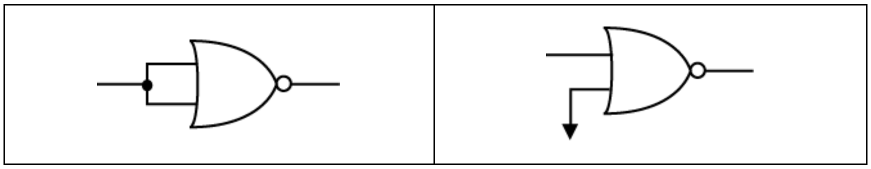
**(a) (b)**

**Figure 10.4: Making an inverter from a NAND gate.**

There are also two ways to force a NOR gate to act as an inverter; Figure 10.1 shows these two approaches. We state these without proof; a few chapter exercises deal with this concept.

[2] The same is true of a NOR gate; we opt not to provide the detailse.

[3] Or all of the inputs if there gates has more than two inputs.

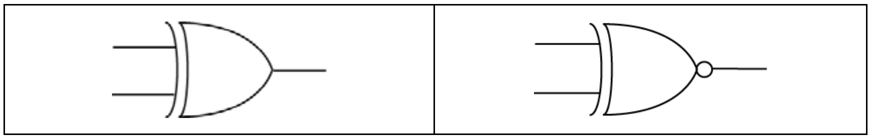


(a) (b)

**Figure 10.5: Making an inverter from a NOR gate.**

## 10.3 XOR and XNOR Gates

The final type of logic gates are the exclusive OR gate (or the XOR gate) and the exclusive NOR gate (or XNOR gate). Figure 10.6 shows the schematic symbol for these two gates. Note the similarity between these gates and the OR and NOR gate symbols. In addition, we often refer to an exclusive NOR gate as an "equivalence gate".



(a) (b)

**Figure 10.6: The exclusive OR (XOR) and exclusive NOR (XNOR) gates.**

Figure 10.7 shows the truth tables that define the XOR and XNOR functions. The XOR and XNOR functions are complements of each other as is true with the OR and NOR gates, etc. Figure 10.8 shows the official Boolean equations describing the XOR and XNOR functions. 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 Figure 10.8(b) does not show[4]: the circled dot.

The equations in Figure 10.8 are both important and useful; you'll use these equations often. You may want to stare at them for a while; I know I sure do[5]. The truth table in Figure 10.7(b) for the XNOR function shows why we refer to it as an equivalence gate: the gate output is a logical '1' when the two gate inputs are equivalent. One thing to note about XOR & XNOR gates: while AND, OR, NAND, and NOR gates can have two or more inputs, XOR and XNOR gates can only have two inputs.

| 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 10.7: Truth tables for exclusive OR (XOR) (a) and exclusive NOR (XNOR) functions (b).**

[4] The equation editor I used when writing this does not contain the required symbol.

[5] Not really.

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

**Figure 10.8: The official equations describing the XOR (a) and XNOR functions (b).**

## 10.4 Logic Gate Abstractions

The fact that we can configure NAND and NOR gates as inverters is useful in digital design. You know the logic behind these gates, but applying basic intuition to these gates allows you to use them in other ways. Once you develop an intuitive feel for basic logic gates, you can use them in many digital designs in clever ways.

Basic gates have three useful functions beyond modeling them as logic elements. These three functions include 1) using gates as inverters, 2) using gates as switches, 3) using gates as buffers. The following verbage more fully describes these functions while Figure 10.9 provides the visual details. For each of these gates, we only consider the case of 2-input gates, keeping in mind that gates of more than two inputs does not apply to XOR-type gates. We omit all mention of the XNOR gate as it is a special case of the XOR gate.

The key to making gates into one of these three functions is connecting an input (or inputs) to the power (logic '1'), ground (logic '0'), of shorting the inputs to the gate. Often times we reference ground as "GND"; we draw with a down-pointed arrow in a circuit diagram. We refer to connecting an input to logic '1' as "tying the input high" or "tied high" and we refer to connecting an input to logic '0' as "tying the input low" or "tied low".

### 10.4.1 Gates as Inverters

When we connect one input to power or ground (logical '1' or '0', respectively), a given gate acts as an inverter. Table 10.1 lists the connections required to create inverter functions from various gates[6]. We previously discussed using NAND and NOR gates as inverters.

| Gate Type | Gate Connected as Inverter |
|---|---|
| NAND | **1)** connect one input to '1' or<br>**2)** have both inputs share the same signal |
| NOR | **1)** connect one input to '0'<br>**2)** have both inputs share the same signal |
| XOR | connect one input to '1' |

**Table 10.1: Gate connections for inverter functionality.**

### 10.4.2 Gates as Switches

The notion of a switch means something we can turn on and turn off. When we "turn off" a gate, we say we are *killing* the gate, which means we prevent the output from changing. This is a useful function in many digital design applications because one input has the ability to disable the gate by forcing the output to a certain value.

[6] Recall that this list does not include AND and OR gates as they are not functionally complete.

| Gate Type | Gate Connected as Switch |
|---|---|
| AND & NAND | connect one input to ‘1’; ‘0’ kills the gate |
| OR & NOR | connect one input to ‘0’, ‘1’ kills the gate |

**Table 10.2: Gate connections for switch functionality.**

### 10.4.3 Gates as Buffers

The word *buffer* is common term electronics. For digital electronics, a buffer function is essentially one that does not change the logic level of an input given signal. This is generally useful because often times you want to pass a signal along in a circuit unchanged. We often use buffering action is conjunction with either a switch or inverter functionality[7].

| 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 10.3: Gate connections for buffer functionality.**

[7] 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).

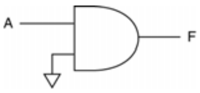
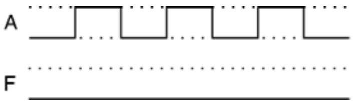
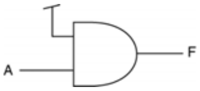
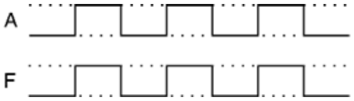
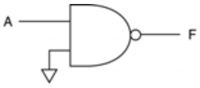
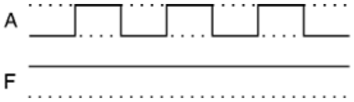
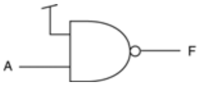
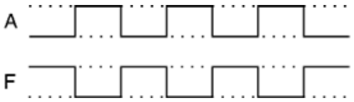
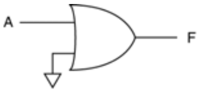
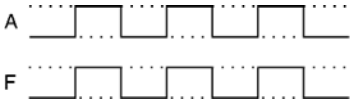
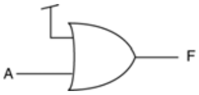
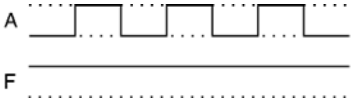
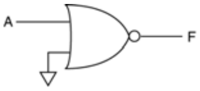
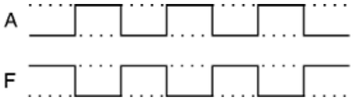
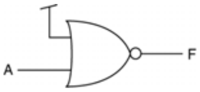
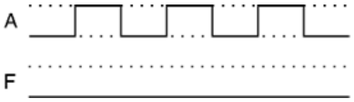
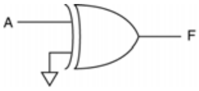
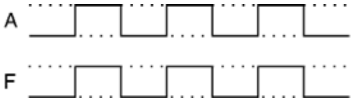
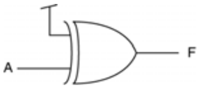
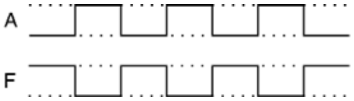
| Gate Configuration | Timing Example | Comments |
|---|---|---|
|  |  | Grounding one input of an AND gate makes the output always ‘0’, which “kills” the gate: ***DEAD*** |
|  |  | Tying one input of an AND gate high allows the other input to pass unchanged to the output: ***BUFFER*** |
|  |  | The NAND gate is dead when you ground one input: ***DEAD*** |
|  |  | Tying one input of a NAND gate high inverts the other input: ***INVERTER*** |
|  |  | Tying one input of an OR gate low prevents the input from effecting the output: ***BUFFER*** |
|  |  | Tying an OR gate input to‘1’ kills the gate by forcing the output to always be ‘1’: ***DEAD*** |
|  |  | Tying one input of a NOR gate to ‘0’ outputs an inversion of the other signal: ***INVERTER***. |
|  |  | Tying a NOR gate input to ‘1’ kills the gate; the output is always low: ***DEAD*** |
|  |  | Tying one input to ‘0’ passes the other input to the output: ***BUFFER*** |
|  |  | Tying one input to ‘1’ outputs an inversion of the other input: ***INVERTER*** |

**Figure 10.9: Everything you didn’t want to know about the secret lives of logic gates.**

**Example 10.1: Half Adder using New Gate Types**

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**: This problem drops a giant hint that you should use a new gate in the solution. Figure 10.10 should help you recall that the HA has two inputs and two outputs. We all remember how a HA works but Figure 10.11 provides the truth table while Equation 10.1 shows the final un-reduced equations.

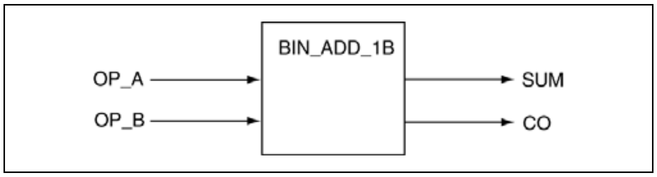


**Figure 10.10: The top-level BBD for a Half Adder (HA).**

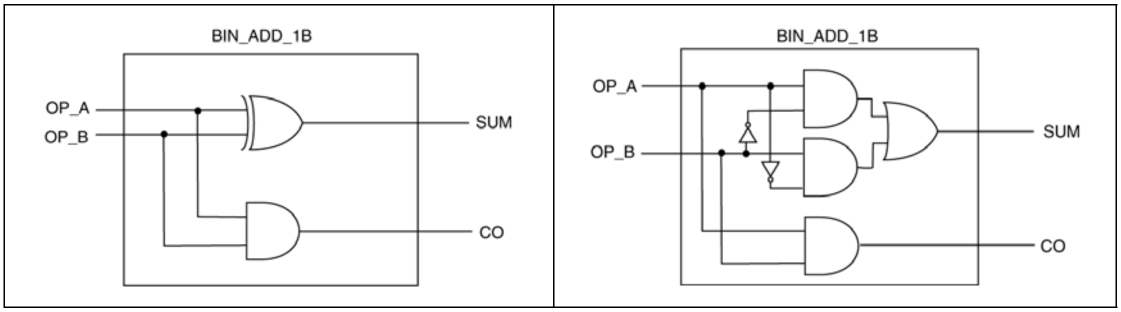
| OP_A | OP_B | SUM | CO |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

**Figure 10.11: Truth table for the HA.**

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

**Equation 10.1: The final equations for a HA.**

From inspection of Equation 10.1, the SUM output is an XOR function and the CO is an AND function. You need to inspect equations quite often in digital design as no one delivers items such as XOR functions on flaming pies[8]. Figure 10.12(a) shows the resulting circuit. Figure 10.12(b) shows the final circuit from the first time we did this problem. The result is two devices for the XOR enabled HA compared to six devices for the original version. The world is saved.



(a) (b)

**Figure 10.12: The HA using the newer gates (a) and older gates (a).**

[8] For you pop music fans out there, this is a historical reference.

**Example 10.2: Extracting XOR Functions from Boolean Equation**

Show that the following equations contain XOR functions.

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

**Solution**: There is no easy way to do this problem; you need to stare at it for a while and then see what XOR-type functions you can factor out. Factoring using Boolean algebra is something none of us wants to do, but sometime we must do it. Here we go.

The starting point we're looking for is a something that we can factor. This problem happens to be set up nicely in that the natural ordering of the terms makes the problem easier. Without too much description,

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

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

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

Figure 10.13 shows the final solution. Including the XOR function significantly reduced the amount of logic in the final Boolean equation. My apologies for failing to find a proper XOR operator in the text editing software.

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

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

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

**Figure 10.13: The factoring of the equation to extract an XOR function.**

**Example 10.3: A RCA with Extra Functionality**

Design a circuit that adds two 4-bit digital values. If the addition operation generates a carry-out, the 4-bit sum output is all zeros; otherwise, the 4-bit output shows the sum of the two 4-bit input values. Also, state what controls the circuit.

**Solution**: The first step in this solution is to draw a BBD; Figure 10.15 shows the BBD for this problem.

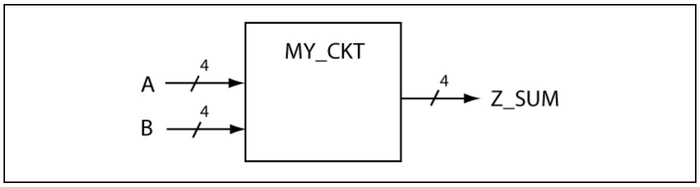


**Figure 10.14: Black box diagram for this problem.**

The next step is to discern if this circuit requires control and what that control needs to be. The problem implicitly states that this circuit requires control because the circuit output can be one of two values: all zeros or the summation of the circuit inputs. What is going to make this decision? Once again, the answer is in the problem statement. If the result of the addition generates a carry-out, we design the circuit to output to be all zeros; otherwise, the circuit outputs the summation. This means that an internal signal provides the control for this problem; that signal must be the carry-out signal. We use an RCA in this problem, as the RCA adds two numbers and provides a carry-out.

We know this circuit includes an RCA. However, to obtain the proper output of the circuit, there must be some other circuitry involved. We don’t know what exactly that entails right now, but we know enough to draw a BBD at a lower level that the top-level BBD. Figure 10.15 shows the result of our thought process thus far. Figure 10.15 shows that we plopped down a BBD and labeled it CKT; we did this as a placeholder, as we still don’t know what’s in the CKT box.

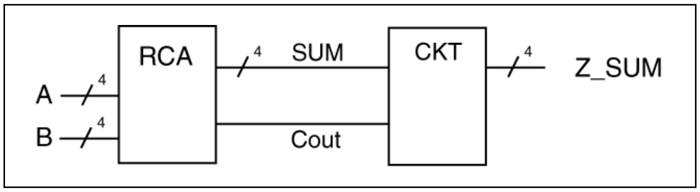


**Figure 10.15: A lower-level black box diagram for this problem.**

Now we need to think about the requirements of the black box labeled “CKT”. What this module must 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’; otherwise clear all of the sum bits. This operation describes the classic switch action, under control of the carry-out. There is a gate that implements such an operation: the AND gate.

Figure 10.16 shows the final solution for this problem using AND gates. We needed to first invert the carry-out signal in order for it to have the correct effect on the associated AND gates. The method we use to connect the AND gates ensures that their output is ‘0’ when the carry-out signal is a ‘1’ is to invert the carry-out before inputting it to the AND gates. We indicate the expansion of the SUM bundle by using parenthetical notation on the signal contained in the bundle.

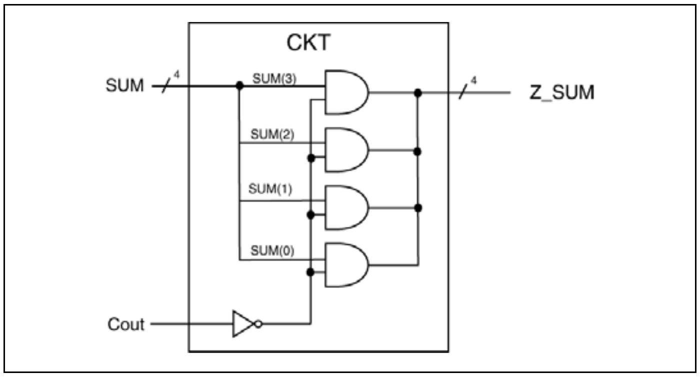


**Figure 10.16: Schematic diagram for the box labeled CKT.**

This problem is significant for one important reason: it's the first problem that we've worked with that has some type of a "control" feature. The problem uses the Cout from the RCA as a control input to the CKT block so that the state of the Cout output from the RCA controls what the final output of the circuit. We refer to this control as "internal control", which is one of four approaches to controlling a digital circuit.

## 10.5 Chapter Summary

- NAND and NOR are formed from complimenting the outputs of the AND & OR gates, respectively. NAND and NOR gates are generally used more often that 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 an AND function, an OR function, or an inversion function. AND & OR gates, however, are not functionally complete.
- We can connect basic logic gates to work as inverters, switches, and buffers. These connections represent an extended functionality of basic gates and are quite useful in digital design.

## 10.6 Chapter Exercises

**1)** Briefly describe why XOR and XNOR gates can only have two inputs.

**2)** Briefly describe why AND & OR gates are not considered functionally complete

**3)** Briefly describe whether you feel XOR gates are considered functionally complete?

**4)** Explicitly describe how to make a NOR gate into an inverter. Explicitly show the inverter functionality in the NOR gate truth table.

**5)** Explicitly describe how to make a NAND gate into an inverter. Explicitly show the inverter functionality in the NAND gate truth table.

**6)** Draw a diagram of a 4-input NAND gate that has been configured as an inverter. Don’t combine inputs for this problem.

**7)** Draw a diagram of a 4-input NOR gate that has been configured as an inverter. Don’t combine inputs for this problem.

**8)** What extended functionality can be obtained from a XNOR gate by connecting one input to either ‘1’ or ‘0’? Briefly explain.

**9)** What extended functionality can be obtained from a XOR gate by connecting one input to either ‘1’ or ‘0’? Briefly explain.

**10)** 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.

a) $F(A,B,C) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + AB\bar{C} + ABC + \bar{A}B\bar{C}$

b) $F(A,B,C) = ABC + \bar{A}BC + A\bar{B}C$

c) $F(A,B,C) = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C$

**11)** Why are there 3-input AND gates but no 3-input XOR gates? Briefly describe why.

## 10.7 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**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 allows only one person into the room when the controller you're designing unlocks the lock on that door. For this circuit, the door remains locked under the following conditions:

- When one person wants into each door
- When no people want in any door
- When one person wants in Door B but no one wants in any other door
- 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 your 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 two and 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.

# 11 Circuit Forms

## 11.1 Introduction

There are many functionally equivalent ways to represent Boolean expressions.. The underlying notion of being able to represent a function in various forms is that one form may have an advantage over other forms. If you can find a functionally equivalent form that you can implement faster, requires less power, is cheaper, etc.[1], then you're most likely going to use that form.

### Main Chapter Topics

**CIRCUIT FORMS:** Previous chapters 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 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 you should use one representation over another. This chapter outlines minimum concepts as they apply to function representations.

### Chapter Acquired Skills

- Be able to generate the eight standard circuit forms from a given Boolean equation
- Be able to find a minimum cost circuit from the set of eight standard circuit forms

## 11.2 Circuit Forms

The term "circuit forms" is a common term in digital logic design. This term generally refers to the fact that you can implement any given digital logic function using physically different but functionally equivalent circuits. In digital systems, 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 circuit form over another; we usually base the more desired form on the notion of "efficiency" of the implemented circuit. The definition of efficiency is a digital circuit is not absolutely definable; we typically base the definition of efficiency on circuit characteristics such as fewer gates, fewer inputs, fewer IC etc. than another. This section discusses forms that we can generate with successive applications of DeMorgan's theorem. This approach is somewhat standard and generates the most commonly seen circuit forms. In reality, there are only about four common circuit forms.

### 11.2.1 The Standard Circuit Forms

We use the term "standard circuit forms" to refer to eight circuit forms that we can easily derive using DeMorgan's theorem. If you examine other digital design textbooks, you'll find that they list bunches of

[1] And also many other reasons not listed here; hopefully you're getting the idea.

strange circuit forms; we opt to stick to the "standard" eight types, which you generate from successive applications of DeMorgan's theorem.

Equations 1(a) and 2(a) of Table 11.1 show the compact minterm and compact maxterm forms of an arbitrary function, respectively. A reduced version of these equations appears in 1(b) and 2(b). The resulting 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 11.2 provides 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)** $\overline{F} = \overline{A}C + A\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}$<br>**2(c)** $\overline{\overline{F}} = \overline{\left(\overline{A}C + A\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{D}\right)}$<br>**2(d)** $F = \overline{\left(\overline{A}C\right)} \cdot \overline{\left(A\overline{C}\,\overline{D}\right)} \cdot \overline{\left(\overline{A}\,\overline{B}\,\overline{D}\right)}$<br>**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)** $\overline{\overline{F}} = \overline{\overline{AC + \overline{C}D + \overline{A}B\overline{C}}}$<br>**1(d)** $F = \overline{\overline{(AC)} \cdot \overline{(\overline{C}D)} \cdot \overline{(\overline{A}B\overline{C})}}$ | **2(f)** $\overline{\overline{F}} = \overline{\overline{\left(A+\overline{C}\right)\cdot\left(\overline{A}+C+D\right)\cdot\left(A+B+D\right)}}$<br>**2(g)** $F = \overline{\overline{\left(A+\overline{C}\right)} + \overline{\left(\overline{A}+C+D\right)} + \overline{\left(A+B+D\right)}}$ |
| **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 11.1: 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, which we refer to as the Sum of Products (SOP) form. We obtain this form by writing a product term for every ‘1’ in the truth table modeling the given circuit. The final function represents a logical summing of the associated product terms. | We obtain the form in 2(b) writing a product term for every ‘0’ in the truth table, which gives us an expression for the complement of the function (**!F**). The expression is in AND/OR form, but we massage it into a different form by writing an expression for **F** rather than **!F** in 2(b) by complementing both sides of the expressions, in equation in 2(c). Dropping the double complement 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 the Product of Sums (POS) form. |
| **NAND/NAND Form** | **NOR/NOR Form** |
| We obtain the NAND/NAND form in 1(c) from the AND/OR form by double-complementing both sides of the equation in 1(b). The double complement on the left side of the equation 1(c) drops out. One of the overbars on the right side of equation 1(c) DeMorganizes the expression. The equation in 1(d) shows the NAND/NAND form of the expression, which refers to each of the individual product terms have overbars (a NAND function). These individual terms are ANDed together and complemented which effectively changes it from an AND function to an NAND function. | We obtain the form in 2(f) from the OR/AND form by double complementing both sides of the equation in 2(e). The double complement on the left side of the equation 2(f) drops out. On the right side of equation 2(f), we use one of the complements to DeMorganize the expression. The equation in 2(g) shows the NOR/NOR form of the expression. We refer to this as NOR/NOR form because each of the individual sum terms have overbars (a NOR function). These NOR functions are ORed together and complemented which changes it to a NOR function. |
| **OR/NAND Form** | **AND/NOR Form** |
| We obtain the OR/NAND form in 1(e) by DeMorganizing the individual terms in 1(d) to change them from product terms to sum terms. The expression retains the overbar over the entire term. | We obtain the AND/NOR form in 2(h) by DeMorganizing the individual terms 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** |
| We obtain the NOR/OR form in 1(f) by DeMorganizing the entire OR/NAND form of 1(e), which distributes the overbar on the right side of the equals sign to the individual terms in the equation. | We obtain the NAND/AND form in 2(i) by DeMorganizing the AND/NOR form in 2(h), which distributes the overbar on the right side of the equals sign to the individual terms in the equation. |

**Table 11.2: Written description of the circuit forms and derivations in Table 11.1.**

## 11.3 Minimum Cost Concepts

The best approach to implement circuits is implementing them at a minimum cost. This is an open-ended concept because minimum cost approach requires a proper definition of the word "minimum". There are about a bajillion definitions of the word "minimum" in terms of implementing a circuit. For digital design courses, this definition usually refers to the number of devices (gates and inverters) in the implemented circuit. "Minimum" can also mean the number of integrated circuits (ICs) you use in your circuit[2], or the number of transistors you use in the ICs in the circuit, etc. The fact you're your company may already have a bajillion ICs on hand that you can use further obscures the definition of minimum cost, as it would be cheaper to use them for your circuit. 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 a definition of "minimum".

Up to this point, you've learned to implemented functions with many different forms. 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 need to check all those forms also[3]. Unless given other specific directions, the form that uses the least amount of gates is generally the minimum cost solution.

**Example 11.1: 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 we previously did most of the work of the grunt work associated with this problem. Going back and examining Table 11.1, you'll be able to generate the information in Table 11.3; it has all the info we need if we know where to look.

From Table 11.3, 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 11.3: The whole enchilada for Example 11.1.**

[2] There are many ICs out there containing different flavors of standard gates such as AND, OR, NAND gates, etc.

[3] 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 to build 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.

## 11.4 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.

## 11.5 Chapter Exercises

**1)** Write the eight standard forms associated for the following function:

a) $F(A,B,C) = \sum(0,1,4,6)$

b) Draw the circuit for the NAND/NAND & NOR/NOR forms using inverters where necessary.

**2)** Show all four AND/OR related forms of the following equation: $F(A,B,C,D) = A\bar{B}C + \bar{B}D + \bar{A}BD$

**3)** Show all four OR/AND related forms for the following equation:

$$F(A,B,C,D) = (B + C + \bar{D})(\bar{A} + \bar{C})(A + \bar{B} + \bar{D})$$

**4)** Show all four AND/OR related forms for the following equation:

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

**5)** Show all four AND/OR related forms for the following equation:

$$F(A,B,C,D) = \overline{(\bar{B}D)}\,\overline{(A\bar{D})}\,\overline{(BC\bar{D})}$$

**6)** Draw a circuit for the following equations using only NAND gates and inverters.

a) $F(X,Y,Z) = XY + X\bar{Y}Z + \bar{Y}Z$

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

c) $F(A,B,C) = AB\bar{C} + A\bar{B}C + \bar{A}BC$

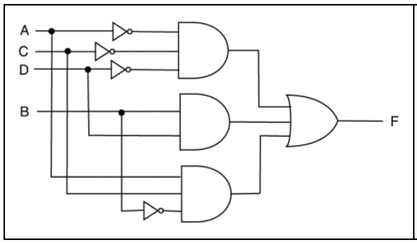
**7)** Draw a circuit for the following equations using only NOR gates and inverters:

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

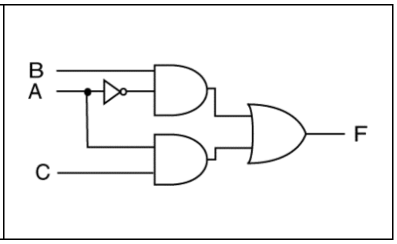
b) $F(X,Y,Z) = (X + Y + \bar{Z})(X + \bar{Z})(Y + \bar{Z})$

c) $F(A,B,C) = (\bar{A})(A + B)(A + \bar{B} + \bar{C})$

**8)** Show the four standard AND/OR-type Boolean equation forms for the following circuits.

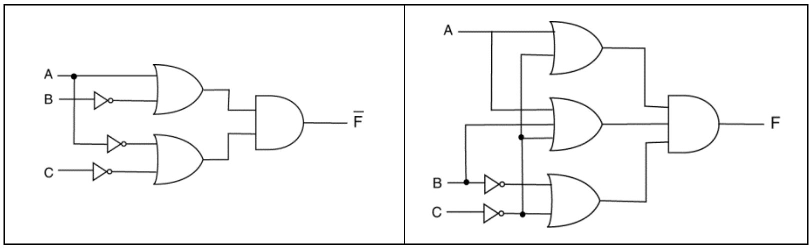


**(a)**



**(b)**

**9)** Show the four standard OR/AND Boolean equation forms for the following circuits.



**(a)** **(b)**

## 11.6 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**1)** Design a circuit that indicates special conditions on a 4-bit input. Consider the 4-bit input to be a binary number. This circuit has two outputs. One output indicates when the input is an even multiple of four and greater than zero. The other output indicates when the input is greater than 2 and less than 11. Design this circuit any way you deem appropriate.

**a)** Use nothing other than NOR gates and inverters in your final circuit

**b)** Use nothing other than NAND gates and inverters in your final circuit.

# 12 Signed Binary Representations

## 12.1 Introduction

The binary numbers we've worked with up until now have all been unsigned representations. This chapter presents three methods for representing signed binary numbers, which then allows us to start designing complex circuits that implement meaningful mathematical operations.

**Main Chapter Topics**

**BINARY NUMBER REPRESENTATIONS:** This chapter presents common representations of signed binary number including sign magnitude, radix complement, and diminished radix complement.

**Why This Chapter is Important**

- Be able to change the sign of numbers in SM, DRC, and RC format.
- Be able to convert form numbers in one representation to numbers in other representations.
- Be able to describe the number ranges for a given number of bits for signed and unsigned binary numbers in SM, DRC, and RC formats

## 12.2 Signed Binary Number Representations

Computers can only represent numbers with ones and zeros, which means we must also represent negative numbers using ones and zeros as well. There is no problem when you're simply writing numbers on a piece of paper because all you do is drop a "-" in front of the number and everyone agrees the number is negative. Computers don't have an easy and efficient way to use a "-" sign to represent negative number. This section describes how to represent signed numbers using only the binary values.

There are a few standard ways to represent signed binary numbers. In particular, there are three representations of interest: sign magnitude (SM), diminished radix complement (DRC), and radix complement (RC). 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.

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, which is the left-most bit position of the number. The left-most-bit in every signed number representation in this text is the sign bit. If the sign bit is a '1', then we interpret the number as negative; if the sign bit is a '0', the number is positive. Figure 12.1 provides a visual representation of the bit positions of the sign and magnitude bits.

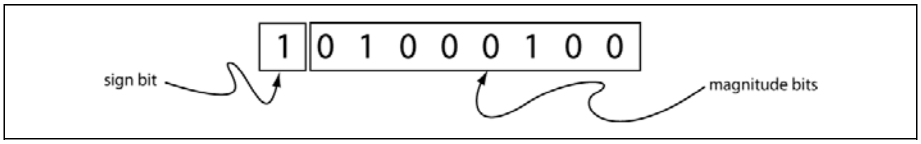


**Figure 12.1: Some generic nine-bit number that we interpret as being a signed value.**

### 12.2.1 Sign Magnitude Notation (SM):

In SM notation, the sign bit indicates the sign of the number, while the other bits represent the magnitude of the number. Table 12.1 lists everything you need to know about tweaking SM numbers.

| 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<br>2) do binary to decimal conversion on magnitude bits<br>3) add in minus sign (from step 1) |

**Table 12.1: Standard operations on binary numbers represented in SM.**

**Example 12.1: Changing the Sign of Numbers in SM Form**

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. 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 12.2: Converting Numbers in SM Form to Decimal**

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. We convert the number 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.

### 12.2.2 Diminished Radix Complement (DRC)

We can best explain the DRC representation by the operations required to change the sign of the number, which only requires that we toggle all the bits in the binary number (which we refer 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 magnitudes differently). Table 12.2 lists everything you need 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<br>2) toggle all the bits (1's complement)<br>3) do binary to decimal conversion on magnitude bits<br>4) add in minus sign (from step 1) |

**Table 12.2: Standard operations on binary numbers represented in DRC.**

**Example 12.3: Changing the Sign of Numbers in DRC Format**

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 12.4: Converting Numbers in DRC Format to Decimal**

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. We can convert to decimal directly using standard binary-to-decimal conversion techniques since the sign bit is zero and adds 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) $110011_2$ → $001100_2$
3) $001100_2$ represents 12 in decimal
4) Adding the negative sign completes the solution: -12

### 12.2.3 Radix Complement (RC):

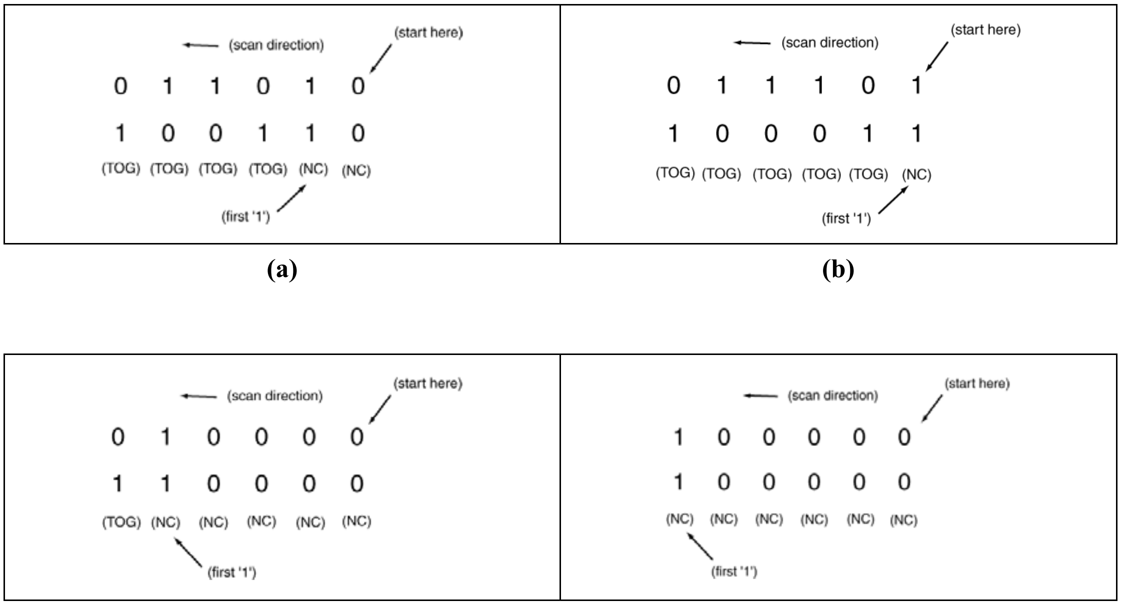
We can best explain RC representations by the operations required to toggle the sign of the number. In RC notation, the sign bit indicates the sign of the number, but it has a unique way of also being part of the magnitude for negative numbers. We once again interpret the magnitude bits differently for positive and negative numbers. For positive numbers, we interpret the magnitude bits directly as a simple binary number. If the number is negative, we consider the magnitude bits to be in a two's complement representation. Table 12.3 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<br>2) take the two's complement of the number<br>3) do binary to decimal conversion on magnitude bits<br>4) add in minus sign (from step 1) |

**Table 12.3: 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. We define the two's complement 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.

The easiest way to find the 2's complement of a number is to apply the following algorithm: 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). Figure 12.2 shows just about every case you'll ever hope to run across. In Figure 12.2, NC stands for "no change" while TOG stands for "toggle".



**(a)** **(b)**

**(c)** **(d)**

**Figure 12.2: Four examples showing the 2's complement conversion algorithm.**

**Example 12.5: Changing the Sign of Numbers in RC Format**

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 12.6: Converting Numbers in RC Format to Decimal**

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. We can convert to decimal directly using standard binary to decimal conversion techniques since the sign bit is zero and adds 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) 10011012 → 01100112
3) 01100112 represents 51 in decimal
4) Adding the negative sign completes the solution: -51

## 12.3 Number Ranges in SM, DRC, and RC Notations

Representing sign numbers in binary requires that we use an extra bit (the sign bit) to represent the sign. It seems that if we use one less bit to represent the magnitude of the number, we can only represent one-half as many numbers by the same amount of bits[1]. This is not the case. The reality is that, generally speaking, the ranges of numbers that are representable with an unsigned binary number shift downwards when representing signed numbers. 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 roughly centered about zero. Figure 12.3 shows what the last few sentences are attempting to convey.

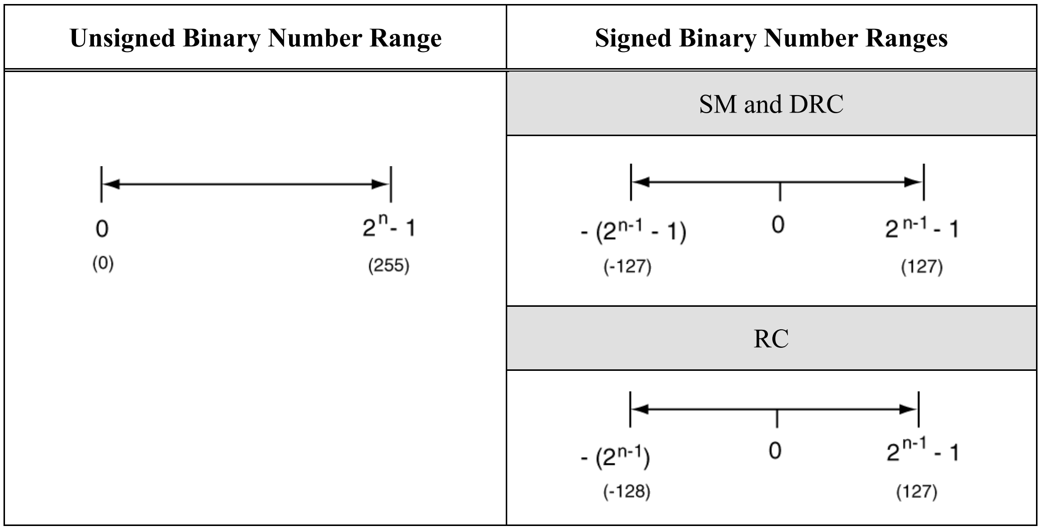


**Figure 12.3: Number ranges for signed and unsigned binary numbers (n=8).**

The key to understanding Figure 12.3 is that the letter n represents the number of bits in the binary number. The smaller numbers in parenthesis in Figure 12.3 shows the number ranges when n=8. Note in Figure 12.3 that with SM and DRC representations, we can only represent $2^n$-1 out of the $2^n$ possible values for a given value of n. However, with RC, we can represent all $2^n$ possible values. This is a major reason why computers commonly use RC for signed binary number representations, as having two values representing zero is challenging for the hardware performing mathematical operations on those numbers.

Twos complement math is an area in digital design that just about everyone is weak in. People generally 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.

[1] 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?

## 12.4 Extending Data Widths

You'll often times find that your design must change data width of a number without changing the value of the number. The most common of these changes is when you need to extend the width of the data without changing the numeric value of that data. You've probably done this many time using decimal numbers, where you add as many 0's to the front of the number (the digits with the largest weights).

### 12.4.1 Unsigned Binary

Extending the bit-width with unsigned binary numbers is the most straightforward as it is similar to decimal numbers. It is straightforward because we don't need to deal with the sign bit. For unsigned binary number we simply add as many zeros as we need to the number to attain the desired width. The numbers we add become the most significant digits of the number, meaning we add the zeros to the left side of the existing bits. We refer to this form of bit stuffing as "zero-extending", or "zero-stuffing", or simply "bit stuffing". Table 12.4 provides a few examples of extending the bit-width of unsigned number from four to eight bits.

| | Unsigned Binary | |
|---|---|---|
| **Decimal** | **(4-bit)** | **(8-bit)** |
| 9 | 1001 | **0000**1001 |
| 3 | 0011 | **0000**0011 |
| 15 | 1111 | **0000**1111 |
| 1 | 0001 | **0000**0001 |

**Table 12.4: Examples of extending bit-widths of unsigned binary numbers.**

### 12.4.2 Signed Binary (RC Form)

Extending the bit-width of signed numbers is slightly more involved than the same action with unsigned number. We only consider signed numbers in radix complement format (RC) for this discussion.

The main issue when dealing with signed numbers is working with the sign bit. It seems natural that zero-extending the any value can't possibly change that value, but it can when dealing with signed numbers. For example, if we zero-extend a negative number, the sign bit of the smaller bit-width is no longer the left-most bit; the new sign bit is '0', which makes the number positive, which is clearly not what we want. The solution when working with signed number is to "sign-extend" the number. Sign extension means that all the extra bits we add to the number need to be the same value as the sign bit. In short, we bit-stuff the number with 1's if the smaller width number is negative; otherwise we bit-stuff it with 0's. Table 12.5 provides a few examples of extending the bit-width of signed binary numbers (RC format) from four to eight bits, which we refer to as sign extension.

| Decimal | Signed Binary (RC) | |
|---|---|---|
| | 4-bits | 8-bits |
| -7 | 1001 | **1111**1001 |
| 3 | 0011 | **0000**0011 |
| -1 | 1111 | **1111**1111 |
| -8 | 1000 | **1111**1000 |
| 1 | 0001 | **0000**0001 |

**Table 12.5: Examples of extending bit-widths of signed binary numbers (RC format).**

## 12.5 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 those methods can represent. Although the different number formats can represent roughly the same quantity of unique number, signed numbers are typically centered about zero, while unsigned numbers start at zero.
- Extending the bit-widths of unsigned and signed binary numbers is different. Typically, unsigned numbers can be bit-stuffed with zeros without changing the value of the number (zero-extended). Signed numbers must take into account the sign bit. Specifically, signed number in RC format are signed extended when the number requires an increase in bit-width.

## 12.6 Chapter Exercises

**1)** Complete the following table:

| # bits | unsigned binary range | signed binary range (RC) |
|---|---|---|
| 4 | | |
| 6 | | |
| 8 | | |
| 10 | | |
| 11 | | |
| 12 | | |
| 14 | | |
| 15 | | |
| 16 | | |

**2)** Which of the following two signed binary (SB) numbers have a greater magnitude? Assume the numbers are given in radix complement (RC) form.

**a)** 1110 1110 0000 0010

**b)** 1000 1101 0111 0111

**c)** 1110 1110 0001 0011

**3)** Which of the following three SB numbers has the largest magnitude?

**a)** 1110 0001(SM), 1001 1101 (DRC), 1001 1100 (RC)

**b)** 1001 1110 (SM), 1000 1101(DRC), 1001 1111(RC)

**5)** 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), CC(DRC), D1(RC)

**b)** F3(SM), EC(DRC), DD(RC)

**6)** Write the decimal equivalents of the following numbers for SM, DRC, and RC formats

a) $BC_{16}$

b) $4A_{16}$

c) $D2_{16}$

**7)** Extend the bit-widths of the following unsigned binary values from 8 to 12-bits. Write the answers as binary values.

a) $A7_{16}$

b) $4A_{16}$

c) $C4_{16}$

**8)** Extend the bit-widths of the following unsigned binary values from 8 to 16-bits. Write the answers in hexadecimal format.

a) $AF_{16}$

b) $4A_{16}$

c) $C4_{16}$

**9)** Extend the bit-widths of the following signed binary values (RC format) from 8 to 12-bits. Write the answers as binary values.

a) $A7_{16}$

b) $4A_{16}$

c) $C4_{16}$

d) $02_{16}$

**10)** Extend the bit-widths of the following signed binary values (RC format) from 8 to 16-bits. Write the answers in hexadecimal format.

a) 0xDE

b) 0x3F

c) 0xC4

d) 0x99

## 12.7 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**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 that inputs an 8-bit signed binary number in RC format. If the input is positive, the circuit outputs the input number; otherwise, the circuit outputs all zeros.

**5)** Design a circuit that inputs an 8-bit signed binary number in RC format. If the input is positive, the outputs a negative version of the input; otherwise the circuit outputs all zeros.

# 13 Binary Mathematics

## 13.1 Introduction

Now that you've seen various number systems and various manipulations of numbers in various radii, we can now start of doing basic math with binary numbers. This introduction provides the background you can use for designing circuits that performs math operations.

**Main Chapter Topics**

**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.

**Chapter Acquired Skills**

- Be able to perform addition and subtraction with signed binary numbers in RC format.
- Be able to determine the validity of results when performing addition and subtraction on numbers in RC format.

## 13.2 Binary Addition and Subtraction

The topic of binary arithmetic and computers is a deep subject that many people spend their entire lives studying. If you design a computer that performs efficient mathematical operations, you'll have a good computer. The problem is that there are a bunch of trade-offs along the way; you'll run into some of these topics later in your digital/computer education but they're beyond the scope of this discussion. We limit this discussion to the addition and subtraction of signed and unsigned binary numbers.

Recall that digital circuits comprise of a fixed set of hardware. What this means is that we generally perform arithmetic operations with fixed sized circuits (fixed data widths). For example, a 12-bit RCA will have trouble adding two numbers of 14 bytes each.

The ramification of a fixed hardware size is that your mathematical operations must stay within these limits in order for the result to be valid. If you stay within these limits, your result is valid; if you exceed these limits, you're result is invalid. The crux of this discussion is that you need 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.

### 13.2.1 Binary Subtraction

One of the many recurring themes in digital design is that you always want to design your circuits to do what they need to while as little hardware as possible. Mathematical operations in computers do not come free: the underlying hardware performs the operations. Hardware, or digital circuitry, requires up space, consumes power, and makes your design more complex as you use more of it.

Design factors such as power consumption and circuit real estate play out directly in this discussion in the context of binary subtraction. Although we could design a circuit that performs subtraction, the better approach is using a circuit we already designed to perform subtraction. The approach we take is to use our RCA to apply indirect subtraction by addition. Equation 13.1: shows the basic formula for this approach.

$$\boxed{N1 - N2 = N1 + (-N2)}$$

**Equation 13.1: Indirect subtraction by addition.**

Changing the sign of a number is straightforward when dealing with RC numbers: you take the two's complement. In order to subtract one binary number from another you must first take the two's complement of that number being subtracted and add it to the other number (as Equation 13.1 says). After this addition operation, you need to examine some signals in order to determine if the result is valid or not, as the result may exceed the given number determined by the hardware.

Consider adding two numbers **A** and **B** with a result **C**: **A** + **B** = **C**. We refer to variable A as the augend, B as the addend, and C as the sum. Consider subtracting one number B from another number A with a result C. In case, we refer to A as the minuend, B as the subtrahend, and C as the difference. This knowledge could be valuable if you were to find yourself on Jeopardy but it does not get a lot of mileage outside of this discussion.

### 13.2.2 Addition and Subtraction on Unsigned Binary Numbers

The results of your mathematical operation can either underflow or overflow the given number range when working with unsigned binary numbers. Underflow would be the result of subtracting a 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[1]. Table 13.1 and Table 13.2 list everything you need to know about the overflow and underflow of binary numbers.

<table>
<tr><th colspan="4">Overflow in Unsigned Binary Addition</th></tr>
<tr><td>Description</td><td colspan="3">The sum of two binary numbers exceeds the range associated with the data width</td></tr>
<tr><td>Indicator</td><td colspan="3">The carry-out from the MSB addition is ‘1’.</td></tr>
<tr><td>Example 13.1</td><td>1001 + 0011 = ?</td><td><table><tr><td></td><td>1001</td></tr><tr><td>+</td><td>0011</td></tr><tr><td>0</td><td>1100</td></tr></table></td><td>The carry from the MSB is 0, which indicates there was no carry. The sum (the four-bit result) is a valid.</td></tr>
<tr><td>Example 13.2</td><td>1011 + 0111 = ?</td><td><table><tr><td></td><td>1011</td></tr><tr><td>+</td><td>0111</td></tr><tr><td>1</td><td>0010</td></tr></table></td><td>The carry out of the MSB is 1, which indicates there was a carry. Therefore, the sum (the four-bit result) is not valid.</td></tr>
</table>

**Table 13.1: The low-down on unsigned overflow.**

[1] An issue here is that we often use “overflow” to describe both underflow and overflow. The notion here is that you can exceed, or “overflow”, the given range in either direction.

<table>
<tr><th colspan="4">Underflow in Unsigned Binary Subtraction</th></tr>
<tr><td>Description</td><td colspan="3">The difference of between two binary numbers is below the range associated with the data width</td></tr>
<tr><td>Indicator</td><td colspan="3">The carry-out from the MSB addition is ‘0’.</td></tr>
<tr><td>Example 13.3</td><td>1001 - 0011 = ?</td><td>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></td><td>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.</td></tr>
<tr><td>Example 13.4</td><td>0111 - 1100 = ?</td><td>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></td><td>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.</td></tr>
</table>

**Table 13.2: The low-down on unsigned underflow.**

### 13.2.3 Addition and Subtraction on Signed Binary Numbers

The results of your mathematical operations on signed binary numbers can either underflow and overflow the given number range. The approach to dealing with operations on signed binary number is more intuitive than dealing with unsigned binary numbers. The list below describes the two main concepts.

- Overflow can never occur if you’re adding a positive number to a negative number; the result from the operation A - B is always valid if both A and B are positive numbers or both negative numbers. Therefore, if the two numbers have different sign bits before the addition[2], the answer is always valid.
- Overflow and underflow only occurs when you add to numbers that have equivalent sign bits but the result has a sign bit of a different value. Overflow and underflow can only happen in two scenarios:
    - Overflow: Adding a positive number to a positive number. However, due to the indirect subtraction by addition, this can include subtracting a negative number from a positive number.
    - 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.

[2] Keeping in mind that either we can 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 range associated with the bit width. | | |
| 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). We never consider the carry-out. | | |
| Example 13.5 | 0011 + 0010 = ? | 0011<br>+ 0010<br>0 0101 | The sign of addend and augend are positive and the sign of result is positive. The result is valid. |
| Example 13.6 | 0100 + 1110 = ? | 0100<br>+ 1110<br>1 0010 | The sign of addend and augend are different so there can be no overflow or underflow. The result is valid. |
| Example 13.7 | 0110 + 0101 = ? | 0110<br>+ 0101<br>0 1011 | The sign of the addend and augend are the same but are different from the sign of the result. The result is not valid. |
| Example 13.8 | 0100 - 1110 = ? | add the negation of 1110<br>0100<br>+ 0010<br>0 0110 | The sign of addend and augend are positive and the sign of result is positive. The result is valid. |
| Example 13.9 | 0100 - 0011 = ? | add the negation of 0011<br>0100<br>+ 1101<br>1 0001 | The sign of addend and augend are different so there can be no overflow or underflow. The result is valid. |
| Example 13.10 | 0100 - 1100 = ? | add the negation of 1100<br>0100<br>+ 0100<br>0 1000 | The sign of the addend and augend are the same but are different from the sign of the result. The result is not valid |

**Table 13.3: The low-down on overflow in signed binary numbers.**

<table>
<tr><th colspan="4">Underflow in Signed Binary Addition and Subtraction</th></tr>
<tr><td>Description</td><td colspan="3">The result of an operation between two binary numbers is below the range associated with the bit width.</td></tr>
<tr><td>Indicator</td><td colspan="3">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). We never consider the carry-out.</td></tr>
<tr><td>Example 13.11</td><td>1111 + 0010 = ?</td><td>  1111<br>+ 0010<br>1 0001</td><td>The sign of addend and augend are different so there can be no overflow or underflow. The result is valid.</td></tr>
<tr><td>Example 13.12</td><td>1110 + 1111 = ?</td><td>  1110<br>+ 1111<br>1 1101</td><td>The sign of the addend and augend are the same and match the sign of the result. The result is valid.</td></tr>
<tr><td>Example 13.13</td><td>1100 + 1001 = ?</td><td>  1100<br>+ 1001<br>1 0101</td><td>The sign of the addend and augend are the same but are different from the sign of the result. The result is not valid.</td></tr>
<tr><td>Example 13.14</td><td>1110 - 1111 = ?</td><td>add negation of 1111<br>  1110<br>+ 0001<br>0 1111</td><td>The sign of addend and augend are different so there can be no overflow or underflow. The result is valid.</td></tr>
<tr><td>Example 13.15</td><td>1100 - 0011 = ?</td><td>add negation of 0011<br>  1100<br>+ 1101<br>1 1001</td><td>The sign of the addend and augend are the same and match the sign of the result. The result is valid.</td></tr>
<tr><td>Example 13.16</td><td>1001 - 0110 = ?</td><td>add negation of 1101<br>  1001<br>+ 1010<br>1 0011</td><td>The sign of the addend and augend are the same but are different from the sign of the result. The result is not valid.</td></tr>
</table>

**Table 13.4: The low-down on underflow in signed binary numbers.**

## 13.3 Special Cases of Validity for RC Numbers

The validity check presented in the previous section unfortunately fails in one case, which is a known issue when working with addition and subtraction using numbers in RC format. The RCA we use in this text has two inputs: **A** & **B**. The RCA only adds number so it always adds the values of **A** & **B**. When we work with numbers in RC format, one or both of these two inputs can be negative.

The case that causes trouble is with when the **B** input is a negative number of the largest magnitude associated with the bit-width of **B**, and **B** is being subtracted from **A**. In this case, the validity check this chapter presented does not provide the correct answer. There are ways to modify the validity check to work in all cases, but we choose not to in this text in order to keep the conversation relatively simple. So as a result, this text never considers this case when checking validity of operations on RC numbers. In the end, creating the circuitry that makes the validity correct in all cases becomes a nice little design problem.

## 13.4 Chapter Summary

- We generally do all of our mathematical operations using radix complement (RC) notations of numbers. While you can do math operations with other number representations, RC format has some definite advantages.
- 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 a given set of bits can represent.
- We often perform binary subtraction by using addition. We refer to this technique 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).

## 13.5 Chapter Exercises

**1)** Briefly explain why adding two numbers of a different sign will always result in a valid number in terms of fixed hardware widths.

**2)** Briefly explain why “underflow” is sometimes classified as “overflow”.

**3)** Briefly explain the difference between the concept of overflow/underflow and the concept of carry-out.

**4)** Briefly explain what is meant by the notion of “fixed hardware widths”.

**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 + 000011

**b)** 001110 + 000111

**c)** 100101 + 101010

**d)** 001000 + 111100

**e)** 000100 + 101111

**7)** 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

**8)** 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

**9)** 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

**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)** 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)** Describe two different algorithms for finding the 2's complement of a signed binary number.

**12)** Consider the case where a Ripple Carry Adder is used to perform addition or subtraction on two n-bit signed binary numbers in radix complement form. Does the value of the carry-out affect the validity of the n-bit sum output of the RCA? Explain fully but briefly.

## 13.6 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**1)** Design a circuit that always does the following operation: A – B. Consider both inputs and the output to be 10-bit binary numbers in RC format. For this problem, assume the result will always be valid.

**2)** Repeat the previous problem but include an output named: VALID. The VALID output is a ‘1’ when the 10-bit result of the subtraction operation is correct (valid); if the result is not correct, the VALID output should be a ‘0’. Keep in mind that depending on the values of the two inputs, the result could exceed the range of the 10-bit output, in which case the value on the output is not correct. In other words, there is always a number on the output; we’re designing an extra output (the VALID signal) to indicate when the number on the result is correct or not.

# 14 Mixed Logic

## 14.1 Introduction

Good digital designers understand mixed logic. While you can go a long way by pretending you understand mixed logic, you'll be bummed out when you realize that you don't really understand it. It's highly unlikely that any digital system you work with only uses one type of logic, so you need to be able to design and/or interface digital circuits in a mixed logic environment.

The thing that stands out most about mixed logic is a comment my digital design instructor made: "nobody really understands mixed logic"[1]. What I've come to realize is that the reason that "nobody understands" this stuff is two-fold. First, I've never seen a textbook that explains the topic in a manner that I could understand. Secondly, it's a topic that that you can avoid understanding by learning a few tricks to deal with the topic when you need to. It's better to truly understand the topic.

**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 we later apply in both circuit design and circuit analysis problems.

**Why This Chapter is Important**

- Be able to describe the importance of being able to work in mixed logic systems.
- Be able to generate all alternative forms of AND, OR, NAND, and NOR gates
- Be able to analyze mixed logic circuits and generate Boolean equations describing those circuit
- Be able to design mixed logic circuits from given Boolean equations

## 14.2 Mixed Logic Overview

The underlying theme of all digital logic is the basic interpretation of signals. A signal in a digital circuit is either at a high or low voltage level[2]. 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 device in the circuit as well as the input to another device in the circuit.

Digital circuits are extraordinarily dumb: the gates in digital circuits have outputs that react to the inputs. Here's the whole story in a few sentences: the way we've been modeling our circuits so far is that a '1' represents the action state or active state of things while '0' represents the non-action state or inactive state. In other words, when the circuit's inputs represent a combination that we were interested in, we assign a '1' to the output; the '1' or the high state generally means something affirmative or positive occurred in the circuit. We refer to this as positive logic.

[1] Spoken by Dr. Marty Kaliski sometime in the late 1980's.

[2] We stay general here by not mentioning the exact voltage levels; we don't need to say anything other than high and low voltage. The notion here is that some external entity has pre-decided what the voltage levels are.

The notion of 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 with negative and/or positive logic, sometimes your design needs to interface with another circuit that is interpreting the 1’s in 0’s differently than your circuit. Thus, you’re facing not just coming up with a digital design, you’re facing coming up with a *mixed logic design*.

The following bullets represent most of the terminology associated with mixed logic. The definitions below are somewhat brief, but they start making sense when we use them in the various explanations in this chapter.

- **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.
- **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**: A way to refer to a positive logic signal
- **Asserted Low**: A way to 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’s in a high state; a negative logic signal is asserted when 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’s low; a negative logic signal not asserted when it’s high.

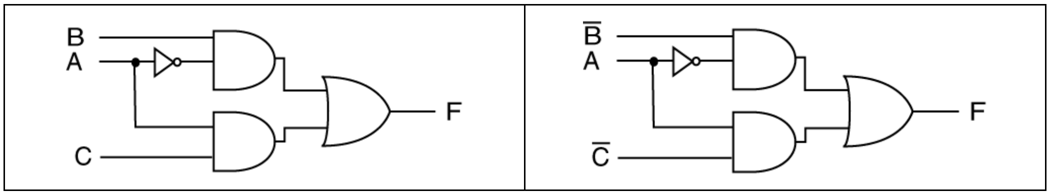
You first need to convince yourself that the circuits you’ve been working with thus far have all been positive logic circuits. Figure 14.1(a) shows a circuit that you’re used to working with. What you may not realize is that by the way the circuit appears in Figure 14.1(a), the inputs and the output of the circuit are all positive logic. 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 (‘0’ indicates a negative condition).

How exactly do we represent negative and positive logic in a circuit? There are two ways: the Positive Logic Convention (PLC) and Direct Polarity Indicators (DPI). For this discussion, we only use DPI since it is easier to work with while learning mixed logic. Once you understand the DPI convention, using either DPI or PLC (or both) won’t be a problem.

Up until now, you’ve only dealt with PLC. The PLC uses overbars on signals to indicate that they are negative logic (not having overbars represents positive logic). For example, the circuit in Figure 14.1(a) contains three input variables and one output variable. Since none of these variables has overbars on them, we interpret them as being positive logic. While it is tempting to interpret the overbars on the signal names as the state of the signal, the overbar (or lack thereof) refers to logic levels and says nothing regarding whether the signal is high or low. Recall that signals are variables: the can take on a value of ‘1’ or ‘0’.

Figure 14.1(b) shows an example of a similar circuit that uses mixed logic. Two of the circuit’s inputs contain overbars, which indicate that signal A is a positive logic while signals B and C are negative logic. All signals in both of these figures are Boolean variables, which means they can be either 1’s or 0’s.

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 in Figure 14.1 look similar, the point is that they perform different logic functions. What exact logic functions they perform is what we figure out in the remainder of chapter.



(a) (b)

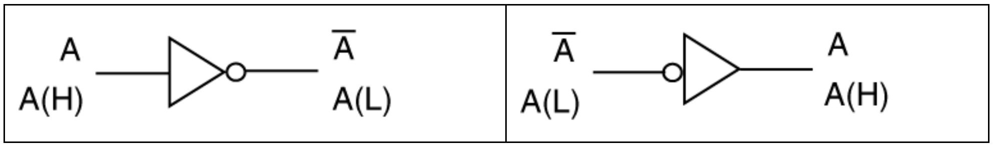
**Figure 14.1: Some similar looking but very different circuits.**

## 14.3 The Inverter and Mixed Logic

We think of inverters as devices that change 1's to 0's and 0's to 1's. While this is a valid interpretation of an inverter, we need to model them differently in order to gives us a foundation for understanding mixed logic. Figure 14.2 shows our new approach to modeling an inverter.

Figure 14.2(a) shows an inverter as you're used to seeing it. The thing that is new about this diagram is that we provide the PLC and DPI indicators above and below the signals, respectively. We use this notation to indicate that we're no longer thinking of the inverter as a device that toggles a signal value; we now view it as a device that 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).

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 14.2(a). With the DPI notation, we indicate the A signal with a directly polarity indicator of H (indicating positive logic) on the input of the inverter. Once the signal passes through the inverter, the direct polarity indicator changes to L (indicating negative logic).



(a) (b)

**Figure 14.2: A different approach to modeling an inverter.**

## 14.4 Equivalent Signals for DPI Notation

We need some tools to work with the mixed logic circuits. Our first tool is to rewrite a negative logic signal as a positive logic signal without officially "changing" the signal. Figure 14.3 shows the equivalent signals we use. The equations in Figure 14.3 show there is more than one way to represent negative and positive logic using the DPI convention. You can indicate a positive logic signal as an equivalent negative logic signal (Figure 14.3(a)) and you can write a negative logic signal as an equivalent positive logic signal (Figure 14.3(b)). These equations represent equivalent forms of the signals.

| $A(H) = \overline{A}(L)$ | $A(L) = \overline{A}(H)$ |
|---|---|
| (a) | (b) |

**Figure 14.3: Equivalent signals relating to inversion.**

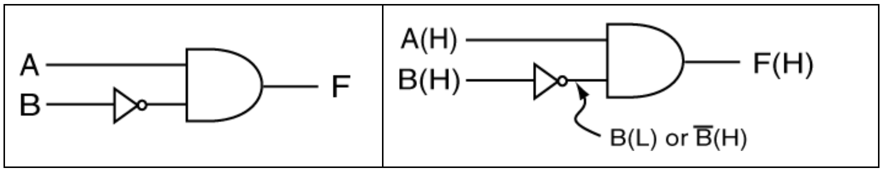
Now we apply the equivalent signals approach in a simple circuit. Figure 14.4(a) shows a two-input AND gate with an inverter in front of one of the inputs; we re-analyze it using mixed logic concepts. We attach a DPI convention to the inputs and outputs of this device; both inputs and the single output 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, we implement the (**A·!B**) logic expression.

Recall that an AND gate's output is a '1' when both inputs are '1'. The question that arises is this: what is the relation between the product term (**A·!B**) and the notion of 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.

The way it's drawn indicates that the AND gate expects to receive two positive logic inputs. The two inputs are both positive logic but the **B** signal goes through an inverter before it reaches the input of the AND gate. This means that that the **B** input is now a negative logic input when it is input to the AND gate; this presents an issue as the AND gate expects positive logic inputs in order to perform the AND function. .

The solution is to rewrite the logic level of the **B** signal using equivalent signals. Once we change the logic level of **B** from the original positive logic to negative logic, '0' is then be the active level of the signal; or to use our new terminology, the signal is active low. We need to rewrite the signal representation after it exits the inverter to make it "look" like positive logic. Figure 14.4(b) shows the logic levels of the signal after it passes through the inverter written in using an equivalent form from Figure 14.3. Once we have the newly labeled signal in place, we can write the equation for the circuit by inspecting the circuit. In official terms, the output of the gate is asserted when the both the A and B inputs are asserted.



**(a)** **(b)**

**Figure 14.4: A mixed logic approach to analyzing familiar functions.**

Figure 14.5 shows that there are two ways of writing the equation for the final circuit. Figure 14.5(a) shows the equations in DPI form while Figure 14.5(b) shows the equation in PLC form.

| | |
|---|---|
| $F(H) = A(H) \cdot \bar{B}(H)$ | $F = A \cdot \bar{B}$ |

**(a)** **(b)**

**Figure 14.5: Two resulting forms of our analyzed mixed-logic circuit.**

## 14.5 Mixed Logic-Based Gate Forms

Let's re-examine the logic gates we've dealt with up to this point. Using a strange mixture of mixed logic concepts and Boolean algebra, we generate alternative forms of these gates and then use these alternative forms in mixed logic problems.

The simplest approach to understanding mixed logic is to examine basic logic gates. Until now we implemented our gate-level designs using primarily AND, NAND, OR, NOR gates and inverters. Remember those bubbles on the outputs of the NAND and NOR gates (and inverters too)? They're somewhat 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.

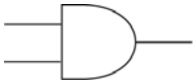
| | |
|---|---|
| <br>$F = A\cdot B$<br>$F(H) = A(H)\cdot B(H)$ | This AND provides an AND function with a positive logic output, which you know because you see that familiar AND gate shape. The AND gate performs an AND function on the two positive logic inputs. Since there are no bubbles on the back of the gate, this AND gate expects positive logic inputs. This is the AND form of an AND gate. |

**Figure 14.6: A mixed logic view of an AND gate.**

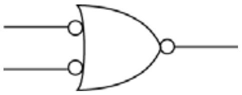
| | |
|---|---|
| <br>$F = A\cdot B$<br>$\overline{\overline{F}} = \overline{\overline{A\cdot B}}$<br>$F = \overline{\overline{A}+\overline{B}}$<br>$F(L) = A(L)+B(L)$ | This gate is an AND gate, because you can use DeMorgan's theorem to generate a different equation describing the gate. You derive the new gate form by double complementing the AND function equation and then DeMorganizing the resulting equation. The distinctive symbol results from the two equations on the left. The key to understanding this gate is to examine both the bubbles and the gate form. If you feed this gate two negative logic inputs, 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. You can thus use an AND gate to perform an OR function. We refer to this gate as the OR form of an AND gate. |

**Figure 14.7: A different mixed logic view of an AND gate**

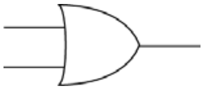
| | |
|---|---|
| <br>$F = A+B$<br>$F(H) = A(H)+B(H)$ | The OR gate provides a high output when either of the gates two inputs is high. If you provide the gate with positive logic inputs, it performs an OR function and generates a positive logic output. The equations on the left show this characteristic. Since there are no bubbles on the back of the gate, this OR gate expects positive logic inputs. Since there is no bubble on the gate output, this gate delivers a positive logic output. This gate is the OR form of an OR gate. |

**Figure 14.8: A mixed logic view of an OR gate.**

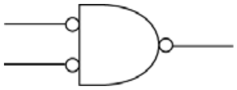
| | |
|---|---|
| <br>$F = A+B$<br>$\overline{\overline{F}} = \overline{\overline{A+B}}$<br>$F = \overline{\overline{A}\cdot\overline{B}}$<br>$F(L) = A(L)\cdot B(L)$ | The gate on the left is an OR gate. We derive this new gate form from double complementing the equation describing the OR function and DeMorganizing the resulting equation. We derive this distinctive symbol from the bottom two equations. This gates looks like an AND gate; 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. You can thus use an OR gate to perform an AND function. This gate is the AND form of an OR gate. |

**Figure 14.9: A different mixed logic view of OR gate.**

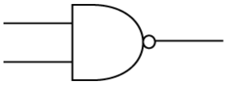
| | |
|---|---|
|  | You know the NAND gate as a AND gate with an inverted output. In a mixed logic sense, this gate performs an AND function on the positive logic inputs and provides a negative logic output. We consider the inputs to be positive logic due to the absence of bubbles on the inputs; the output is a negative logic output since there is a bubble on the output. One way to view this circuit is that the output of '0' is now the active state rather than the '1' output, which is the active state from a normal AND gate. This is the AND form of a NAND gate. |
| $F = \overline{A \cdot B}$<br>$F(L) = A(H) \cdot B(H)$ | |

**Figure 14.10: A mixed logic view of an NAND gate.**

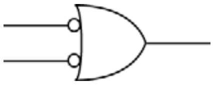
| | |
|---|---|
|  | This gate is a NAND gate. If we apply DeMorgan's theorem to the gate we arrive at a new equation describing the gate. The final two equations on the left describe this gate in the context of mixed logic: this gate performs an OR function on its two negative logic inputs and returns a positive logic output. Seeing the distinctive OR symbols implies that this gate performs an OR function; this gate only performs an OR function if the two input values are 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 is the OR form of a NAND gate. |
| $F = \overline{A \cdot B}$<br>$F = \overline{A} + \overline{B}$<br>$F(H) = A(L) + B(L)$ | |

**Figure 14.11: Yet another mixed logic view of an NAND gate.**

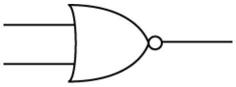
| | |
|---|---|
|  | This gate is a NOR gate; you're used to thinking of this gate as an OR gate with an inverted output. In a mixed logic context, this gate actually performs an OR function on its two positive logic inputs and outputs a negative logic result. The absence of bubbles on the inputs indicate 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. This is the OR form of a NOR gate. |
| $F = \overline{A + B}$<br>$F(L) = A(H) + B(H)$ | |

**Figure 14.12: A mixed logic view of a NOR gate.**

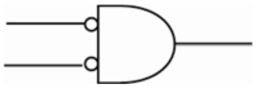
| | |
|---|---|
|  | This gate is a NOR gate; we can apply DeMorgan's theorem to the equation describing the NOR gate and arrive at a new equation. The final two equations on the left describe the operation of this gate using mixed logic. This gate performs an AND operation (note the AND symbol) if we provide two negative logic signals as inputs; the resulting output of the AND operation is positive logic. 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 is the AND form of the NOR gate. |
| $F = \overline{A + B}$<br>$F = \overline{A} \cdot \overline{B}$<br>$F(H) = A(L) \cdot B(L)$ | |

**Figure 14.13: A mixed logic view of a NOR gate.**

Figure 14.14 shows a summary of all the standard gates forms. At this point, you may be wondering why there are some many forms of gates out there. The short answer is that in some situations, we need flexibility in implementing logic functions. We always need to choose the gate that most appropriately represents the logic function we are performing, which is trickier in a mixed logic environment. In reality, there are still only AND, OR and inversion 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. If you don't use the proper gate in your design, you may have a working circuit but no one can understand your circuit.

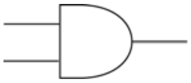
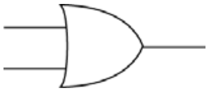
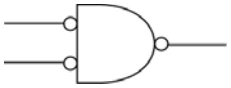
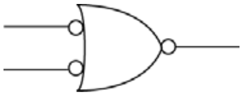
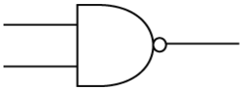
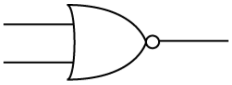
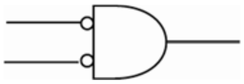
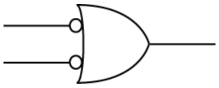
| **Standard Gate Forms** | |
|---|---|
| **AND functions** | **OR functions** |
|  |  |
| **AND form of AND gate** | **OR form of OR gate** |
|  |  |
| **AND form of OR gate** | **OR form of AND gate** |
|  |  |
| **AND form of NAND gate** | **OR form of NOR gate** |
|  |  |
| **AND form of NOR gate** | **OR form of NAND gate** |

**Figure 14.14: The giant summary of the strange new gate forms.**

## 14.6 AND/OR and NAND/NAND Forms

The AND/OR, NAND/NAND, OR/AND, and NOR/NOR forms are the most common forms. The relationship between these forms is nicer than you may be initially thinking after plodding through the algebraic manipulation in Table 11.1. Let's examine the AND/OR form and it's relation to the NAND/NAND form.

Figure 14.15 shows the common AND/OR form circuit implementation. In this implementation, overbars on the input signals replace the inverters in an effort to save me time drawing the circuit. The form in Figure 14.15 matches the equation in equation 1(b). Figure 14.16(a) shows the subsequent NAND/NAND circuit implementation as it appears in Equation 1(d). While the circuit implementation is correct in that it only uses NAND gates, it is misleading because it no longer resembles the AND/OR form it originated from.

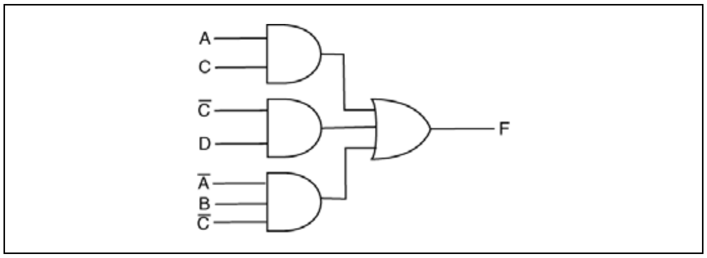
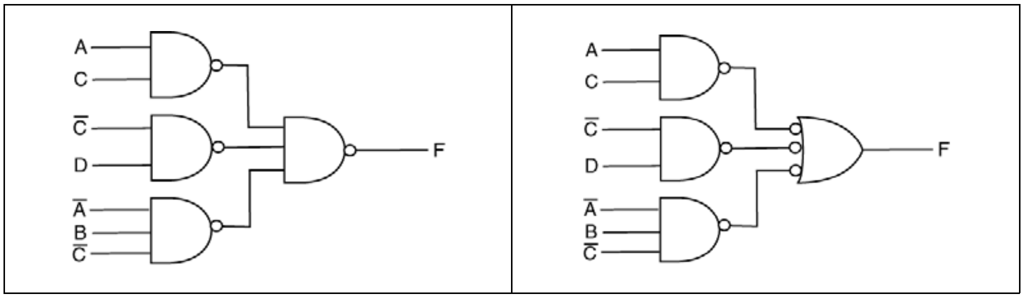


**Figure 14.15: The beloved AND/OR form.**

There are two forms of NAND gates as Figure 14.16(b) indicates. Since the right-most NAND gate of Figure 14.16(a) is actually implementing an OR function, you should use some type of OR-looking gate. Since this is a 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 Figure 14.16(b) shows.

Another thing that is disconcerting about the circuit of Figure 14.16(a) is that the bubbles "don't match"[4]. This is an indicator that something may be wrong. Although the implementation in Figure 14.16(a) is truly correct, someone not familiar with the circuit may have doubts. In summary, you should not the similarities between the circuit of Figure 14.15 and Figure 14.16(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 14.15 and add the bubbles in the appropriate location to make the circuit appear like that of Figure 14.16(b). I like calling this the no-brainer approach to circuit forms[5]. Moreover, these are two of the most popular circuit forms, with the NAND/NAND form being the most popular form.



**(a)** **(b)**

**Figure 14.16: The confusing (a) and clear (b) approach to NAND/NAND representations.**

## 14.7 OR/AND & NOR/NOR Forms

A similar type of argument can be made for the OR/AND & NOR/NOR circuit forms. Figure 14.17 shows the circuit implementation of the OR/AND form in Equation 2(b). We omit the inverters and replace them with complemented input signals (don't try this at home). If we implement this circuit in the NOR/NOR form of 2(f), you would end up with the circuit in Figure 14.18(a).

While the circuit in Figure 14.18(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 14.18(b). In this implementation, the right-most NOR gate is

[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 academic administrators wear brainlessness like a badge of honor.

[6] Mixed logic is an important concept that is covered in a later chapter.

implemented using the AND[7] version of the NOR gate. The comforting thing here is that the NOR/NOR form implementation of Figure 14.18(b) is strikingly similar to that of Figure 14.17. Once again, if you implement a function in NOR/NOR form, the circuit in Figure 14.18(b) is the best approach.

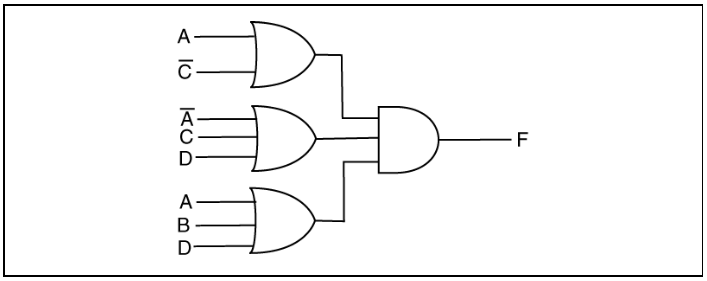
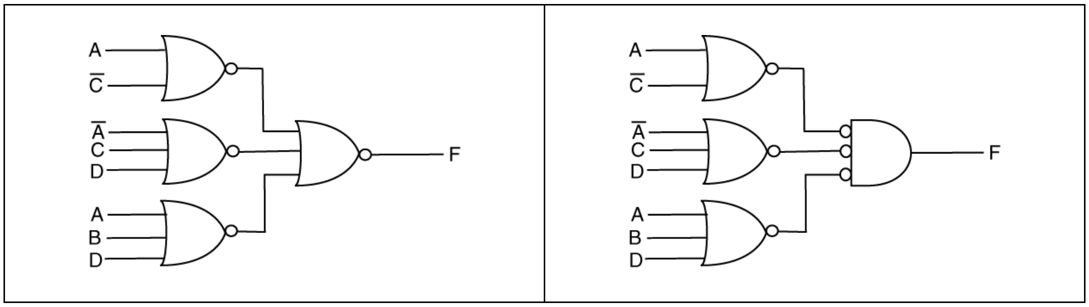


**Figure 14.17: The good'ole OR/AND form.**



**(a)** **(b)**

**Figure 14.18: The confusing (a) and totally clear (b) approach to NOR/NOR representations.**

## 14.8 Mixed Logic Analysis

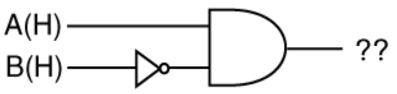
Yes, this is somewhat strange. The best way to learn about mixed logic is to use it in some actual examples. This section contains a few simple examples that show you the power of mixed logic analysis.

**Example 14.1: Mixed Logic Analysis**

Write equations describing the following circuit for the cases when the output is:

a) positive logic

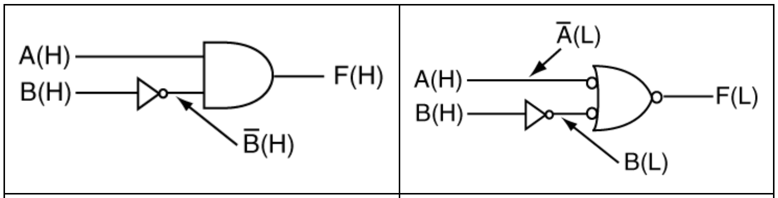
b) negative logic



**Solution**: 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. We left out the output logic level so that we can analyze the circuit using both negative and positive logic outputs[8].

[7] Once again, don't worry about this wording for now; this is another reference to mixed logic.

[8] Not listing the output logic level is a horrendously bad thing.



| $F(H) = (A \cdot \overline{B})(H)$ | $F(L) = (\overline{A} + B)(L)$ |
|---|---|
| **(a)** | **(b)** |

**Figure 14.19: Mixed logic analysis where (a) & (b) show the positive and negative logic interpretations of the output, respectively.**

**(a)** Figure 14.19(a) shows the case where the output is positive logic; the direct polarity indicator shows the logic level of the output. Note here 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 are both 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 is to use an equivalent signal representation for the **B(L)** signal, which allows us to list the signal as positive logic because this is what the gate expects. Once we write both inputs in positive logic form, we can write the resulting equation (shown under the circuit in Figure 14.19(a)). Note in this equation that the polarity indicators on both sides of the equation match. If the polarity indicators did not match, the equation would make no sense.

**(b)** Let's analyze this circuit as having a negative logic output as in Figure 14.19(b). In other words, we want to know what logic function the circuit executes if we interpret the output as negative logic. The first step is to redraw the gate such that there is a bubble on the output. Simply adding a bubble to the output would effectively change the gates, which is not what we want. We need to replace the original AND gate with an equivalent gate that has a bubble on the output.

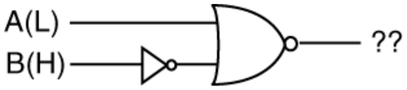
Figure 14.19(c) shows that the equivalent gate form for an AND gate is the OR form of an AND. Once we replace the gate and the bubble appears on the output, we rewrite the output of the gate to show that it is negative logic. The inputs to the new gate form need some modification also. The new gate form performs 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; this input requires no modifications. The **A** input is also positive logic but must be in negative logic as the bubbled input to the gate indicates. Since there is no inverter on this input, the approach we take is to rewrite the signal with an equivalent signal name; Figure 14.19(b) shows the result. The equivalent signal names contain a polarity indicator that indicates the gate receives a negative logic signal as the gate expects. Figure 14.19(b) shows the resulting equation below the circuit diagram.

**Example 14.2: Mixed Logic Analysis**

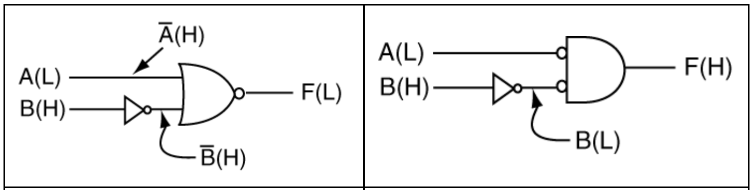
Write equations describing the following circuit for the cases when the output is:

a) negative logic

b) positive logic



**Solution:** This example differs from the previous example in that the inputs are in a true mixed logic form: the A and B inputs are in negative and positive logic forms, respectively.



| $F(L) = (\overline{A}+\overline{B})(L)$ | $F(H) = (A\cdot B)(H)$ |
|---|---|

**(a)** **(b)**

**Figure 14.20: An example of mixed logic analysis.**

**a)** The circuit in Figure 14.20(a) has two inputs, one is positive and the other is negative logic. The circuit in Figure 14.20(a) assumes the output is asserted low, which is the implication from the original drawing of the gate (because of the bubbled output). 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; Figure 14.20(a) shows that we do this by using an equivalent signal for the **A** input. The **B** input is in positive logic but the inverter changes its logic level. 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.

**b)** 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. In this case, we show the equivalent gate in Figure 14.20(b), which is the AND form of a NOR gate. This gate performs an AND function with a positive logic output if the two inputs are negative. The **A** input requires no modification because it is already in negative logic. The **B** input is originally in positive logic format but the inverter changes the logic level to negative logic. Once the inputs to the circuit are in negative logic form, we can write the equation performed by the gate.

The two previous examples provided us with a 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 note about this form of analysis.

- The output logic level always matched the gate output level. 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 we make is that we matched all the interior logic levels so there is no need to include them in the final equation.

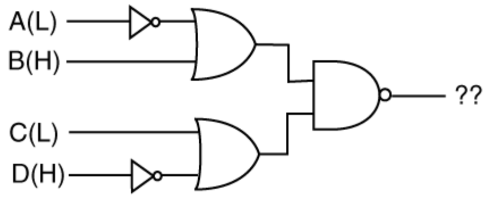
- In the final equation, the polarity indicators of the inputs and outputs match. If they did not match, the equation would not make sense; it would be evil confusion.
- Although we only had one circuit, we seemed to have generated two equations from it. This is true because we base the two final equations for these circuits on our interpretation of the circuit's output. In other words, depending on how we interpret the logic level of the circuit output, we are 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).

**Example 14.3: Mixed Logic Analysis**

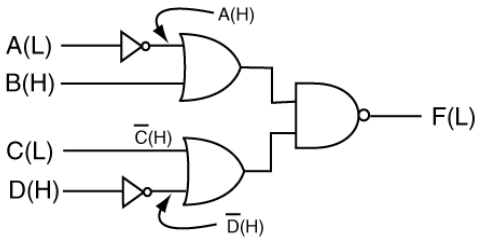
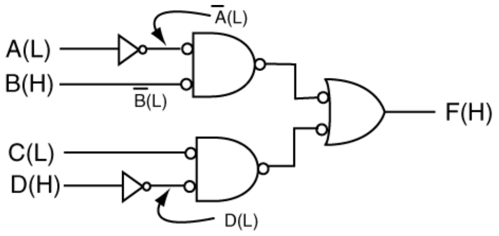
Write equations describing the following circuit for the cases when the output is:

a) negative logic

b) positive logic



**Solution**: This solution to this example is similar to the previous examples, so we omit the bloviated explanation. Once again, we matched all the logic levels in the circuit (input and output assertation levels match the presence (or lack thereof) of bubbles and the assertation levels of the final equation match.

|  A(H)<br>A(L)<br>B(H)<br>$\overline{C}$(H)<br>C(L)<br>D(H)<br>$\overline{D}$(H)<br>F(L) |  $\overline{A}$(L)<br>A(L)<br>B(H)<br>$\overline{B}$(L)<br>C(L)<br>D(H)<br>D(L)<br>F(H) |
|---|---|
| $F(L) = [(A+B)\cdot(\overline{C}+\overline{D})](L)$ | $F(H) = (\overline{A}\,\overline{B} + CD)(H)$ |

**(a)** **(b)**

**Figure 14.21: The total mixed logic analysis approach.**

For the circuit in Figure 14.21(a), we need the output to be negative logic; the NAND gate provides a negative logic output as evident from the bubble on the output. The NAND gate has positive logic inputs; the circuit is thus properly configured because the two OR gates provide positive logic output. The final step in this part of the solution is to use equivalent signals to re-write the gate inputs to all be in positive logic at the OR gate inputs. Only B is in proper form; we use equivalent signals on the other three input signals. We complete the problem by reading the circuit and writing the final equation.

For the circuit in Figure 14.21(b), we need to provide a gate that with no bubble on the output so that we can write the output in positive logic form. We can't simply remove the bubble, be we can use an equivalent gate with no bubble on the output. We do this by changing from the AND form of the NAND gate to the OR form of the NAND gates. When we make this change, the inputs to the NAND gate are now negative logic, which does not match the outputs of the OR gates. What we now must do is substitute the OR form of the OR gate with the equivalent AND forms of the OR gate, which provides negative logic outputs. The final step in this part of the solution is to use equivalent signals to re-write the gate inputs to all be in negative logic at the OR gate inputs. We complete the problem by reading the circuit and writing the final equation.

## 14.9 Mixed Logic Design

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.

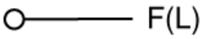
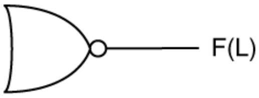
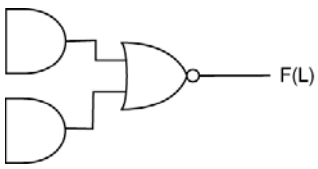
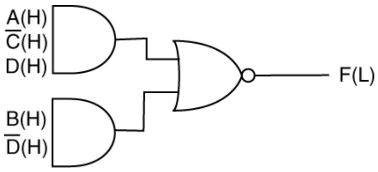
**Example 14.4: Mixed Logic Design**

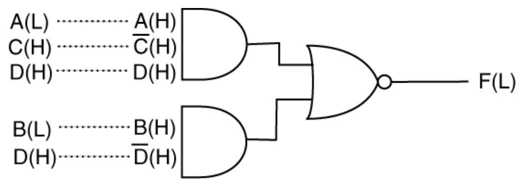
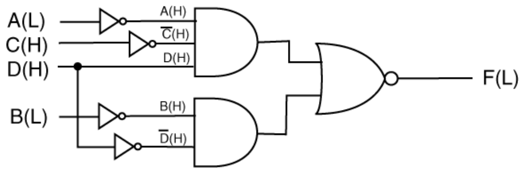
Design a circuit that implements the following function:

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

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 systematic approach.

| | |
|---|---|
| **Step 1:** Draw and label the output. Since we know the output is asserted low, draw a bubble, and label it to support the original problem description. |  |
| **Step 2**: Draw a gate such that it satisfies the logic function. The required logic function is an OR function so we draw a gate that looks like an OR gate. The example does not specify a gate type, so we use a NOR gate, which provides an OR function with a negative logic output. |  |
| **Step 3**: The equation includes two product terms, which we implement with an AND form of an AND; this gate is sufficient because the input to the OR gate is expecting positive logic inputs (note the absence of bubbles on the input). The AND gates provide positive logic outputs. The bubbles (or lack thereof in this case) match. |  |
| **Step 4**: We're ready to assign some logic for the inputs to the AND gates. Write the logic that the AND gates expect based on the problem's equation. The AND gates expect positive logic inputs, so list all the inputs in positive logic form. |   A(H)<br>$\overline{C}$(H)<br>D(H)<br>B(H)<br>$\overline{D}$(H)<br>F(L) |

| | |
|---|---|
| **Step 5**: We now include the input signals with the logic levels stated by the problem. The dotted lines mean nothing in particular; we draw them to refer to what each AND gate requires relative to the original equation and the logic levels of the inputs from the outside world. |  A(L)<br>C(H)<br>D(H)<br>A(H)<br>$\overline{C}$(H)<br>D(H)<br>B(L)<br>D(H)<br>B(H)<br>$\overline{D}$(H)<br>F(L) |
| **Step 6**: In the previous step, some of the input signals are not at the correct level required by the AND gates. For these cases, we need an inverter in order to switch the logic levels. Our approach was to switch some signal directly using inverter, and rewrite others using equivalent signal forms. |  A(L)<br>C(H)<br>D(H)<br>B(L)<br>A(H)<br>$\overline{C}$(H)<br>D(H)<br>B(H)<br>$\overline{D}$(H)<br>F(L) |

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 must implement functions using only one type of gate.

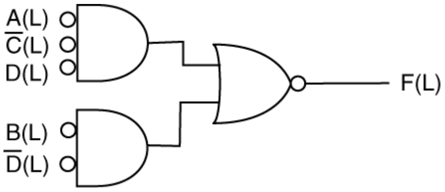
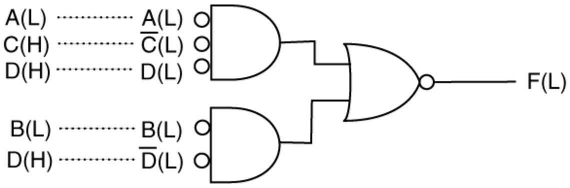
**Example 14.5: Mixed Logic Design**

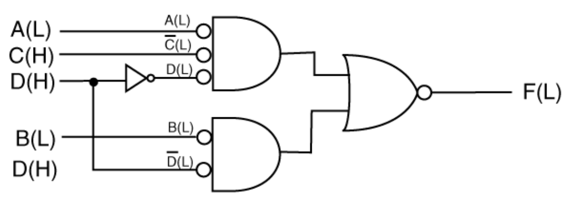
Design a circuit that implement the following function:

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

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 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**: We jump in at Step 4 because the first three steps are the same as the previous problem. We now need to choose NOR gates for the input gates rather than the AND gates. We need to choose a NOR gate that performs an AND function, so we choose the AND form of an NOR gate. This gate performs an AND function if the inputs are in a negative logic format. The diagram shows the signal requirements as they relate to the problem. We list each of the polarity indicators on the input signals as L as required dictated by the bubbles on the gate inputs. |  A(L)<br>$\overline{C}$(L)<br>D(L)<br>B(L)<br>$\overline{D}$(L)<br>F(L) |
| **Step 5**:. 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. |  A(L)<br>C(H)<br>D(H)<br>A(L)<br>$\overline{C}$(L)<br>D(L)<br>B(L)<br>D(H)<br>B(L)<br>$\overline{D}$(L)<br>F(L) |

| | |
|---|---|
| **Step 6:** The last step is matching the logic levels of the provided signals to those of the require function. We once again do this by a combination of inverters and equivalent signals. |  A(L)<br>C(H)<br>D(H)<br>B(L)<br>D(H)<br>A(L)<br>$\overline{C}$(L)<br>D(L)<br>B(L)<br>$\overline{D}$(L)<br>F(L) |

**Example 14.6: Mixed Logic Design**

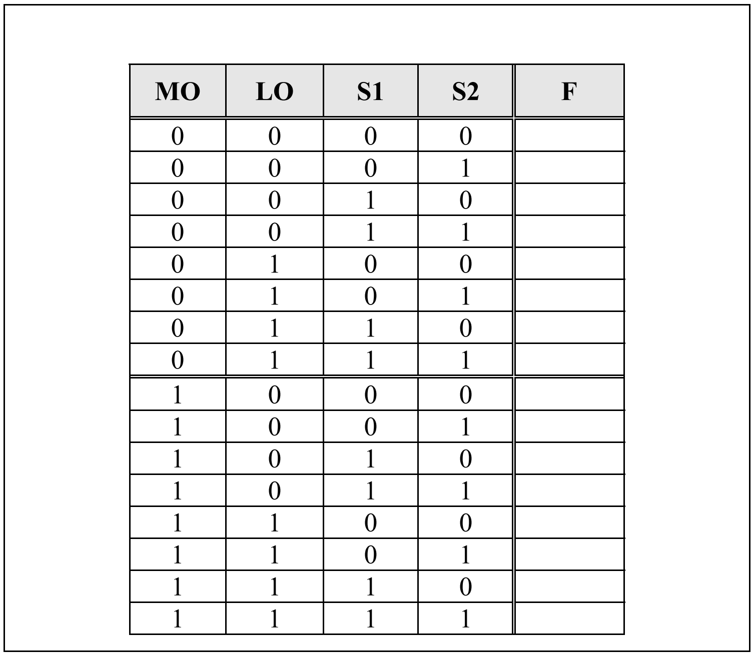
The previous two examples implemented the same equation using two different, but functionally equivalent circuits. Comment as to which circuit is "better".

**Solution:** This is somewhat of a trick problem, because the problem does not define the notion of "better". The truth is that "better" can mean about anything. The circuit of previous example uses four devices (three gates and one inverter). The circuit in the example before that uses seven devices (three gates and four inverters). From the standpoint of device count, the circuit implemented with four devices is clearly better. Therefore, from the standpoint of minimum device count, one circuit is "better".

**Example 14.7: 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 14.22 shows the empty truth table.



| 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 14.22: Truth table for Example 14.7.**

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 take here is somewhat more straightforward 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 also convert the negative logic output to positive logic. Once we specify the output, we complement it before we generate the subsequent logic. We don't need to do anything with the inputs at that point since the inputs still reflect the ordering (but not the numbering) used in 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 14.23: The modified truth tables with negative (a) and positive (b) logic outputs.**

The final logic we're looking for is in Figure 14.23(b). Once you toss the column for F into a truth table, you'll arrive at the POS equation in Figure 14.24.

$$F = (\overline{MO}\ \cdot\ \overline{LO})(\overline{S1} + \overline{S2})$$

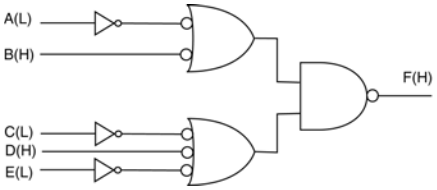
**Figure 14.24: The final equation for this problem.**

## 14.10 Chapter Summary

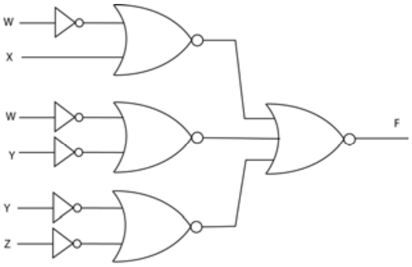
- The concept of mixed logic is based upon the “action” state of a digital signal. In all cases, either the ‘1’ or ‘0’ state is considered to be the action state if a digital signal. If a ‘1’ is considered the action state, the design is considered to be positive logic while if the ‘0’ is the action 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.

## 14.11 Chapter Exercises

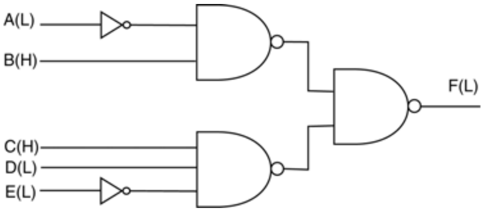
**1)** Write an equation for F(H) that describes the following circuit. Put your answer in DPI form.



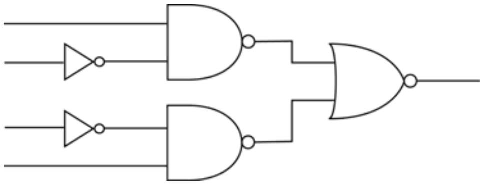
**2)** Write an equation for F(W,X,Y,Z) in NAND/NAND form.



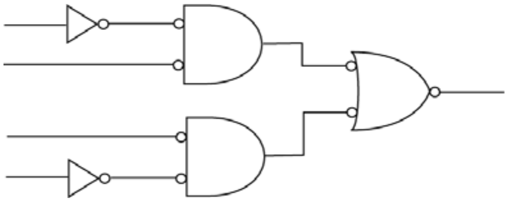
**3)** Write an equation for F(L) that describes the following circuit using DPI.



**4)** Without altering the function implemented by the circuit below, redraw the circuit using only OR gates and inverters. Minimize device count where possible.



**5)** Without altering the function implemented by the circuit below, redesign this circuit using only NAND gates and inverters. Minimize device count where possible.



**6)** Using only NAND gates and inverters, design 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).

**7)** Using only NOR gates and inverters, design 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).

**8)** Design 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.

**9)** Design a circuit that implements the following function. Use only standard gates and inverters in your solution.

$$F(H) = [(A + B + \overline{D})(\overline{A} + \overline{C})](H)$$

Consider the inputs to be: A(L), B(L), C(H), D(L);

**10)** Design a circuit that implements the following equation using any type of gate and inverters. Minimize the device count in your implementation. Consider the inputs and outputs to be: A(H), B(L), C(H), D(H), E(L), F(L).

$$F(L) = [(\overline{A} + B + C)(\overline{D} + E)](L)$$

**11)** Design a circuit that implements the following function; use only NAND gates in your solution. Consider the inputs to be: A(L), B(H), C(H), D(L). Show a gate-level schematic of your solution

$$F(A, B, C, D)(H) = [(A \cdot \overline{B}) + (B \cdot \overline{C} \cdot \overline{D})](H)$$

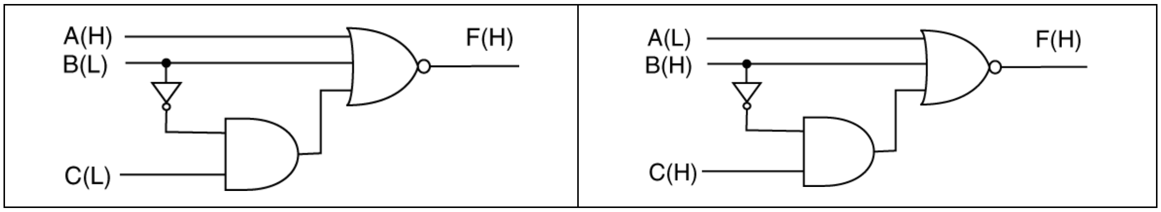
**12)** Design a circuit that implements the following function. Consider the inputs to be: A(L), B(H), C(H), D(L). Show a gate-level schematic of your solution; *Use only* ***NOR*** *gates in your solution.*

$$\mathrm{F(A,B,C,D)(H)} = [\mathrm{A} + \mathrm{B})(\overline{\mathrm{B}} + \overline{\mathrm{C}} + \mathrm{D})](\mathrm{H})$$

**13)** Provide a circuit diagram that implements the following mixed logic Boolean equation. Consider the logic levels of the input to be A(L), B(L), C(H). Use any type of gates you want but minimize the number of gates you use.

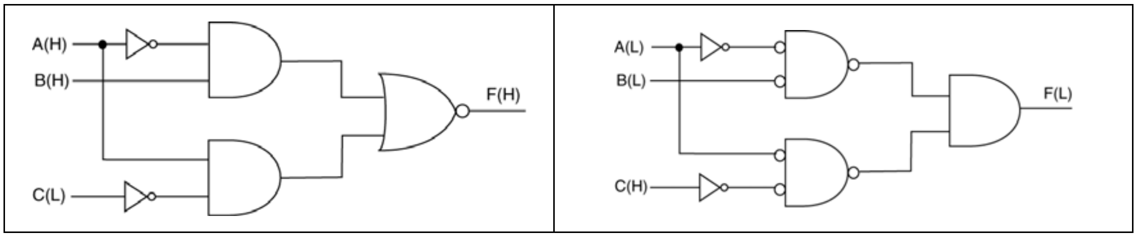
$$F(L) = \left[(A \cdot \overline{B}) + (\overline{B} \cdot \overline{C})\right](L)$$

**14)** Write a Boolean equation in DPI form that describes the following circuit. As the diagram indicates, make sure your answer is written in positive logic form.



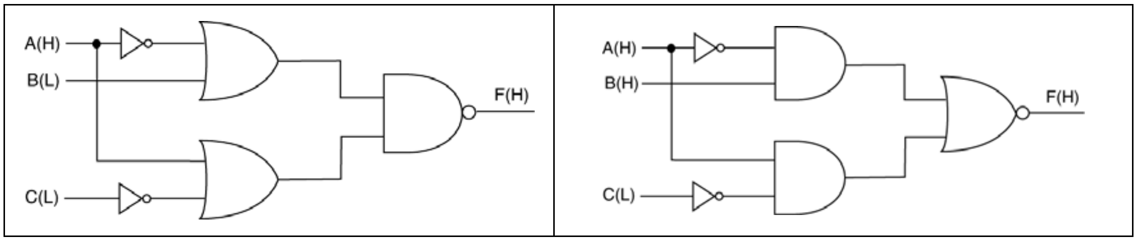
**(a)** **(b)**

**15)** Write a Boolean equation that describes the following circuit. The equation you generate should use proper and complete direct polarity indication. In other words, write the other side of the following equation: F(H) = ?? Do not attempt to reduce the final circuit equation; make sure your final equation is in proper form.



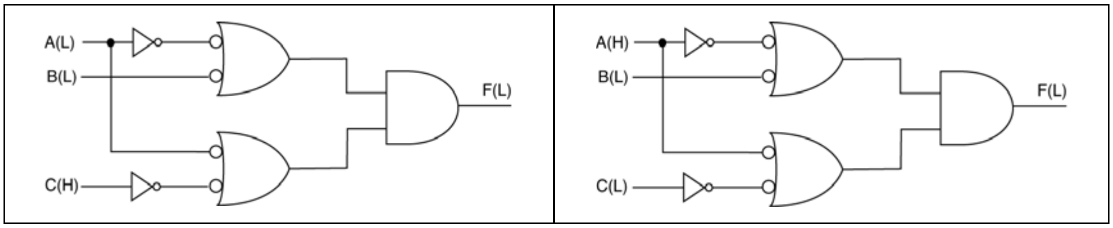
**(a)** **(b)**

**16)** Write a Boolean equation that describes the following circuit. The equation you generate should use proper and complete direct polarity indication. In other words, write the other side of the following equation: F(H) = ?? Do not attempt to reduce the final circuit equation; make sure your final equation is in proper form.



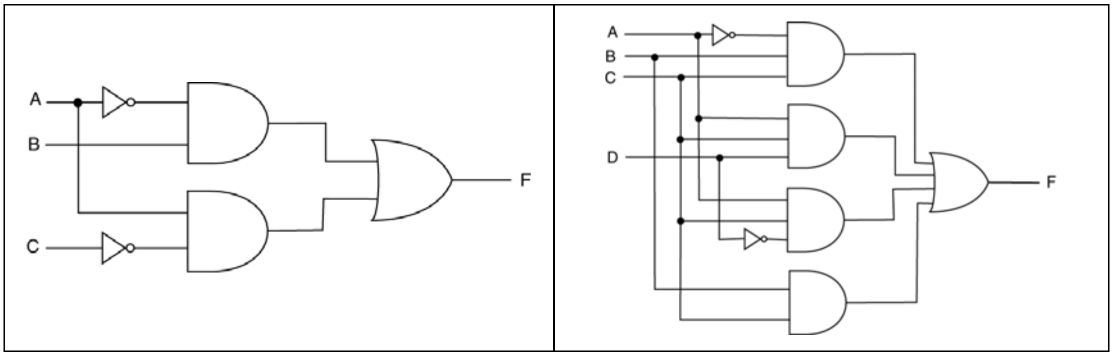
**(a)** **(b)**

**17)** Write a Boolean equation that describes the following circuit. The equation you generation should use proper direct polarity indication. In other words, write the other side of the following equation: F(L) = ?? Do not reduce the final circuit equation.



**(a)** **(b)**

**18)** Change the following circuit from AND/OR form to NAND/NAND form.



**(a)** **(b)**

## 14.12 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**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 generates a low output voltage if and only if there are two (or more) adjacent vacant slots.

# 15 Modular Design

## 15.1 Introduction

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 to digital design: Modular Design, or MD. MD is the most powerful approach to digital design, though you won't see a lot of that power in this since we still need to introduce more digital design foundation modules.

**Main Chapter Topics**

**MODULAR DESIGN:** This chapter presents the basics of Modular Design (MD) starting with an overview and ending with design examples that we can best solve using the MD approach.

**Why This Chapter is Important**

- Be able to describe the differences between BFD, IMD, and MD approaches to digital design.

## 15.2 The Big Digital Design Overview

Because of your increased your knowledge and abilities, you're ready for 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 fit into one of these approaches. The following list highlights the three design approaches and includes some modest explanation as well. Table 15.1 compares and contrasts these three approaches.

**BRUTE FORCE DESIGN (BFD):** This was the first design approach we worked with; it involves assigning a single output to every possible input combination via a truth table. The tabular format (truth tables) limits this approach to designs with a four or less inputs.

**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 RCA.

**MODULAR DESIGN (MD):** We applied this approach in a few problems, though you did probably not realize it. This approach involved drawing bunches of black boxes to model our designs. We also drew boxes within boxes within boxes, which we labeled as hierarchical design. It turns out that hierarchical design is a form of modular design.

| Design Approach | Pros | Cons |
|---|---|---|
| Brute Force Design | Straight forward | Limited by number of inputs |
| Iterative Modular Design | Straight forward | Not applicable to all designs |
| Modular Design | Massively powerful | Requires a working brain[1] |

**Table 15.1: Matrix explaining why Modular Design can save the world.**

## 15.3 Modular Design Overview

Modern digital design primarily uses Modular Design. You perform modular design by plopping down black boxes and connecting them up in intelligent ways such that they solve problems. Modular design involves keeping a bag full of standard digital modules (which we refer to as the digital design foundation modules) and assembling those modules in such a way as to solve digital design problems. The half-adder, the full-adder, and the ripple carry adder (RCA) are first modules we worked with; the RCA was our first digital design foundation module.

The overall approach of MD is to abstract circuits to a higher level in order to increase your 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 model based, you must make sure that the entity reading your design[2] can fully comprehend what you're attempting to convey. Here are some guidelines you must follow when doing the MD thang:

> **Be Clear and Concise:** A messy BBD hinders efficient information transfer. Use a ruler if you're modeling by hand, but there is no big need to use a drawing program if your BBD is neat.
>
> **State All Assumptions:** Any unstated assumption you make could quickly confound your design if the reader does not know and/or understand you're assumptions
>
> **Label Everything:** Make labels help prevent readers from making assumptions about the circuit.
>
> **Provide a definition for all black boxes:** Every box you use in your model should either be clearly defined somewhere or be a digital design foundation module. If you call out a foundation module in your design, everyone knows what it is and there is no need to define it at a lower level. Be aware that you must use these modules exactly as we originally defined them, or people can't be sure of what you're modeling. Table 15.2 shows a visual representation of this point.

[1] Thus, you will not find viable digital designers in an academic administrative setting.

[2] It could be a person or a computer (such as the HDL synthesizer).

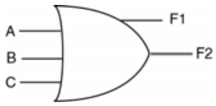
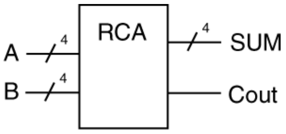
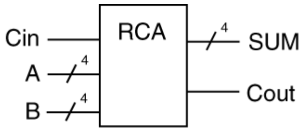
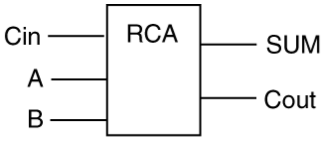
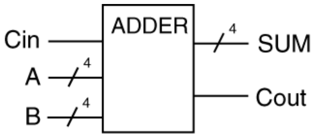
| Model | Comment |
|---|---|
|  | This model sort of looks like a 3-input OR gate, but having two outputs makes it non-standard. Being non-standard, the circuit's output characteristics are a mystery. **This is an invalid model**. |
|  | This is a true digital box. Since we know what an RCA is, and the inputs and outputs of the box labeled RCA match what we know about RCAs, we know exactly how it works. **This is a valid model** and there is no need to define it anywhere else in your model. This RCA does not show a carry-in input, but it's still an RCA. |
|  | This is also a true digital box. If you replace the HA in a RCA with a FA, you 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 circuit has the RCA label, 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 an ADDER definition in your design. |

**Table 15.2: Some good and bad example of standard digital black boxes.**

Not that rules are good things, but they can help when first embarking on the MD approach to digital design. There is one excellent quality regarding MD: the problems have a strange way of doing themselves based primarily on the problem description. We outline this approach in Figure 15.1 and apply this approach in the design examples that follow. The final comment: you need to be creative and clever with your solutions, as you will inevitably run into situations that you have not seen in a previous example.

- **Read the problem:** Yes, a great start.
- **Draw a high-level black box diagram that shows the design's interface (I/O):** This is not always an easy step based on the problem statement as sometimes the important information buries itself deep in the problem description. Completing this step is that it inevitably helps you understand the overall problem.
- **Make an inventory of the modules your solution requires:** This can be a straightforward step because the problem typically provides major clues. For example, if a problem says something like "add" or "sum", you know your design requires an RCA.
- **Connect the Lower-level Design Entities**: The previous step leaves you with a bunch of black boxes in your design; this step entails connecting those black boxes in an intelligent manner.
- **Provide Adequate Models for Any Non-Standard Black Boxes used in the Design**: 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. You must, however, you completely describe any non-standard module you use in your design.
- **Check your final diagram for the following:**
    - Make sure the highest-level black box has an appropriate label
    - Make sure all module inputs connect to something
    - Make sure all signals include appropriate labels
    - Make sure to label all bundle widths
    - Make sure all lower-level design entities include labels
    - Make sure all labels in diagram are self-commenting in nature

Most importantly: ***DON'T GET STUCK!*** Digital design is an inherently iterative process. Also, recall Mealy's Fourth law of digital design that states the design process is circular, not linear. Do your best to complete all the bullets listed above in the order, but realize the main goal is to solve the problem. If you leave something out of your design, simply add it when you realize it's missing. Lastly, realize that the listed bullets are one person's view of digital design. If you want to become a good digital designer, ***YOU MUST DEVELOP YOUR OWN STYLE!*** The only constraint is that you solve the problem in a reasonably efficient way and in a reasonable amount of time.

**Figure 15.1: The desired approach to solving modular design problems.**

The notion of modularity in digital design is so important, we coin yet another one of Mealy's laws:

**Mealy's Fifth Law of Digital Design**: Model circuits using many smaller sub-modules as opposed to fewer larger sub-modules; as this approach supports testing and increases the chances module reuse.

**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 foundation modules in your solution, provide an adequate description. State what controls the circuit in your solution.

**Solution**: The first step is to draw the high-level BBD; Figure 15.2 shows a nicely labeled model for this problem.

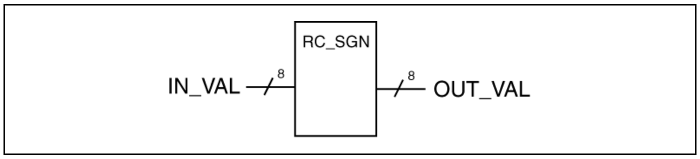


**Figure 15.2: Black box diagram for the RC Sign Changer.**

The next step is to gather in what you know about changing the sign of binary numbers in RC format. The standard method we know is 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. We need to use the other approach to changing the sign which, was taking the 1’s complement and then adding ‘1’. Taking the 1’s complement of the input requires an inverter for each individual bit input to the circuit.

The next step in this problem is to make of initial inventory of the modules the final circuit requires. Taking the 2’s complement requires that we add three values, which means this circuit needs an RCA. It seems for now that’s all the circuit needs, but if we later find that the circuit requires other modules, we add them. The making an inventory step in digital design is always going to be an iterative step; we give the BBD our best shot, but we know can always add more modules learn more about the problem during our progression towards the solution (digital design is circular, not linear). Figure 15.3 shows the lower-level BBD for this example. We list a few more interesting things about this circuit below:

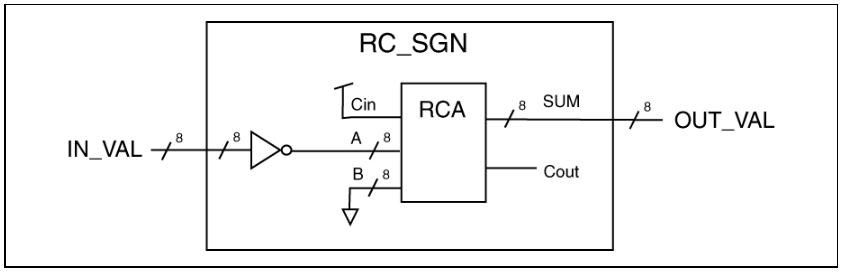


**Figure 15.3: Black box diagram for RC Sign Changer.**

- 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 bundle notation in Figure 15.3 appears on both the inside of the RC_SGN box as well as the outside; either listing is fine.
- It appears that the 8-bit bundle uses a single inverter. This is a common shorthand notation for indicating the inversion of every signal in the bundle. We could have drawn eight inverters but it would have cluttered our diagram.
- 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 in the circuit, which we consider to be a ‘1’.

- There is a funny shape connected to the **B** signal, which indicates that the **B** input of the RCA is connected to "ground" or a logical '0'. This notation indicates that each of the eight individual signals in the bundle connects to ground.
- The **Cout** signal of the RCA is unconnected; which is no big deal, as your design does not use it. Although you always need to connect your inputs to something, you don't need to connect the outputs if the circuit does not use them.
- 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 we connected the circuit 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 = !A + 1**.

This circuit has no control because the RCA has no control inputs; the output reacts to the input independent of the input values.

As a final note, there are two ways to configure the RCA in this problem. The goal for this problem was to output **(!A + 1)**; the solution does this by grounding one of the bundled inputs to the RCA and using the **Cin** to provide the '1'. Another equivalent approach would be to ground the **Cin** input and then connect a one to the **B** input of the RCA. The "one" on the B input would be "00000001", as that RCA input expects 8-bit data.

**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. State whether the circuit has "no control", "internal control", or "external control".

**Solution**: The solution starts with drawing a BBD of your solution, as we nicely show in Figure 15.4.

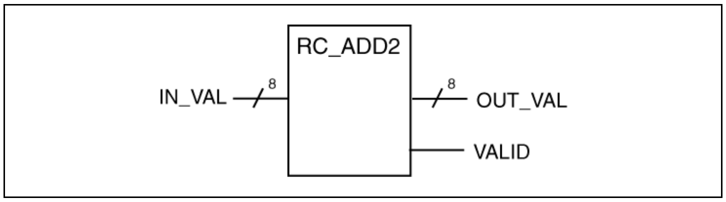


**Figure 15.4: Black box model for the solution.**

The next step is to make an inventory of the modules that go inside the top-level BBD. We do this by first looking back at the problem description for clues. The first clue is that you'll be adding a number to another number, which means that you're going to need an RCA. The next thing we need is some type of circuitry indication when the solution is valid or not. We refer to this "control" circuitry. OK… let's put it down; check out Figure 15.5.

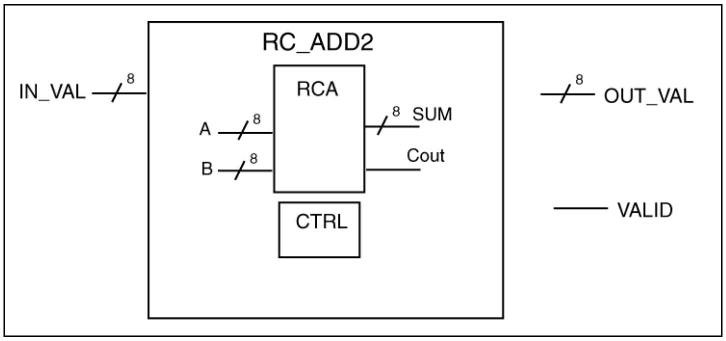


**Figure 15.5: The next step in the solution.**

Here are some interesting to note about Figure 15.5 that helps you move toward the solution. Figure 15.6 shows the result of listing all of these items.

- The RCA adds two things: the IN_VAL and the number "00000010" (which is 2 represented as an 8-bit in signed binary number). Therefore, we can connect **IN_VAL** to one of the RCA operands and "hardwire" a binary "2" to the other operand. We indicate this in the diagram by listing "00000010" near the bundle in question.
- The output of this circuit is the result of the addition so we can connect the output of the RCA to the **OUT_VAL** signal.
- The CTRL circuit indicates if the operation was valid or not. Although the inputs to the CTRL box are still unknown, we know the output is 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 is only invalid if the two input operands have the same sign but generate a result of a different sign. Therefore, the CTRL box needs three inputs: the sign bits of the two RCA operands and the sign-bit of the **SUM** operand.
- We don't 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; one way to do with is with a truth table. 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. Because the sign-bit of the **A** input is always '0', we list the table entries of **A**=1 as don't cares (dashes).

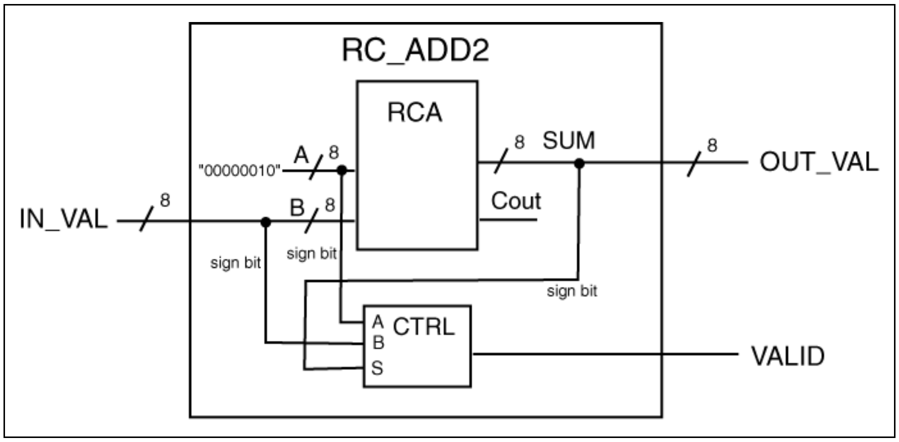


**Figure 15.6: The next step in the solution.**

| 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 describing the truth table of Figure 15.7 is **VALID = B + !S**. 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 if you think about it for a few minutes.

Figure 15.8 and Figure 15.9 show the final solution. 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 Additionally, this circuit has no type of control; the output react to the input in the same manner independent of the input values.

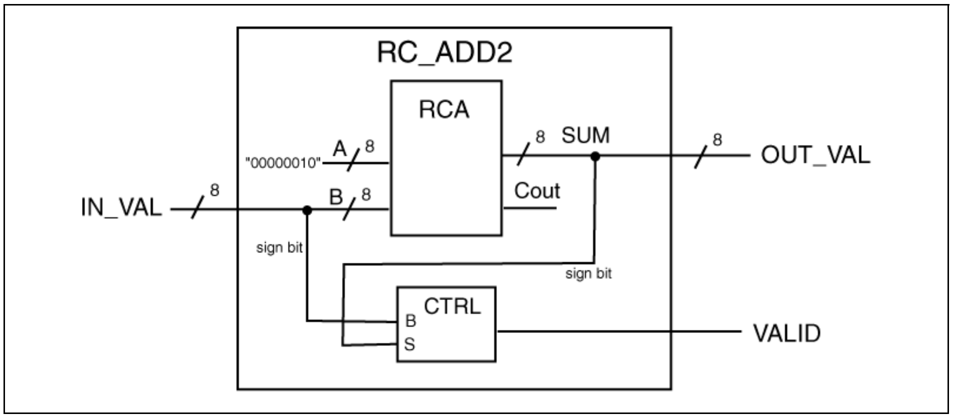


**Figure 15.8: The final solution to this example.**

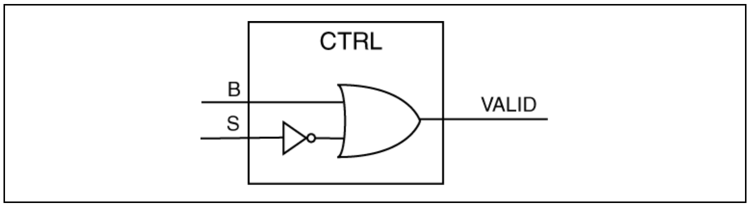


**Figure 15.9: The other part of the final solution to this example.**

This is a true hierarchical design. It did not need to be 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 clearer. We didn't have 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. This is an important concept because you may not have a good idea about the solution, but you'll have a direction to go in. As you go in that direction, you'll pick up clues to the solution, or maybe toss your approach and start over. This is a circular approach to design, which is much better than attempting to force a linear approach, as many technical people often do.

**Example 15-3: 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 signed binary inputs **A** and **B** (RC format). When the **SUB** input is high, the 8-bit signed binary output (RC format) indicates the result of **B** subtracted from input **A**. Otherwise, the output of the circuit indicates of addition of the **A** and **B**. For this problem, assume that the result is always valid. Use any support logic you may require but minimize the amount of hardware used in this circuit. Use the modular design approach and provide a top-level and lower-level BBD for your solution. State whether the circuit has "no control", "internal control", or "external control".

**Solution**: The first step is drawing a BBD of the circuit; Figure 15.10 shows this step.

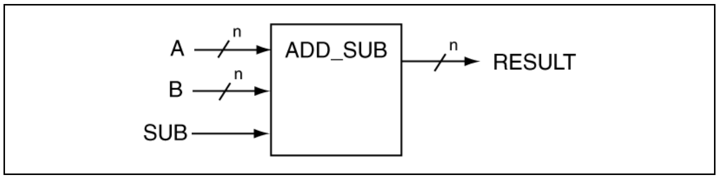


**Figure 15.10: Black-box diagram of the Adder/Subtractor circuit.**

Our approach is to recall that we can perform subtraction in binary by first multiplying one operand by -1 and then adding the result to the other operand. This means performing a two's complement on one operand when using numbers in RC format, which is the indirect subtraction by addition approach. We obtain the two's complement by taking a 1's complement and adding 1. Equation 15.1 shows the RCA's operation; the task in this problem is to change the sign of the **B** input when the circuit must perform subtraction.

**SUM = A + B + Cin**

**Equation 15.1: What 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 something like the circuit in Figure 15.11. Another way to view this circuit is that the value of the **SUB** signal is always included in the addition operation of the RCA_FA. If the **SUB** = '0', thus indicating an addition operation, it has no effect on the result.

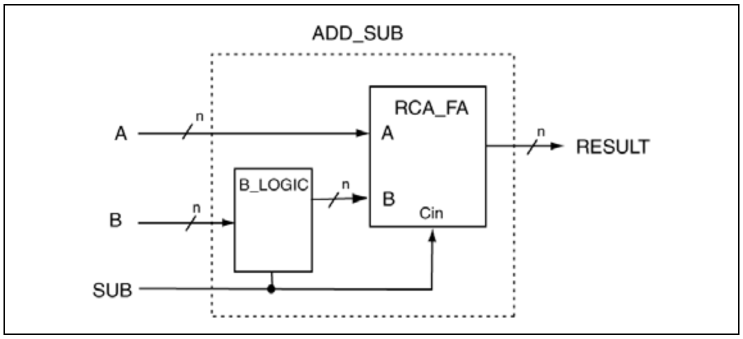


**Figure 15.11: The final circuit.**

The final step in this problem is defining the B_LOGIC block in Figure 15.11. There is a well-known approach to this problem, which is to notice that signal **B** sometimes requires inversion before it connects 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 **B** and add ‘1’ to the RCA_FA module via the **Cin** input. This circuit uses the **SUB** input as a control input as it decides what value is output on the result signal.

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. Therefore, the XOR gate here is ether an inverter or buffer.

Figure 15.12 shows the final circuit for the B_LOGIC block with a few interesting features worth noting. First, we decompose signal **B** into its parts on the diagram with the assumption that **B(7)** is the MSB while **B(0)** is the LSB. We reassemble the output from its parts back into a bundle.

| SUB value | RCA_FA operation | Operation |
|---|---|---|
| ‘0’ | $SUB = A + B + 0$ | A + B |
| ‘1’ | $SUB = A + \overline{B} + 1$ | A - B |

**Table 15.3: Tabular view of RCA_FA operation.**

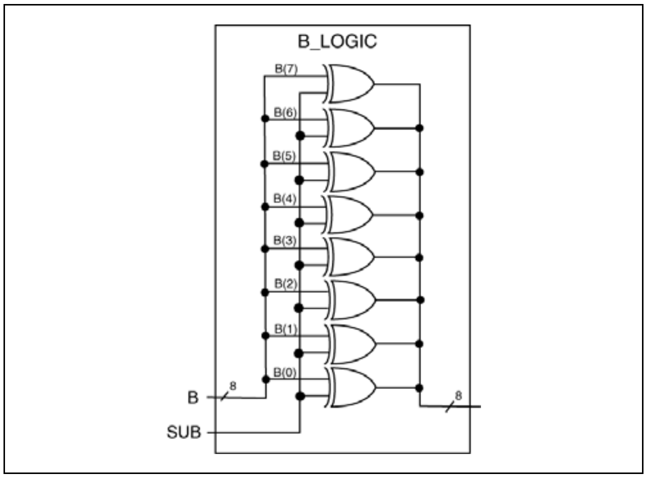


**Figure 15.12: The schematic for the B_LOGIC block.**

This is a well-known solution to this problem. Figure 15.13 shows a better approach to the final solution of this problem; this solution is better because it was easier to draw. Figure 15.13 uses a special shorthand notation; although XOR gates only have two inputs, Figure 15.13 seems to indicate that the XOR gate can accept a bundle input. In actuality, the special XOR gate in Figure 15.13 is the same circuit as the B_LOGIC block in Figure 15.12.

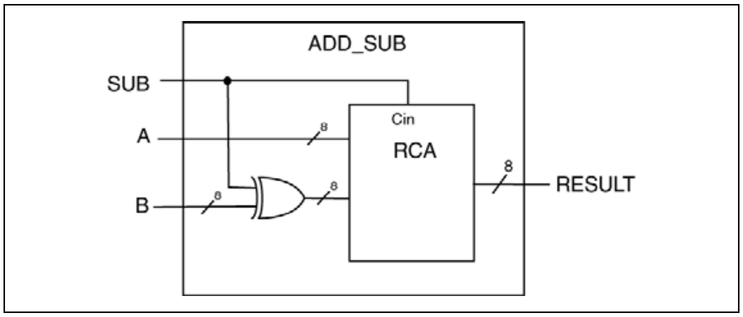
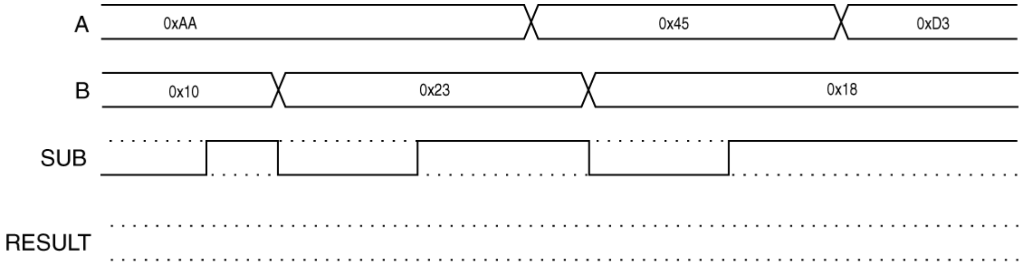


**Figure 15.13: An alternate and popular approach to the final circuit.**

### Example 15.4: Timing Diagrams

Based on the solution to Example 15-3, complete the following timing diagram.



**Solution**: The problem states the value on **B** is either subtracted from or added to **A** based on the value of signal **B**. Figure 15.14 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**.

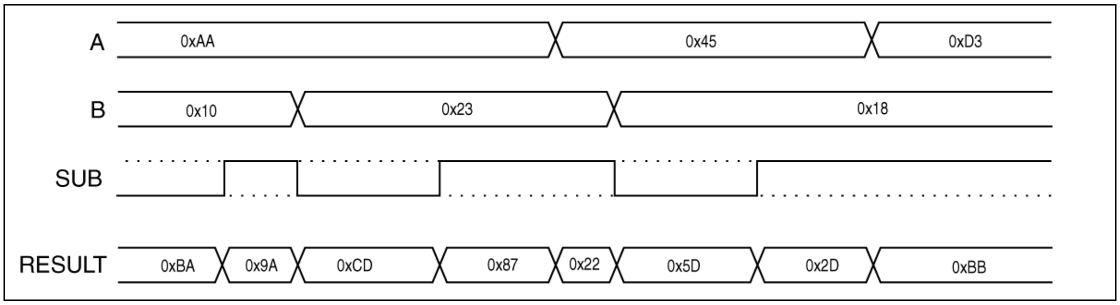


**Figure 15.14: The solution to Example 15.4.**

## 15.4 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.
- The general rules for creating hierarchical BBDs are:
  - **Be Clear and Concise:**.
  - **Label All Assumptions:**
  - **Label Everything:**
  - **Provide a definition for all black boxes:**
- 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.
- MD is an inherently iterative design process; don't expect to complete a working design in one pass.
- An overview of the approach to MD-type problems can be stated as:
  - **Read the Problem**
  - **Draw a High-level Black-box Interface Diagram**
  - **Create an inventory of  the Lower-level Design Entities**
  - **Connect the Lower-level Design Entities**
  - **Provide Adequate Models for Any Non-Standard Modules**
  - **Check Your Final Diagram for All Important Details**

## 15.5 Chapter Exercises

**1)** Briefly explain what exactly the notion of a modules "interface" refers to.

**2)** Why do we consider IMD to be a subset of MD? Briefly but fully explain.

**3)** List several advantages to using a self-commenting style in your black box diagrams.

**4)** Describe why MD is more powerful than BFD.

**5)** Explain why you don't need to provide models for underlying foundation modules but you do need to provide models for all other modules you use in your designs.

**6)** Briefly describe why the best approach is to use as many foundation modules as possible in your designs as opposed to defining new modules.

**7)** Briefly describe why "brute force design" or "iterative design" was a limited and inefficient design approach?

## 15.6 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**1)** Design a circuit that subtracts '3' from an 8-bit signed binary number. Assume the number is in radix complement form and that the output is an 8-bit value in RC format. . This circuit has an output signal VALID that is '1' when the subtraction operation is valid.

**2)** Design a circuit that multiplies an 8-bit signed binary number by two. Assume the number is in radix complement form and that the output is an 8-bit value in RC format. This circuit has an output signal VALID that is '1' when the operation is valid.

**3)** Design a circuit that multiplies an 8-bit signed binary number by three. Assume the number is in radix complement form and that the output is an 8-bit value. This circuit has an output signal VALID that is '1' when the operation is valid.

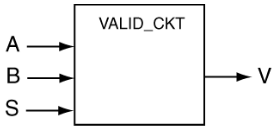
**4)** Design a circuit that can add four 10-bit values and generates a 10-bit output. If any of the four inputs values is greater than 255, then the output is always 0; otherwise the output reflects the summation of the four values. Assume that the inputs and output s are in unsigned binary format.

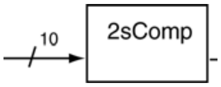
**5)** Design a circuit that can add four 8-bit values and generates an 8-bit output. This circuit also has a VALID output that indicates when the value on the 8-bit output is correct.

**6)** Design a circuit that can add four 8-bit values and generates a 9-bit output. This circuit also has a VALID output that indicates when the value on the 9-bit output is correct. Assume that the inputs and outputs are in unsigned binary format.

**7)** Design a circuit that has two 8-bit signed binary (RC format) inputs. The circuit has four outputs, which includes the value of A+B and a signal to indicate if the value is correct or not, and the value of A-B and a signal to indicate whether the output is valid or not. Assume the results of the sum & subtraction are both 8-bit values in RC format.

**8)** Design a circuit that performs the following operation: C – A – B (the values of A & B are subtracted from C). Assume that A, B & C are 10-bit signed values in RC form. This circuit has two outputs: RES, which is a 10-bit result (also in RC form) and VALID, which indicates if the 10-bit RES output is value is valid based on the math operations performed by the circuit. Feel free to use the 2sCOMP and VALID_CKT box provided below. Include a block box diagram for both the top-level circuit as well as the underlying circuitry.





**9)** 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.

**10)** 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 can only use 10-bit RCAs for this circuit.

# 16 Decoders

## 16.1 Chapter Overview

Decoders provide one of the most straightforward ways of modeling certain types of digital circuits. Up until now, we've primarily been dealing with logic functions in the form of Boolean equations. This is a good beginning approach, but using Boolean equations as a basis for digital design limits the complexity of the circuit. Digital design rarely has much to do with Boolean functions as it represents a low-level and thus inefficient approach to digital design. We're ready to generate our designs at a higher level because that is the most efficient approach.

### Main Chapter Topics

**DECODERS**: The chapter introduces the decoder, which is a standard digital circuit. We divide decoders into one of two types: *standard* and *generic* decoders. Because of their inherent genericity, decoders are quite versatile in digital design.

### Chapter Acquired Skills

- Be able to describe the differences between generic and standard decoders
- Be able to use generic and standard decoders in digital designs
- Be able to describe the underlying hardware of a simple standard decoder

## 16.2 Introduction to Decoders

We use the word *generic decoder*, or just *decoder*, to refer to the standard digital device where the values of the decoder's input always produce the same values on the decoder's output. This is a generic definition of a decoder, thus we refer to most decoders as "generic" if we can model them in tabular format (a truth table). The basis of all things digital are basic gates, which we defined using tables; we can thus consider basic logic gates as decoders because of their tabular definitions.

In addition to the generic decoder, there is a *standard decoder*. The terms "generic" and "standard" decoders are terms that you won't find in other digital design texts; I created these names to simplify the digital design paradigm. The standard decoder is a special type of a generic decoder and has a special relationship between the inputs and outputs. Figure 16.1 shows that, a standard decoder is a subset of a generic decoder. Standard decoders have specific uses while generic decoder usage is open-ended.

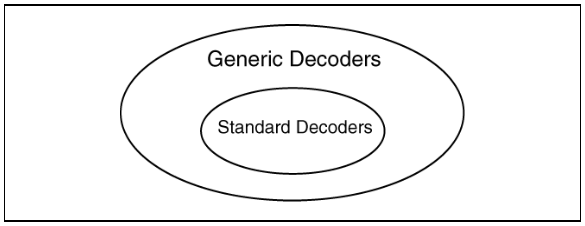


**Figure 16.1: Venn diagram showing the hierarchy of decoders.**

Modeling digital circuits using tables is powerful because we can easily translate the tables to a hardware description language (HDL). You may have a notion of the "power of tables" from your programming career in that using "look-up-tables" or "LUTs"; the same usefulness of LUTs applies to hardware modeling. The approach in modern digital design is to allow the development tools to do the work for you. Thus, modeling circuits using decoders (LUTs) hands a significant portion of the circuit implementation effort to the tools. If you need to implement some "logic" using an HDL, the best approach is to model the function in tabular format, then allow the tools to do the rest.

Our new working definition of a generic decoder is this: any digital device that establishes a functional relationship between the device input(s) and output(s). We use generic decoders to model LUTs. This is important, so we need to coin yet another one of Mealy's new laws. You should always be on the lookout for opportunities to use decoders rather than trying to generate some fancy logic.

**Mealy's Seventh Law of Digital Design**: Always first consider modeling a digital circuit or part of a digital circuit using some type of a look-up table (LUT).

## 16.3 Generic Decoders

Generic decoders are so general, it's tough to say much useful about them. If you can describe a circuit in tabular format, you've officially modeled a decoder. Figure 16.2 shows a black box diagram of a generic decoder. There can be any non-zero number of inputs and outputs; the number of inputs and outputs don't need to match.

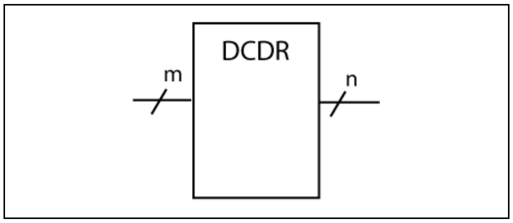


**Figure 16.2: A black box diagram of a generic decoder.**

You can define two general types of tables: 1) complete tables, and, 2) incomplete tables. Both tables are equally straightforward to model using an HDL. We define a complete table as a table that has a row for every unique combination of the circuit's inputs; a non-complete table is any table that is not a complete table. We make this distinction so you realize that you don't need to completely specify every possible input combination for generic decoders. Additionally, HDLs have solid support for modeling incomplete tables.

Figure 16.3 shows completely and incompletely specified tables. The table in Figure 16.3(a) has three inputs; because there are eight rows in Figure 16.3(a), we consider this table completely specified. The table in Figure 16.3(b) has three inputs, but only five of those three inputs combinations have outputs. Not declaring outputs indicates that for the missing input combinations, the designer for some reason does not care about the outputs. Another approach to non-complete tables is to list the missing inputs and state the outputs as don't cares, which we do in Figure 16.3(c).

| A | B | C | VAL |
|---|---|---|---|
| 0 | 0 | 0 | 011 |
| 0 | 0 | 1 | 110 |
| 0 | 1 | 0 | 010 |
| 0 | 1 | 1 | 011 |
| 1 | 0 | 0 | 111 |
| 1 | 0 | 1 | 100 |
| 1 | 1 | 0 | 000 |
| 1 | 1 | 1 | 111 |

**(a)**

| X | Y | Z | VAL |
|---|---|---|---|
| 0 | 0 | 0 | 011 |
| 0 | 0 | 1 | 110 |
| 1 | 0 | 1 | 100 |
| 1 | 1 | 0 | 000 |
| 1 | 1 | 1 | 111 |

**(b)**

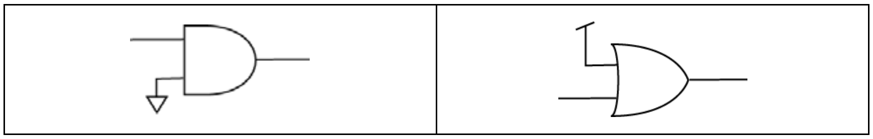
| X | Y | Z | VAL |
|---|---|---|---|
| 0 | 0 | 0 | 011 |
| 0 | 0 | 1 | 110 |
| 0 | 1 | 0 | - - - |
| 0 | 1 | 1 | - - - |
| 1 | 0 | 0 | - - - |
| 1 | 0 | 1 | 100 |
| 1 | 1 | 0 | 000 |
| 1 | 1 | 1 | 111 |

**(c)**

**Figure 16.3: A completely specified table (a), and an incompletely specified table (b) & (c).**

## 16.4 Standard Decoders

Before we study the internals of a standard decoder, we first need to review some characteristics of a few basic gates: the basic AND & OR logic gates. You can effectively kill the output of a AND & OR gates by connecting an input to ‘0’ and ‘1’, respectively. Figure 16.4 shows a gate-level depiction of the gate-killing functionality. The circuit in Figure 16.4(a) uses an inverted arrowhead to indicate a connection with ground (‘0’). Figure 16.4 (b) shows the slanted T symbol to indicate a connection to the circuit’s high voltage (‘1’).



**(a)** **(b)**

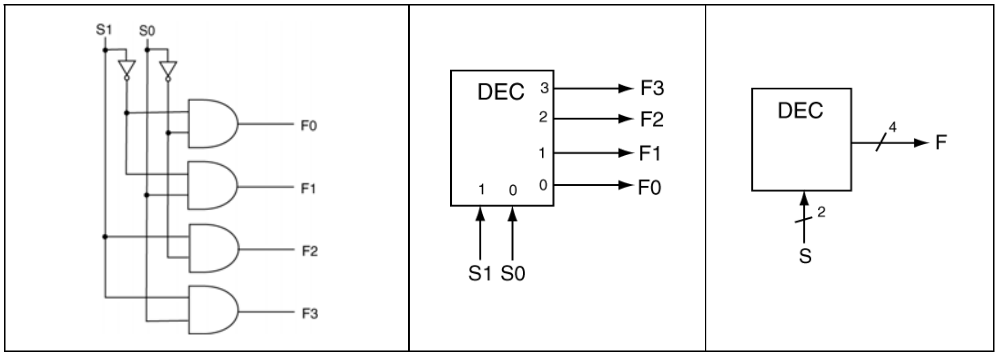
**Figure 16.4: Killing the AND (a) and OR (b) gates.**

While generic decoders have an unspecified number of inputs and outputs, standard decoders have a “standard” relationship between the number of inputs and outputs, as well as the form of the outputs. When you hear the word “decoder”, it does not refer to a specific type of input/output relationship for the circuit. As a result, we choose to model decoders are either generic or “standard” decoders. When you hear decoder, you don’t know much about the circuit; if you hear “standard decoder”, you know something about the circuit.

The standard decoder fixes the relationship between the number and form of circuit inputs and outputs. Figure 16.5(a) shows a gate-level model of a 2:4 standard decoder. Due to the configurations of the inputs **S1** and **S0** in Figure 16.5(a), only one of the AND gates is non-dead at a given time. Thus at any given instance in, only one of the outputs **F0**, **F1**, **F2** or **F3** is a ‘1’, while the three others are ‘0’. The condition that makes this a standard decoder is the relationship between the number and form of the inputs and outputs. The bulleted items below highlights these main attributes:

- Standard decoders always have a binary-type relationship between the number of inputs and outputs. For example, standard decoders come in flavors such as 1:2, 2:4, 3:8, 4:16, etc. This progression has an n:$2^n$ relationship. The first digit refers to the number of inputs to the circuit (control variables) while the second variable refers to the number of circuit data outputs. The “n” input variables can reference $2^n$ unique output combinations.
- Although the schematic diagram of circuit of Figure 16.5(b) is adequate to describe a standard decoder, the schematic diagram of Figure 16.5(c) is more common. The small numbers associated the circuit inputs and outputs in Figure 16.5(b) indicate a weighting on those inputs and outputs. You must attach these numbers unless you use the bundle notation in Figure 16.5(c).

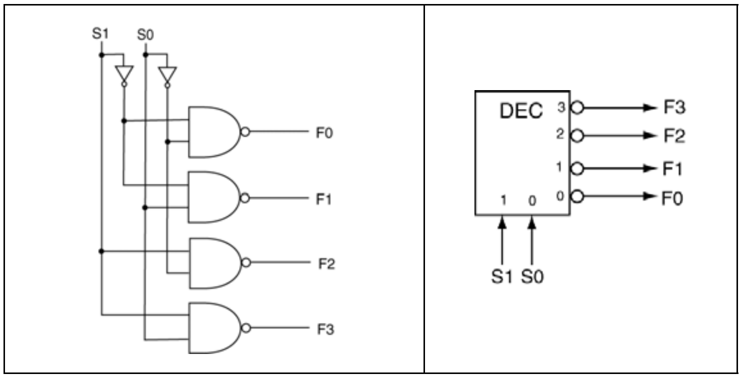
- Only one output of the standard decoder is active at a given time because we configure the control variables such that only one of the internal AND gates is non-dead. All of the outputs except one are high at a given time while the other output is low. The 2:4 decoder has four possible output combinations: “0001”, “0010”, “0100”, “1000”, which is a one-hot code.



(a) (b) (c)

**Figure 16.5: A standard 2:4 decoder in schematic and circuit forms.**

Figure 16.6 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 16.6(b) are roughly the same bubbles on the NAND gates. This version of the 2:4 decoder has four possible output combinations: “1110”, “1101”, “1011”, “0111”. We refer to this as a “one-cold” code; an ingenious name.

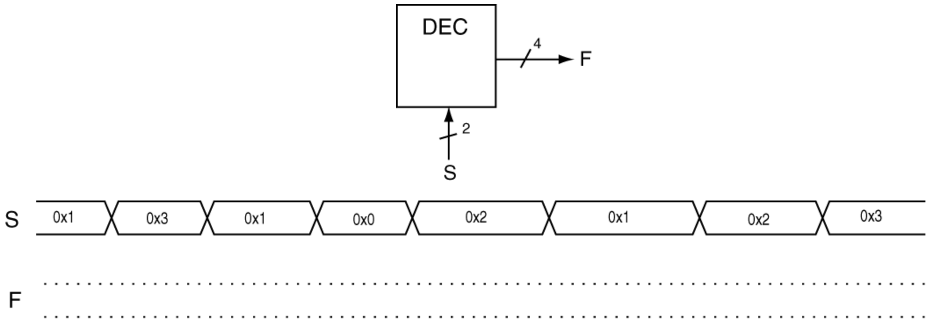


(a) (b)

**Figure 16.6: A standard 2:4 decoder with inverted outputs.**

**Example 16.1: Timing Diagram for a Standard Decoder**

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 standard decoder, the **S** input must be the selector inputs while the **F** are the outputs. The selection input bundle is two bits wide, which enables it to select one of four different possible outputs. The problem description uses bundle notation in order to simplify the problem.

There are only two selector bits, but the solution uses hex notation; the unused bits are all zero. For example, the hex value of “0x3” represents “0011”, but the first two bits do not affect the output selection; this problem uses only the two lower bits of the hex notation.

Figure 16.7 shows the final solution to this example; the solution opts to use hex notation. The solution has a nice binary relationship between the selector inputs and the outputs. Figure 16.8 shows an alternate solution to this example, which clearly shows that only one output is a ‘1’ at any given time.

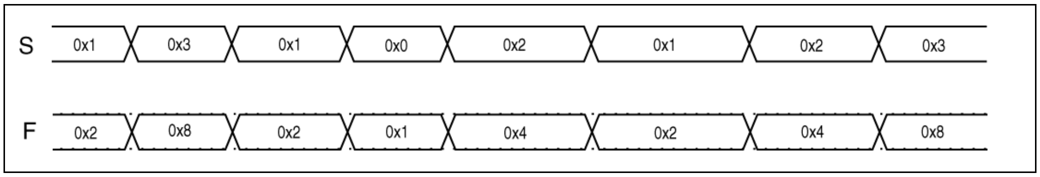


**Figure 16.7: The solution to this example.**

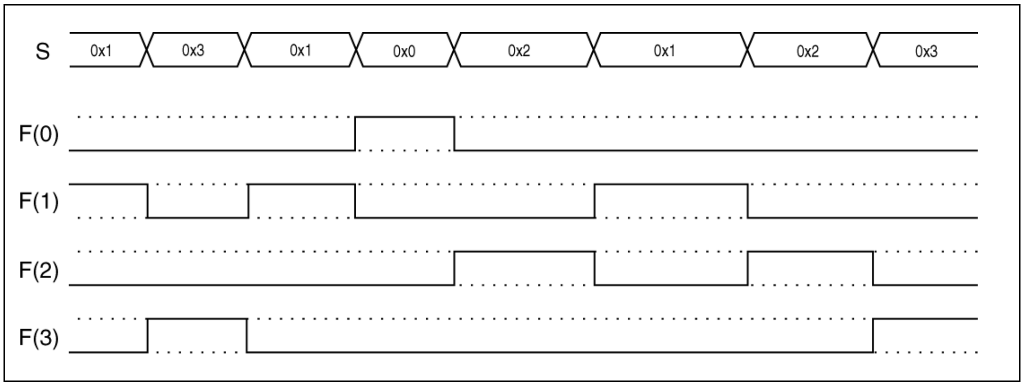
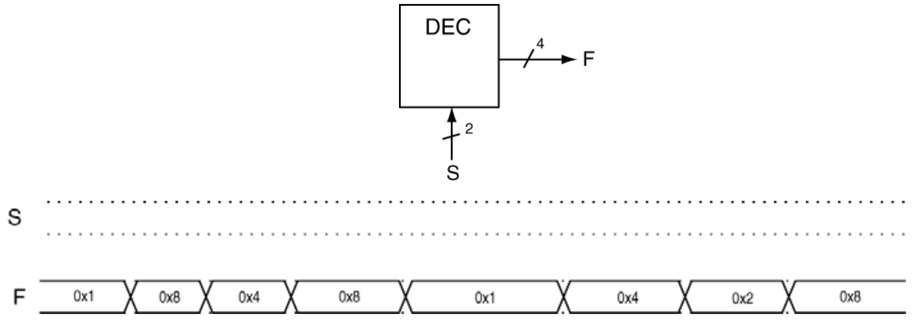


**Figure 16.8: An alternate solution to for this example.**

**Example 16.2: Timing Diagram for a Standard Decoder**

Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



**Solution**: We start this problem by knowing what the output values are. We need to 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. This solution also uses bundle notation. Fun stuff.

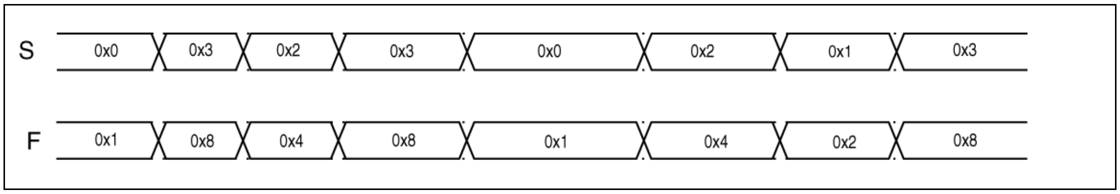
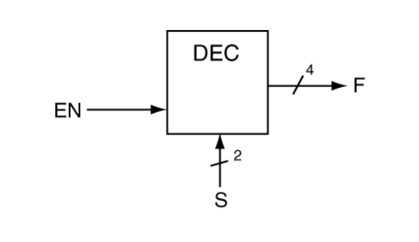


**Figure 16.9: The solution to this problem.**

**Example 16.3: Standard Decoder with Enable Input**

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**: Standard decoders often include more than the standard control inputs. One of the typical controls on the decoder’s inputs is an enable signal. Once standard decoders add more input control signals, the underlying circuitry becomes more complicated and not worth drawing. We provide a table that describes the behavior of such a circuit. Figure 16.10(a) and Figure 16.10(b) show a schematic symbol and a table describing the operation of a standard decoder with an enable input, 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 16.10: A 2:4 decoder with an enable (a) and its behavior described in tablature format (b).**

Figure 16.11 shows a timing diagram that describes the behavior of the circuit described in this example. There are two main features worth noting in this timing diagram.

- 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.

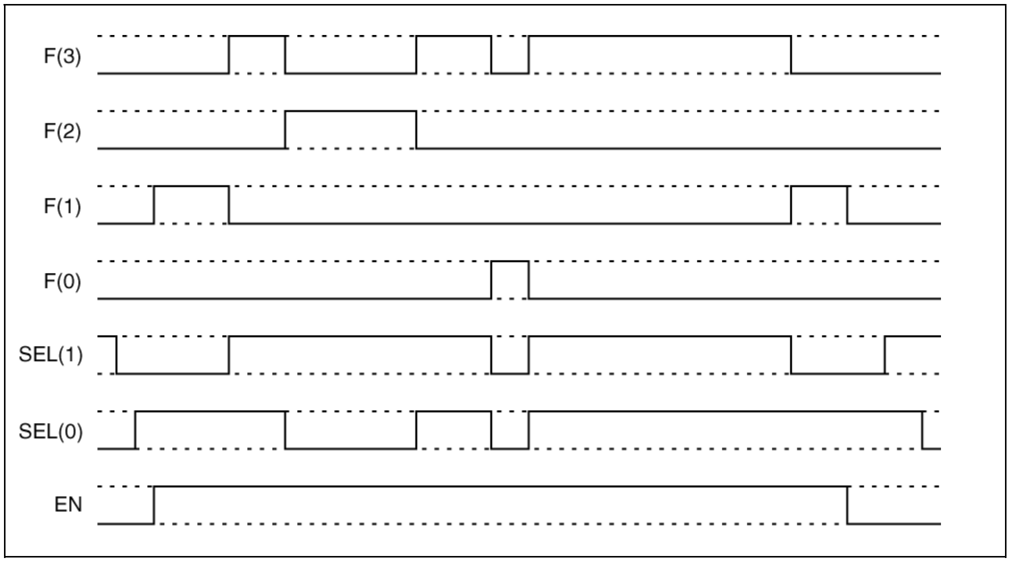


**Figure 16.11: An example timing diagram for this example.**

## 16.5 Digital Design Foundation Notation: Generic Decoder

We consider the generic decoder to be one of our Digital Design Foundation circuits. We consider the generic decoder to be a controlled circuit; Figure 16.12 shows the generic decoder in appropriate foundation notation. The generic decoder models a table, so the **DATA_IN** inputs act as the independent variables and the **DATA_OUT** signals are the dependent variables. We consider the generic decoder does not have either control inputs or status outputs. Table 16.1 provides a description of all the inputs and outputs to the generic decoder.

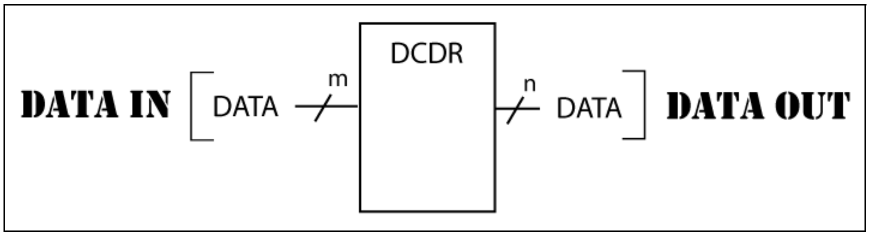


Figure 16.12: Data signals for a generic decoder.

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **DATA** | The independent variable of the look-up-table |
| OUTPUT DATA | **DATA** | The dependent variable of the look-up-table |
| CONTROL | **n/a** | - |
| STATUS | **n/a** | - |

Table 16.1: The foundation matrix for a generic decoder.

## 16.6 Digital Design Foundation Notation: Standard Decoder

We consider the generic decoder to be one of our Digital Design Foundation circuits. We consider the standard decoder to be a controlled circuit; Figure 16.12 shows the standard decoder in appropriate foundation notation. The standard decoder has no data inputs; the only inputs are the **SEL** inputs, which decide the exact format of the **DATA_OUT** signals. By definition, the **DATA_OUT** signals form a one-hot code. Table 16.2 provides a description of all the inputs and outputs to the standard decoder.

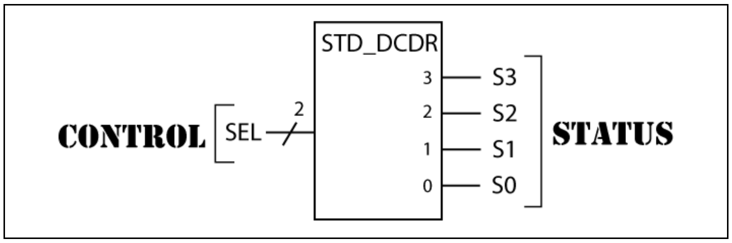


Figure 16.13: Control and status signals for a 2:4 standard decoder.

| | Signal Name | Description |
|---|---|---|
| **INPUT DATA** | **n/a** | - |
| **OUTPUT DATA** | **n/a** | - |
| **CONTROL** | **SEL** | The inputs that select the desired form of the output. |
| **STATUS** | **S(3:0)** | The output signals chosen by the **SEL** input. |

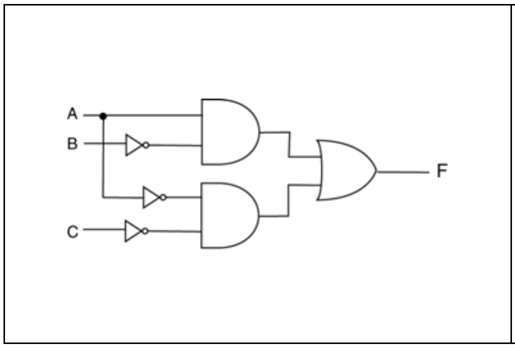
**Table 16.2: The foundation matrix for a standard decoder.**

## 16.7 Chapter Summary

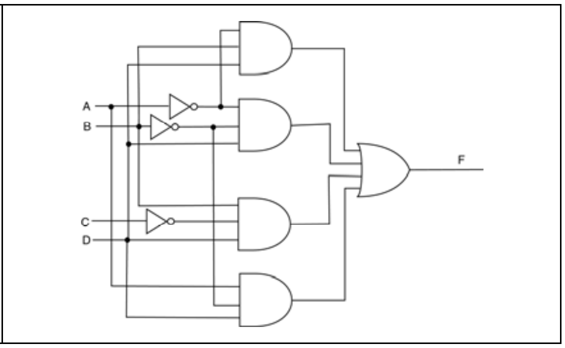
- The official definition of a decoder: combinatorial (or non-sequential) digital device that establishes a functional relationship between the device input(s) and output(s). This definition defines a generic decoder, which is not to be confused with the standard decoder. We typically refer to any circuit we can model using a table (such as a truth table) a decoder.
- Standard decoders are a special type of decoder. The inputs and outputs of the standard decoder exhibit an n:$2^n$ relationship. In particular, if a standard decoder has n inputs, it necessarily has $2^n$ outputs. We often use standard decoders in conjunction with hardware designed to access memory.
- **Mealy's Seventh Law of Digital Design**: Always first consider modeling a digital circuit using some type of a look-up table (LUT).

## 16.8 Chapter Exercises

**1)** Implement the following functions using a generic decoder.

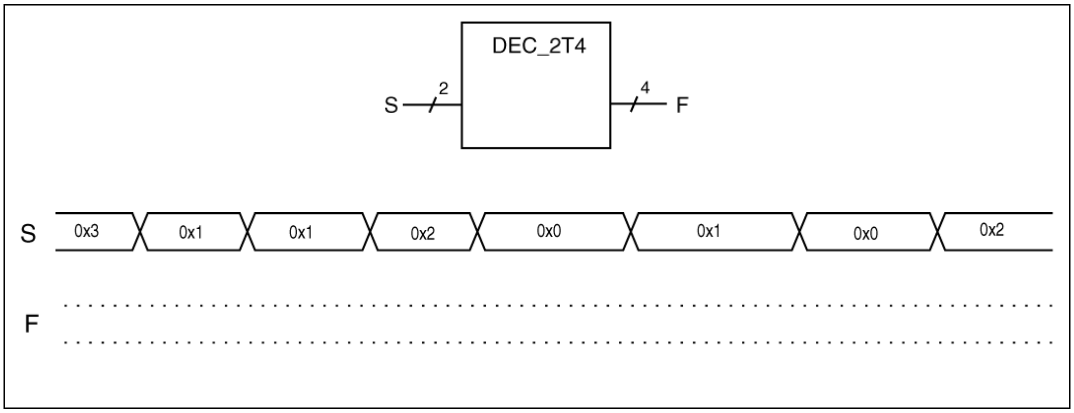


**(a)**

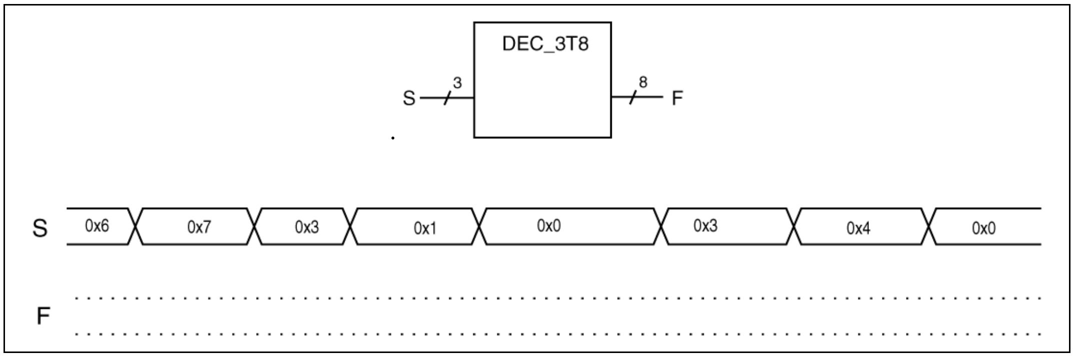


**(b)**

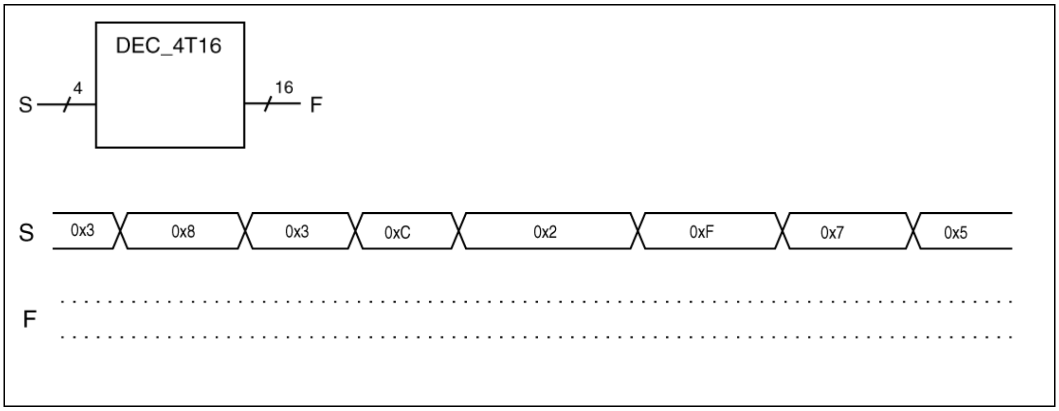
**2)** Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.



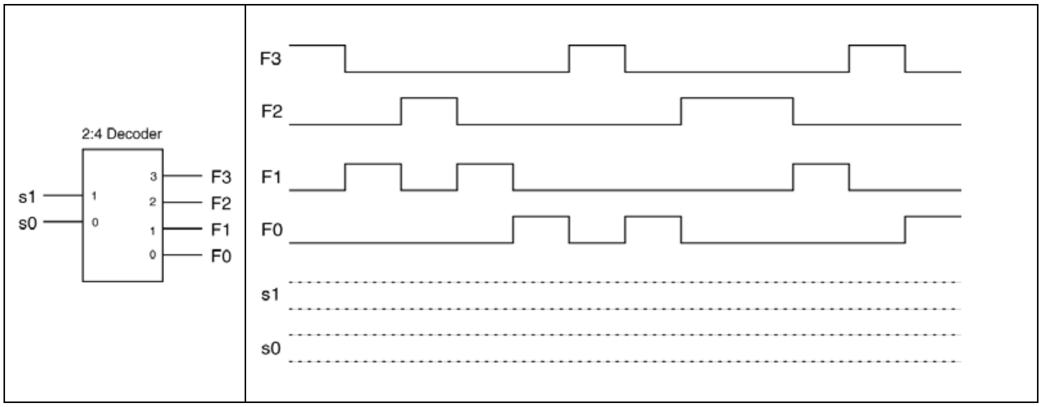
**3)** Use the following black box model for a standard 2:4 decoder to complete the following timing diagram



**4)** Use the following black box model for a standard 2:4 decoder to complete the following timing diagram.

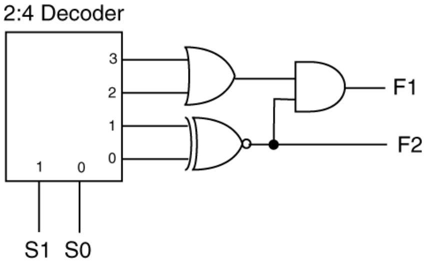


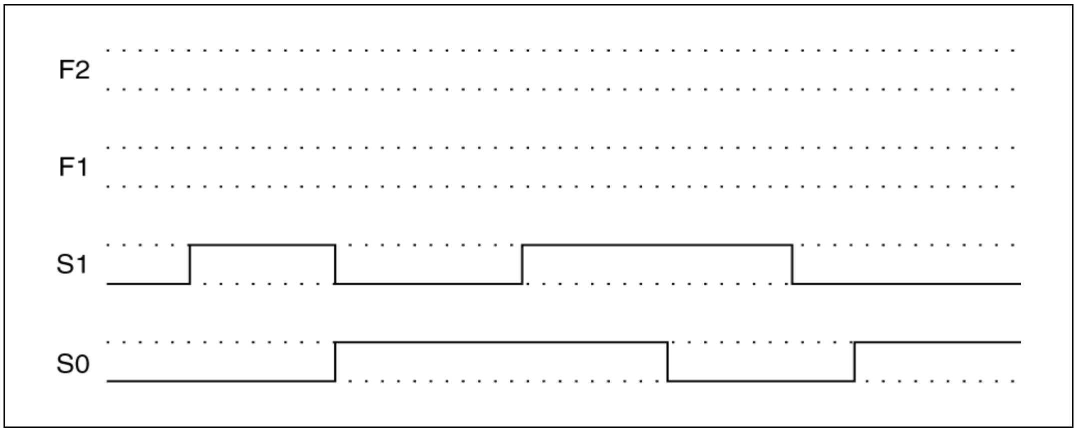
**5)** Based on the standard 2:4 Decoder below, complete the following timing diagram by entering the values for signals s1 and s2 that would generate the listed output waveforms. Assume that propagation delays are negligible. Be sure to annotate you solution to this problem.



**6)** Briefly describe the differences between a generic and standard decoder.

**7)** Use the schematic diagram to complete the F2 and F1 outputs of the provided timing diagram. Consider the decoder to be a standard 2:4 decoder. Assume that propagation delays are too small to worry about.





## 16.9 Design Problems

For the following problems:

- Use some type of decoder in your design; you can use other foundation modules when appropriate, but minimize your use of simple logic gates in favor of decoders.
- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**1)** Design a 2's complement validity checking circuit for an RCA module. The single output of the circuit indicates when the result of the addition is valid.

**2)** Design a BCD-to-7 segment decoder. The circuit converts a number in BCD format to a form that can be used to indicate that number on a 7-segment display.

**3)** Design a BCD-to-7 segment decoder. The circuit converts a number in BCD format to a form that can be used to indicate that number on a 7-segment display and look like proper decimal digits when read in a mirror.

**4)** 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.

**5)** Design a circuit that has four inputs and two outputs. One output indicates when the four inputs (considered a binary number) are even and less than 8; the other output indicates when the four input bits are odd and greater than 10.

**6)** 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 $111_2$.

**7)** Design a circuit that has four inputs and three outputs. The four inputs are considered 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".

**8)** Design a circuit that has an output that indicates when the four-bit binary number on the input is a prime number. For this problem:

- assume an input value of "0000" never occurs (be sure to note this fact where appropriate)
- assume the decimal value of 1 is a prime number

**9)** Design a circuit whose unsigned binary output represent the square of the circuit's 4-bit unsigned binary input.

**10)** Design a circuit whose outputs represent the square root of the circuit's 4-bit input. Round the output either up or down when necessary.

**11)** 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, the output should represent the input rounded down to the nearest integer.

**12)** Design a circuit whose unsigned binary output represent the square of the circuit's 4-bit signed binary input (RC format).

**13)** Design a circuit that converts a three-digit decimal number to an 8-bit unsigned binary number. This circuit has three BCD inputs, which means four bits for the 100's, 10's, and 1's digit. The 8-bit output will always be sufficient to encode the three digital input value.

**14)** Design a circuit in with the following specifications: if the RND input is asserted, the 24-bit input is rounded down to the nearest multiple of 8; otherwise the input is passed through unchanged to the output.

**15)** 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: $0.1_2 = 0.5_{10}$.

# 17 Multiplexors

## 17.1 Introduction

Our approach to digital design has been somewhat limited because we are missing one of the most basic modules in digital design: the multiplexor. This chapter introduces the multiplexor at a low level, then quickly abstracts to the module-level in order to retain the simplicity of the device's operation.

**Main Chapter Topics**

**DIGITAL DESIGN FOUNDATION 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 increase in complexity.

**Chapter Acquired Skills**

- Be able to describe the basic operation of a multiplexor
- Be able to use multiplexors in digital designs
- Be able to describe the underlying hardware of a simple multiplexor

## 17.2 Making Decisions in Hardware and Software

Because computer programs generally "react" to various things, there are programming constructs that handle these reactions. The general notion in programming is that there is one processor and this processor does one thing at a time[1]. Computer programs make decisions based on the current conditions in a program: the program either executes one set of instructions or "decides" to execute another set of instructions. A "conditional statement" is the mechanism the program uses to choose one path of execution over another.

Hardware design is similar to software design. Your hardware must be able to react to certain conditions in the circuit and choose one "result" over another "result" based on those conditions. A multiplexor allows the hardware to choose one thing over another thing in digital circuits. There is a huge difference between decisions making in software vs. decision making in hardware. The notion in software design is that the computer program chooses one path of execution over another path; it would be inefficient to "execute" both paths and then choose the result of interest.

The problem arises in hardware when you choose between two "results“. In hardware, the circuitry generates all options in parallel (or concurrently). The multiplexor inputs all the options, and then allows a signal to choose which of the inputs appears on the output of the multiplexor. If you need to choose between two different results, the hardware generates both results and then chooses the desired result based on some signal (condition) in the circuit. Therefore, when you're designing "choosing" operations in hardware, you must generate every possible "desired result" and then choose the result you need.

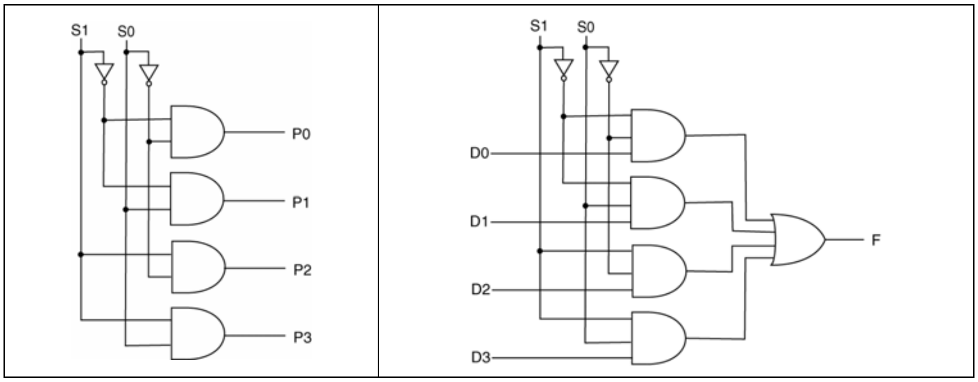
[1] Generally speaking, a processor executes one instruction at a time; it executes one instruction and then moves on to the next instruction.

## 17.3 Multiplexors

The multiplexor is another Digital Design Foundation module. When you hear the word multiplexor, or MUX as most people refer to it[2], you need to think “selector circuit”. A MUX is a generally a circuit with many inputs and one output; the single output of the device represents a direct transfer of one of the inputs to the output under direction of the MUX’s control input.

The first step in developing the MUX is the selection circuitry in Figure 17.1(a), which you should recognize as a standard decoder. Two variables **S1** and **S0** serve as selection variables, so only one of the **Px** outputs is a ‘1’ at any given time . In Figure 17.1(a), three of the AND gates are dead, which officially creates a one-hot code on the Px outputs.

Figure 17.1(b) shows the final portion of the MUX circuitry. Knowing that three of the AND gates are dead, the only way that the circuit output **F** can be a ‘1’ is if the **Dx** input on the un-dead AND gate is a ‘1’. If the **D** input on the non-dead gate is a ‘0’, all of the AND gates are dead and the **F** output is a ‘0’. If however, the **D** input on the non-dead gate is a ‘1’, then the non-dead AND gate output is a ‘1’, the OR gate has an input of ‘1’, so the OR gate output **F** is ‘1’. The circuit in Figure 17.1(b) effectively transfers the value of one **D** input to the output **F**. The MUX thus selects one of the **D** inputs to appear on the **F** output. The **D** input that appears on the **F** output depends upon which AND gate is un-dead, which depends on the values of the **S1** and **S0** data selection inputs.



**(a)** **(b)**

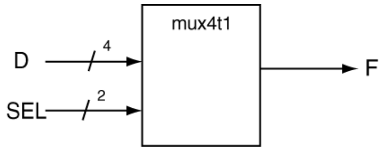
**Figure 17.1: The MUX input circuitry (a) and the complete MUX (b).**

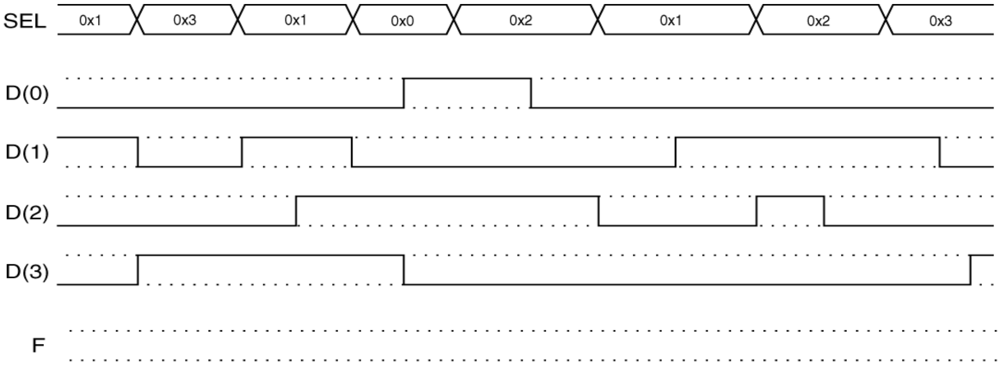
We refer to the MUX in Figure 17.1(b) as a 4:1 MUX because the device chooses between one of four inputs to appear on the single output. MUXes generally have a binary relationship between the number of selection variables and the number of data inputs; common flavors of MUXes include 2:1, 4:1, 8:1, 16:1 etc. This is the most basic form of a MUX. In reality, MUXes come in many different flavors and quickly become complex enough that you’ll avoid modeling them on the gate-level.

[2] And for the record, the correct pronunciation is “mucks” and not “mooks”.

**Example 17.1: 4:1 MUX Timing Diagram**

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 matches one of the four data inputs depending upon which input the selector inputs select. The **SEL** input is in bundle notation while we expand the **D** input bundled to make the problem clearer. Figure 17.2 shows the solution to this example.

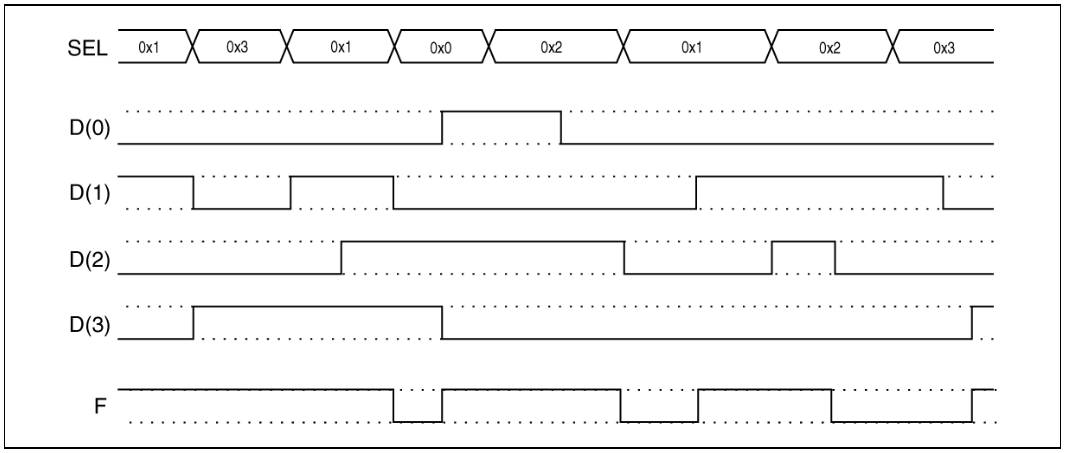
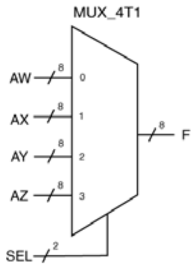
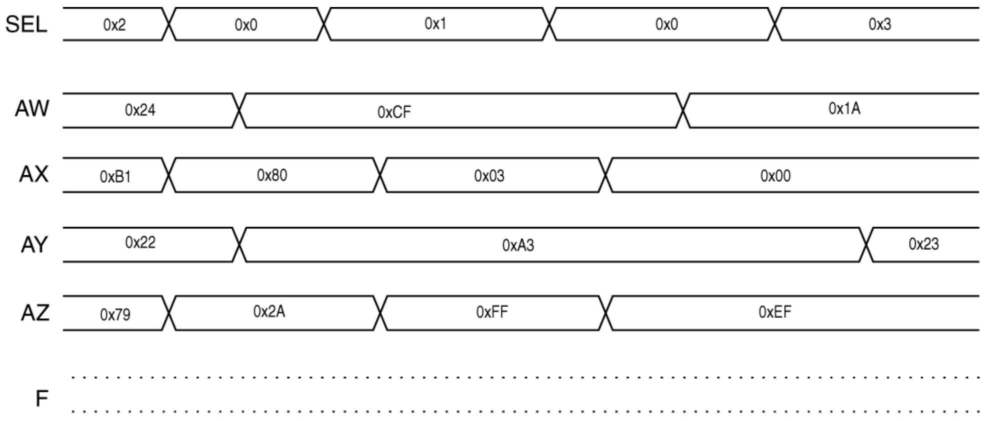


**Figure 17.2: The solution to** Example 17.1**.**

**Example 17-2: 4:1 Bundle-Based MUX Timing Diagram**

Using the following diagram of a 4:1 MUX, complete the provided timing diagram.





**Solution**: First, the schematic uses a new and distinctive shape for the MUX. Circuit diagrams always use this shape to represent MUXes as the shape transfers information to the person reading the diagram. When you see this shape in a schematic, you'll immediately know the purpose of the device: choosing which input appears on the output.

It is important that the MUX's data inputs also contain indexing numbers. The numbers associated with the data inputs range from [0,3], which by design corresponds to the numbers that you can represent by the two control inputs. If the data inputs are not numbered, you'll not know which input appears on the output. Figure 17.3 shows the solution to this example. Using vertical dotted lines helps you generate the solution.

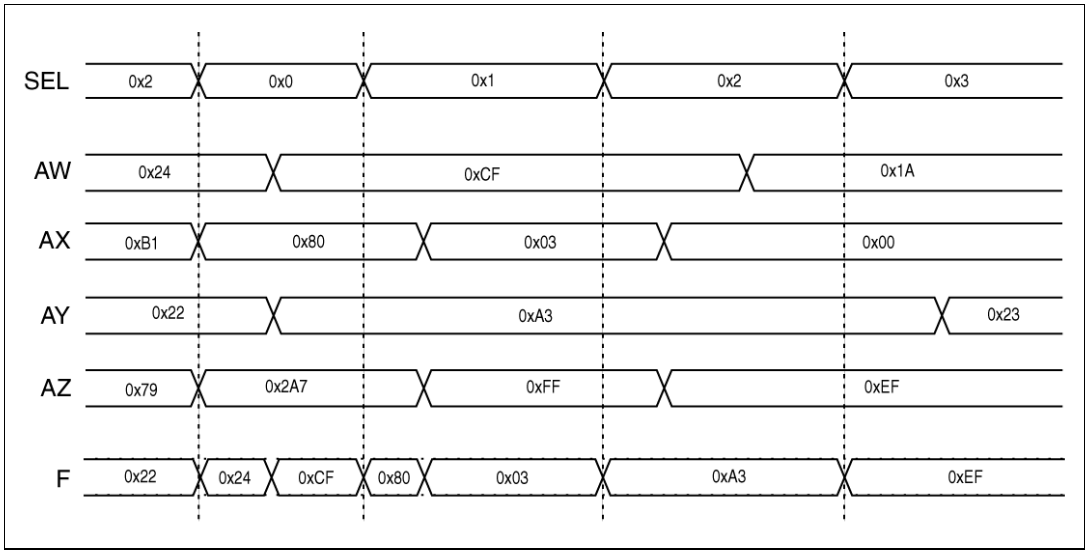


**Figure 17.3: The solution to this example.**

**Example 17.3: Selection Circuit #1**

Design a circuit that has three 8-bit unsigned binary inputs **A**, **B**, and **C**. The circuit outputs **A + B** if the circuit's button inputs is asserted (**BTN**='1'); otherwise, the circuit outputs **A + C**. The circuit's output is also an 8-bit unsigned binary value. Don't worry about the validity of the sum outputs. Provide the top two levels of BBDs for this problem. Use no more than one RCA in your design, but in general, minimize your use of hardware for this design. Also, state what is controlling this circuit.

**Solution**: The first step is drawing a block diagram of the final circuit as we show in Figure 17.4.

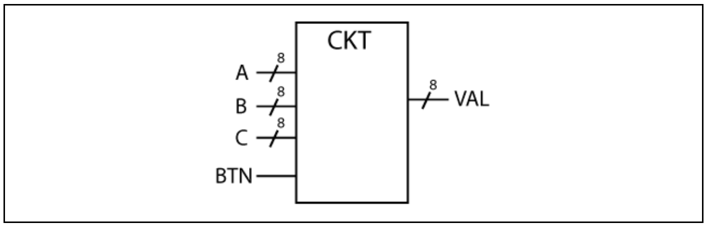


**Figure 17.4: Top-level BBD for the solution.**

The next step is to make an inventory of modules this circuit requires to solve the problem. The problem states not to use more than one RCA, so you know there is an RCA. This means that we must configure the RCA to do both of the addition operations. The problem states that we need to add **A** to either **B** or **C** dependent upon a button press; this means there is a "selection" happening in the circuit. Anytime a circuit is "selecting" something, the circuit requires a MUX . Examining the two summing operations shows that we're always adding **A**; the item we need to select is the value we're adding, which is either **B** or **C**. This means the inputs to the MUX are **B** & **C**, the output of the MUX becomes the second input to the RCA, and the circuit's **BTN** input connects to the MUX control. Figure 17.5 show the lower-level BBD for this problem.

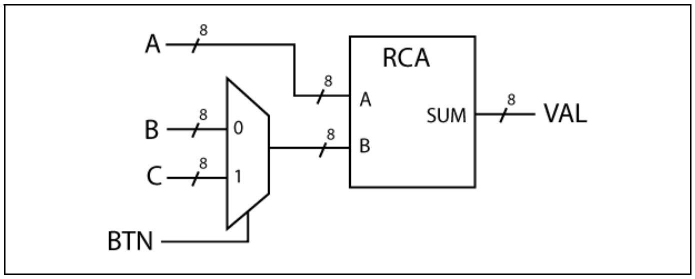


**Figure 17.5: The final circuit solution for this example.**

The circuit in Figure 17.5 has external control. The MUX contains a control input, which connects to a signal external to the circuit.

We could have done this problem using two RCAs. In this case, each RCA would be responsible for the one of the two addition operations; the MUX would then choose between the desired sums for the circuit's outputs. This solution would be less desirable than our solution because the circuit requires two RCAs, whereas our solution requires one RCA.

**Example 17.4: Selection Circuit #2**

Design a circuit that has three 8-bit unsigned binary inputs **A**, **B**, and **C**. The circuit outputs **A + B** if that addition operation does not generate a carry; otherwise the circuit outputs **A+ C**. The circuit's output is also an 8-bit unsigned binary value. Don't worry about the validity of the sum outputs. Provide the top two levels of BBDs for this problem. Minimize your use of hardware for this design. Also, state what is controlling this circuit.

**Solution**: The first step is drawing a block diagram of the final circuit as we show in Figure 17.6.

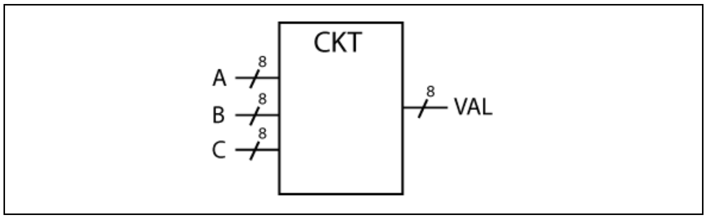


**Figure 17.6: Top-level BBD for the solution.**

The next step is to make an inventory of modules this circuit requires to solve the problem. The problem states that we need to output the result of one of two additions. Unlike the previous example, this problem does not constrain us to using only one RCA, which is fortunate because we could not solve the problem otherwise. The condition that needs to select the output is dependent upon one of the addition operations, which means the **Co** from that particular RCA chooses either the **A + B** or **A +C** results. Because something in this circuit is being "chosen", our circuit also requires a MUX. The MUX in this problem chooses the outputs of one of two RCAs to appear on the circuit's output. Figure 17.7 shows the final lower-level BBD for this problem.

The MUX in this circuit has a control input, which connects to the **Co** output of the one of the circuit's RCAs. Because of this internal connection, this circuit has internal control.

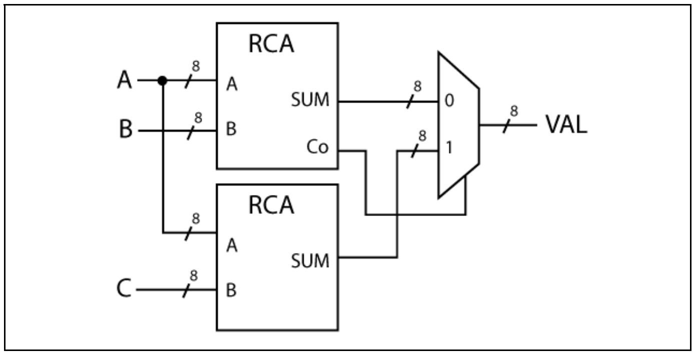


**Figure 17.7: The final circuit solution for this example.**

**Example 17.5: Selection Circuit #3**

Design a circuit that has three 8-bit unsigned binary inputs **A**, **B**, and **C**. The circuit outputs **A + B** if the circuit's button inputs is asserted (**BTN**='1'); otherwise, the circuit outputs **A + C**. The circuit's output is also an 8-bit unsigned binary value. If the summation generates a carry-out, then the circuit outputs zero. Provide the top two levels of BBDs for this problem. Minimize your use of hardware for this design. Also, state what is controlling this circuit.

**Solution**: The first step is drawing a block diagram of the final circuit as we show in Figure 17.8.

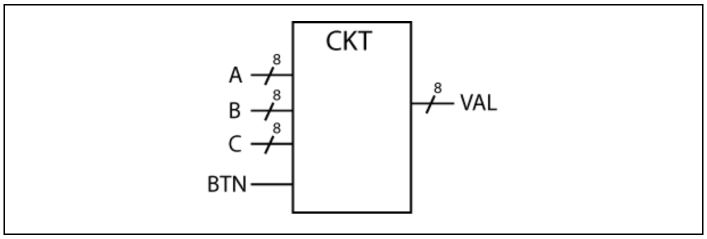


**Figure 17.8: Top-level BBD for this problem.**

The next step is to make an inventory of modules this circuit requires to solve the problem. This problem is a combination of the previous two problem, as we're both choosing the addition operation and we're choosing between the result of the chosen operation or zero to appear on the circuit output based on whether the circuit generated a carry-out or not. This means we only need one RCA; we use a MUX to choose the addition operation, which the **BTN** input controls. We also require a second MUX to choose between the result of the addition operation or zero, which depends upon whether the chosen addition operation generated a carry or not. Figure 17.9 shows the lower-level BBD for our solution.

This circuit has two MUXes; an external input controls one MUX while an internal input controls the other MUX. This circuit thus has both external and internal control.

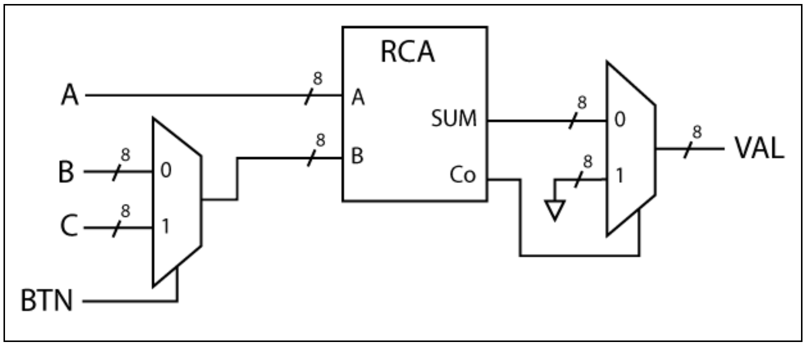


**Figure 17.9: The final circuit solution for this example.**

## 17.4 Digital Design Foundation Notation: MUX

We consider the MUX to be one of our Digital Design Foundation circuits. The MUX is a controlled circuit; Figure 17.10 shows the MUX in appropriate foundation notation. The SEL signal is a control input and decides which DATA_IN signal becomes the DATA_OUT signal. The MUX thus has a control input but has no status outputs. Table 17.1 provides a description of the MUX's inputs and outputs.

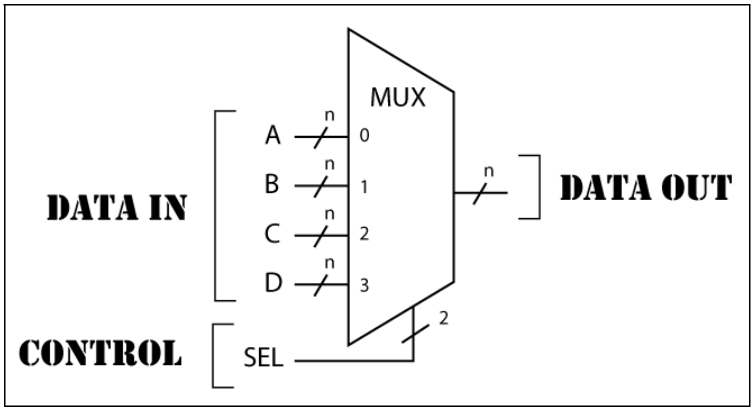


**Figure 17.10: Data and control signals for a 4:1 MUX.**

| | Signal Name | Description |
|---|---|---|
| **INPUT DATA** | **A, B, C, D** | Data inputs to the MUX; MUXes can have any number of data inputs. One of these data inputs becomes the single data output. |
| **OUTPUT DATA** | **F** | A single output, which is one of the inputs as selected by the **SEL** signal. |
| **CONTROL** | **SEL** | Selects which data input appears on **F**. The width of the **SEL** signal is such that $2^{SEL} \geq$ to the number of data inputs. |
| **STATUS** | **n/a** | - |

**Table 17.1: The foundation matrix for a MUX.**

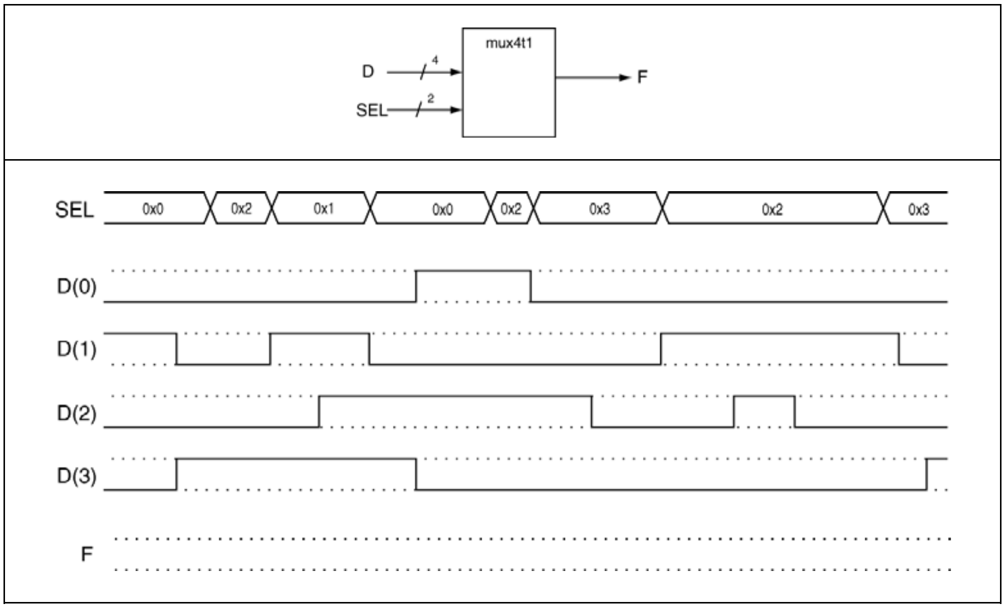
## 17.5 Chapter Summary

- 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.
- 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.
- 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.

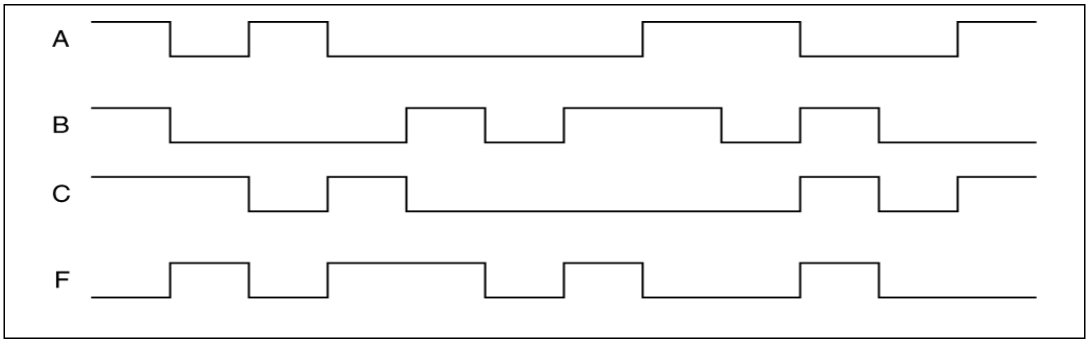
## 17.6 Chapter Exercises

**1)** Briefly describe the special relationship between a MUX and a standard decoder.

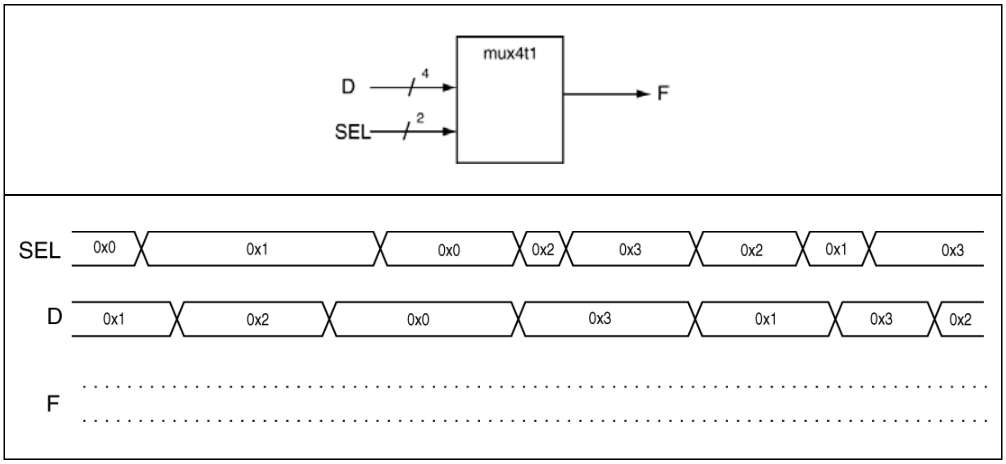
**2)** Use the following block diagram to complete the provided timing diagram. For this problem, consider the block diagram to represent a basic 4:1 MUX.



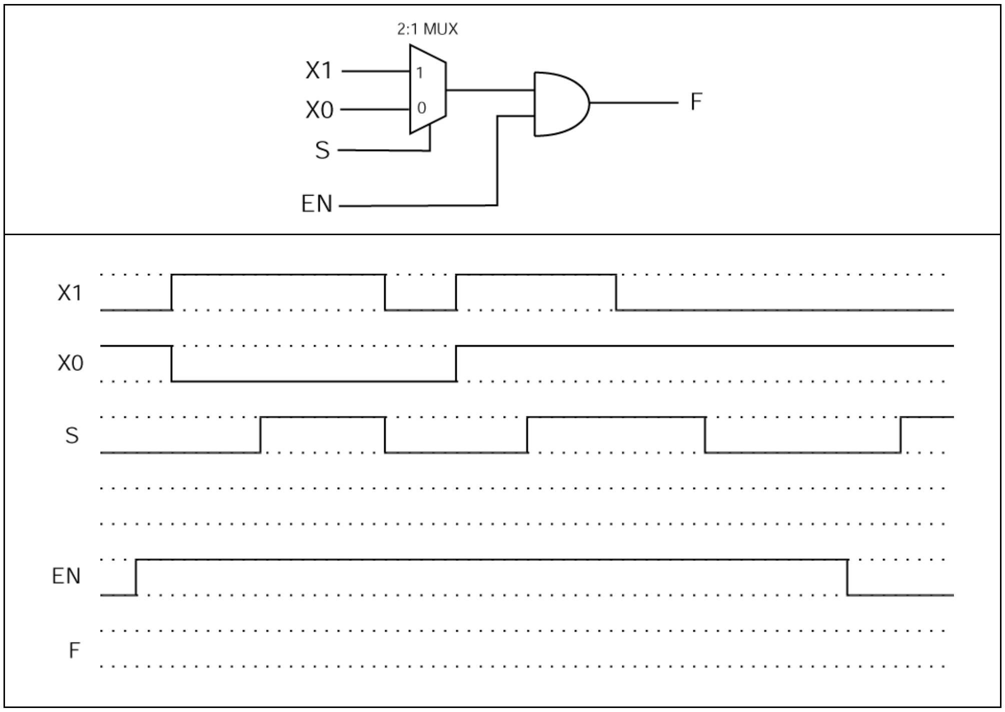
**3)** The following timing diagram completely defines a function F(A,B,C) that has been implemented on an 8:1 MUX. The control variables are A, B, and C (A is the most significant bit and C is the least significant bit) and the output is F. Write an expression for this function in reduced NAND/NAND form. Assume propagation delays are negligible.



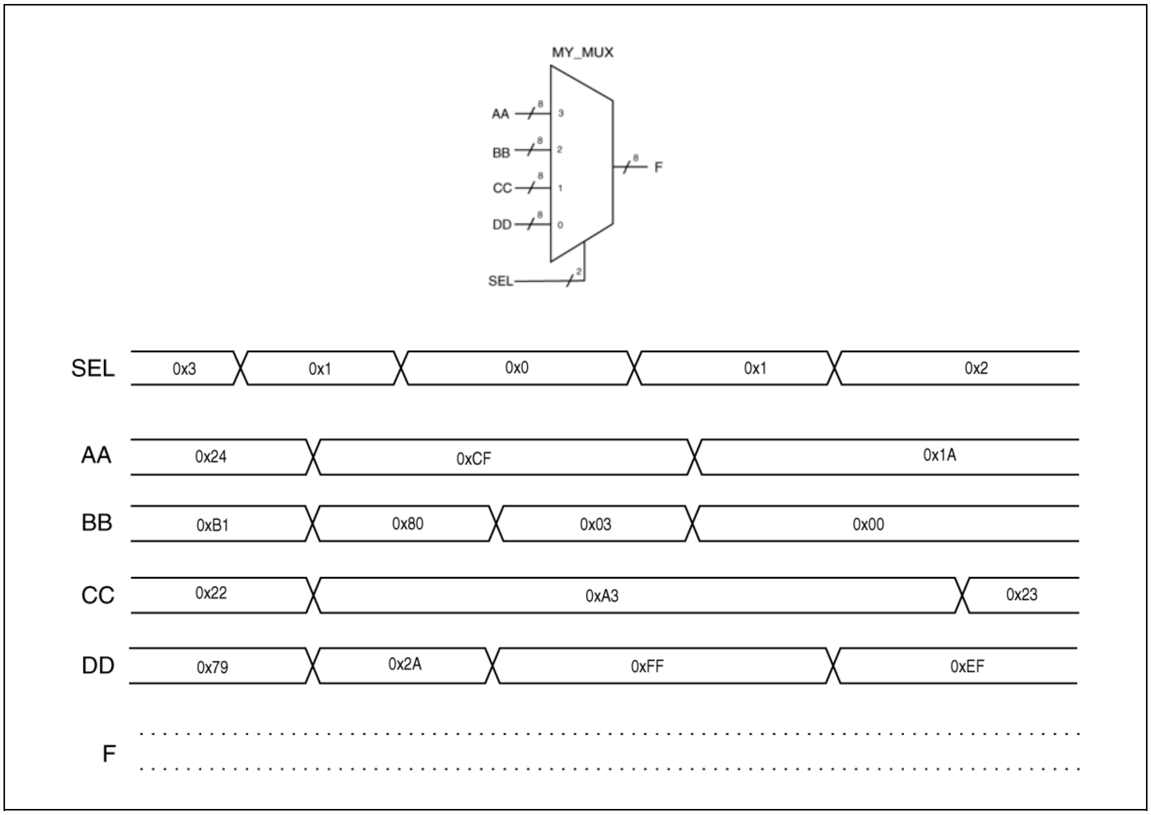
**4)** Use the following block diagram to complete the provided timing diagram. For this problem, consider the block diagram to represent a basic 4:1 MUX.



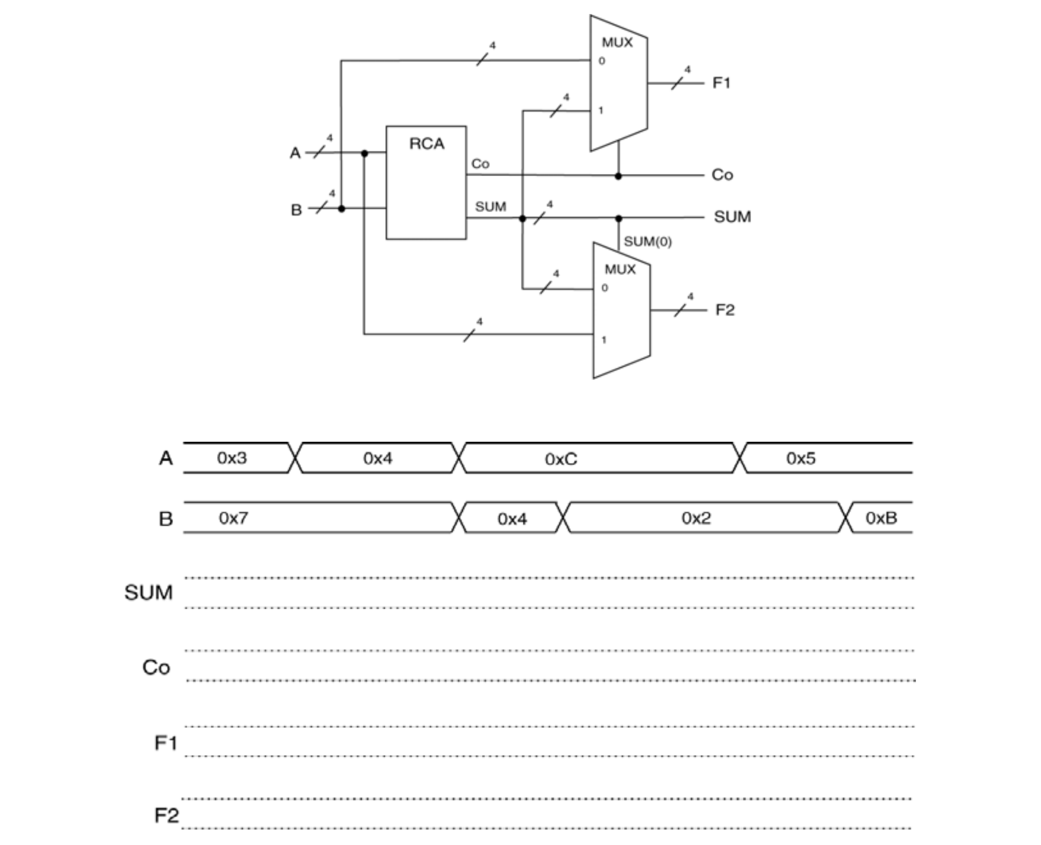
**5)** Use the listed circuit to complete signal F in the following timing diagram.



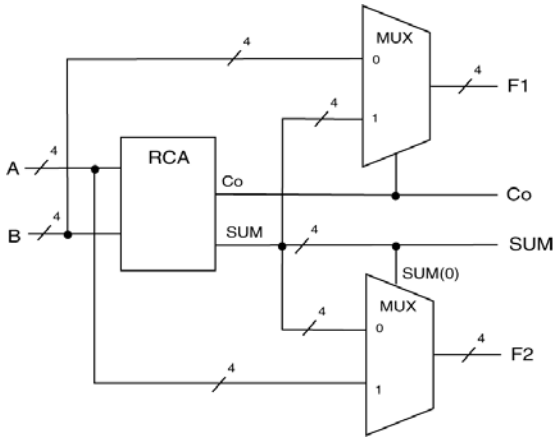
**6)** Using the following diagram of a 4:1 MUX, complete the provided timing diagram.

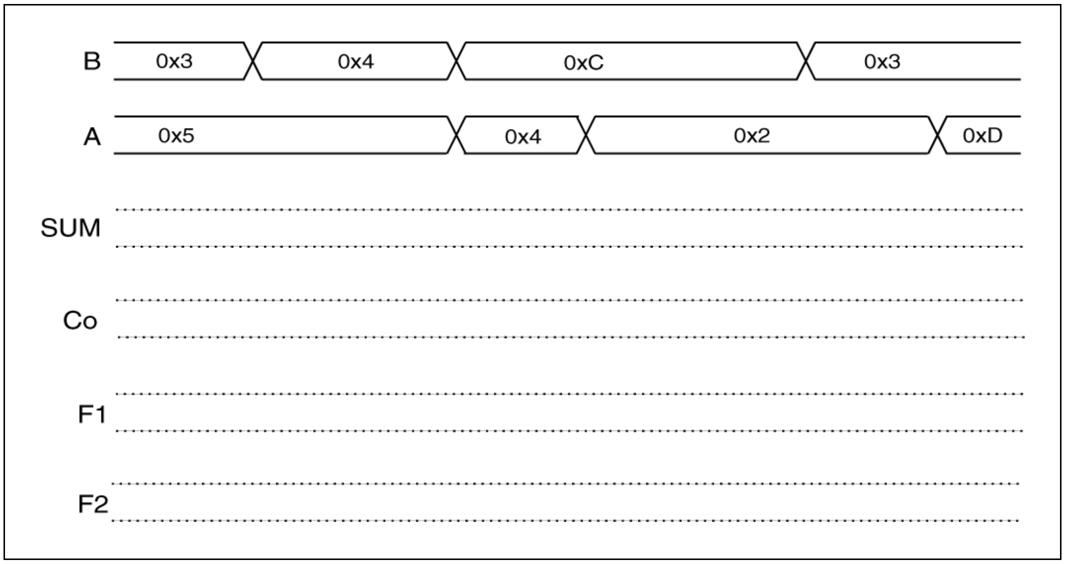


**7)** Use the following circuit diagram to complete the empty rows on the accompanying timing diagram. Use bus notation for all bundles (Co is the only non-bundle signal; 0x indicates hexadecimal).



**8)** Use the following circuit diagram to complete the empty rows on the accompanying timing diagram. Use bus notation for all bundles. Assume the inputs and outputs are unsigned binary.





**9)** Briefly describe the special relationship between a MUX and a standard decoder.

## 17.7 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- Use only digital design foundation modules in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**1)** Design a circuit that translates an 8-bit number in signed magnitude form to an 8-bit number in diminished radix complement form.

**2)** Design a circuit that translates an 8-bit number in diminished radix complement form to an 8-bit number in signed magnitude form.

**3)** 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.

**4)** 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.

**5)** Design a circuit that adds two 8-bit values. If the summation generates a carry-out, the output then displays all zeros; otherwise, the output displays the 8-bit summation result.

**6)** Design a circuit that calculates the absolute value of a 10-bit signed binary number in RC form.

**7)** Design a circuit that provides the absolute value for an 8-bit signed binary number. Assume the number is in sign magnitude form.

**8)** Design a circuit that provides the absolute value for an 8-bit signed binary number. Assume the number is in diminished radix complement form.

**9)** Design a circuit that performs the following operation: if a button is pressed, the circuit outputs the results of A+B; otherwise the circuit outputs the value of A-B. Consider the values of A & B to be 8-bit signed binary values in RC form. This problem does not check the validity of the result. For this problem, assume the pressed button generates a '1'.

**10)** Design a circuit the checks the validity of a the result of an addition operation on two 10-bit signed binary numbers in RC form.

**11)** Design a circuit that can add two numbers according to the following specifications. If the sum of A & B generates a carry, the circuit outputs A; otherwise the circuit outputs the sum of A & B. Assume A, B, & the sum are unsigned 8-bit values.

**12)** Design a circuit that outputs the sum of A & B if both A & B are both less than 128; otherwise the circuit outputs B. Assume A, B, & the sum are unsigned 8-bit values. No need to deal with carry generation.

**13)** Design a circuit that outputs the sum of A & B if and only if one of the operands is negative; otherwise the circuit outputs A-B. Assume A, B, and the result are all signed 8-bit binary numbers in RC (2's complement) format. If the result is not valid, the circuit output zero and turns on an LED.

**14)** Design a circuit that outputs the result of the following operation if the operation does not generate a carry: A+B+C (addition); otherwise, the circuit should output C. The circuit has an extra output that indicates which result is being output. For this problem, assume A, B, C, & the result are al 12-bit unsigned binary numbers.

**15)** Design a circuit that adds three 10-bit unsigned binary numbers and outputs the correct result under all circumstances.

**16)** 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; SEL = "00"

RES = A + C; SEL = "01"

RES = A + B; SEL = "10"

RES = B + C; SEL = "11"

For this problem:

- Assume inputs A, B, C and the output are all 12-bit unsigned binary values
- Assume there are no issues or problems with carry out values

**17)** Repeat the previous problem but use only one RCA in your design

**18)** Design a circuit on a block diagram level with an output that represents either a mathematical operation or another input. 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.

For this problem:

- Assume inputs A, B, C, D, and the output are all 12-bit values
- Assume there are no issues or problems with carry out values

**19)** 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. The circuit also has a VALID output that indicates when the result of the addition is valid or not.

**20)** 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 complement form. For this problem, disregard any issues having to do with a carry-out.

**21)** 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.

**22)** 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. The circuit also has a VALID output that indicates when the result of the addition is valid or not.

**23)** 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.

**24)** 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.

**25)** 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 input as an unsigned binary number; the output indicates when the 4-bit input is greater than 7. 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 and odd.

**26)** 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).

**27)** 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. If all the inputs values are not even multiples of 8, then the circuit outputs all zeros.

**28)** 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 !A; if only an overflow is generated with no carry generated, the circuit outputs !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). The notion of overflow is the same as the answer being valid, meaning that if there is an overflow, the result is not value. If there is no overflow, the answer is valid.

**29)** 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 !A
- if (A + B) generates an overflow without a carry, the circuit outputs !B
- if (A + B) generates both an overflow and a carry, the circuit outputs (A – B)

**30)** Use as many 2:1 MUXes as you need to effectively create a 4:1 MUX. For this problem, consider all MUX inputs to be one-bit wide signals.

**31)** Use as many 2:1 MUXes as you need to effectively create an 8:1 MUX. For this problem, consider all MUX inputs to be one-bit wide signals.

**32)** Design a circuit that outputs the sum of A + C when a button is pressed, otherwise outputs the sum of A + B. Consider A, B, C, and the SUM to be 10-bit unsigned binary outputs. Assume the addition does not generate a carry. Consider a pressed button to output a '1'.

**33)** Design a circuit that adds two 8-bit unsigned binary values. If the addition operation generates a carry, the circuit outputs zero and turns on an LED; otherwise the circuit outputs the sum of the two values and turns the LED off. Assume the addition does not generate a carry.

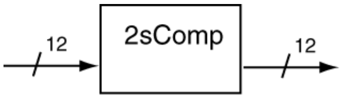
**34)** Design a circuit that performs as follows: The two-bit SEL input selects which one of four operations appears on the output as indicated to the right. The circuit also has an output Z that indicates when the output is zero. Consider the inputs and the non-Z output to be 10-bit signed binary values (RC format). Also, include a VALID output that indicates when the output is valid. Use only one RCA in your design.

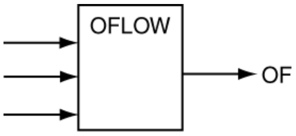
| SEL | Operation |
|---|---|
| "00" | 2A |
| "01" | 2B |
| "10" | A+B |
| "11" | -1 |

**35)** 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.

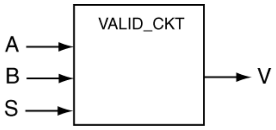


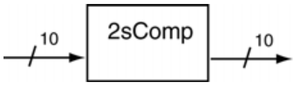


**36)** Design a circuit that inputs two 6-bit unsigned values and outputs one 6-bit unsigned value. If both inputs are even and one input is ≥ 32 while the other input is < 32, output the sum of the two inputs; otherwise output zero. Don't worry about any carry-out issues.

**37)** 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. For this problem, assume the result of the addition operations are always valid.

**38)** Design a circuit (provide a block diagram) that performs the following operations. If the BTN is asserted, the circuit outputs 2A + 2B; otherwise, the circuit outputs 2A – 2B. Assume that A & B are 10-bit signed values in RC form. This circuit has two outputs: RES, which is a 10-bit result (also in RC form) and VALID, which indicates if the 10-bit RES output is value is valid based on the math operation performed by the circuit. Feel free to use the provided 2sCOMP (does a 2's compliment) and VALID_CKT (checks for validity) boxes below (no need to define them).





**39)** 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.

**40)** 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. Use o

**41)** 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. Include a VALID output the indicates if the output value is valid.

**42)** 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. Include a VALID output that indicates if the output value is valid.

**43)** 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 foundation modules in your design.

**44)** Design the following circuit. The circuit has three 8-bit unsigned binary inputs A, B, & C. If the result of A + B generates a carry-out and the A & B are not equivalent, the circuit outputs C; otherwise the circuit outputs the sum of A + B. This circuit also has an output GT that indicates when the output is greater than $E4_{16}$. Use only standard digital modules in your design. Assume the sum will always be valid.

**45)** 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.

**46)** 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 eight. 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 odd (as in odd vs. even, not normal vs. strange).

**47)** 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.

**48)** 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.

**49)** 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. Assume the result of the addition will always be valid.

# 18 Comparators

## 18.1 Introduction

The comparator is a simple but versatile circuit. The comparator is one of our digital design foundation circuits, and is the second circuit that we design using the iterative modular design (IMD). Basic comparators are simple and instructive to design on the gate level, but their design quickly becomes complicated as we add more features.

**Main Chapter Topics**

**COMPARATORS:** This chapter introduces the comparator circuit, one of the digital design foundation circuits.

**Chapter Acquired Skills**

- Be able to describe gate-level implementations of simple comparators
- Be able to use comparators in digital design solutions

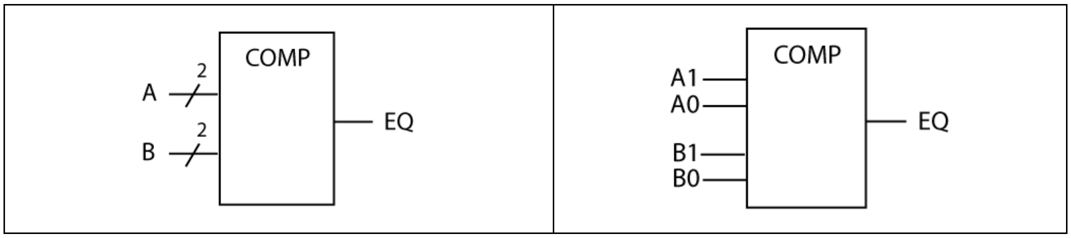
## 18.2 Comparators

The comparator is a common device in digital-land and we consider it one the digital design foundation circuits. Modeling complex comparators is relatively effortless using an HDL, which is why we only spend time discussing the design of basic comparators. The derivation of a gate-level implementation of a basic comparator provides you with some useful practice dealing with XOR-type functions and function reduction using factoring. Basic comparator design also provides us with another application of IMD.

**Example 18.1: 2-Bit Comparator**

Design a circuit that compares the values of two unsigned 2-bit binary inputs and indicates when the input values are equal.

**Solution**: We refer to this circuit as a “2-bit comparator”; we initially use the BFD approach for this design. A 2-bit comparator compares two 2-bit binary numbers; the single output indicates when the two 2-bit inputs are equivalent. Step one is drawing the BBD; Figure 18.1(a) shows the BBD using bundles notation while Figure 18.1(b) shows an equivalent version we use in our solution.



**(a)** **(b)**

**Figure 18.1: Two different forms of a 2-bit comparator.**

Next, we generate a truth table and entering the desired output values. Figure 18.2 shows that we arbitrarily list the **A** inputs as the two left-most columns in the truth table. The **A1** and **B1** inputs have a higher weighting than the **A0** and **B0** inputs[1]. Figure 18.2 shows the completed the truth table and indicates when the two inputs are equal with a '1' in the **EQ** column.

| 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 18.2: The truth table for the 2-bit comparator.**

The next step is to use the truth table to generate a set of equations representing the **EQ** output and use Boolean algebra to reduce the equation. Figure 18.3 shows this derivation, which is important to understand, as you'll occasionally need to perform similar algebraic manipulations in digital design. Note the relationship between the final equation of Figure 18.3 and the circuit implemented in Figure 18.4.

[1] The reality is that we can place the inputs in different columns the truth table; as long as you're consistent with the number values, all choices lead to the same answer.

| | |
|---|---|
| **(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 18.3: The ugly details of the final equation derivation for the 2-bit comparator.**

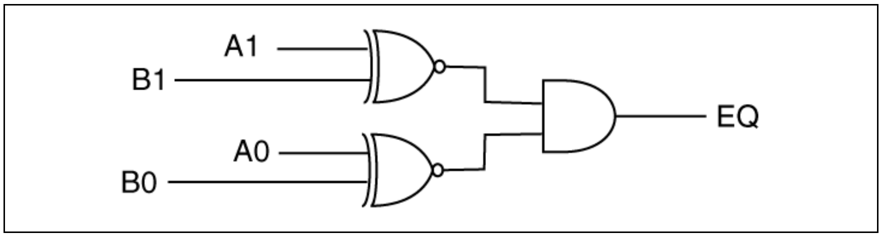


**Figure 18.4: The final circuit for the 2-bit comparator as equation (e) in this example.**

Although the circuit in Figure 18.4 seems to be nothing special, the circuit implicitly indicates a possibility to apply the IMD. 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 hardware, the AND gate is only satisfied when each of its inputs are a ‘1’. Each of the AND gate’s inputs are an output of the individual XNOR functions. Recalling that an XNOR gate is also an “equivalence gate”, so each bit position must be equivalent in order for the final number to be equivalent.

**Example 18.2: A 4-Bit Comparator**

Design a circuit that compares the values of two 4-bit inputs and indicates when the input values are equal. Show the resulting circuit diagram.

**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 variables or 256 rows ($2^8$). Would this be possible? Yes. Would anyone really do it? No. The key here is to realize that making a 4-bit comparator is a matter of adding two XNOR gates to a 2-bit comparator; this approach is a classic application of the iterative-modular design (IMD) approach; Figure 18.5 shows the final circuit diagram for a 4-bit comparator.

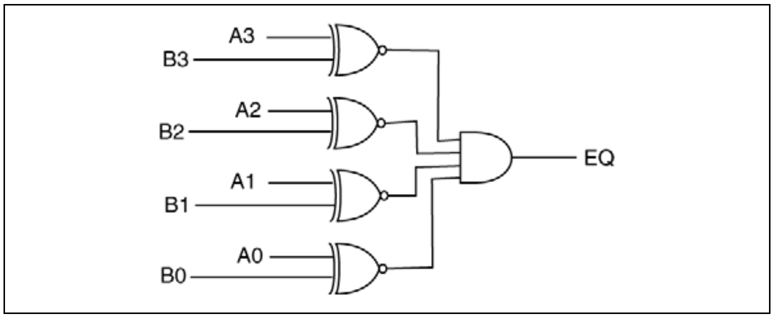


**Figure 18.5: The final circuit for a 4-bit comparator.**

In order to simplify the previous problem, we described a comparator with only one status output that indicates when the two inputs were equal. Comparators typically have other status outputs as well. Figure 18.6 shows a BBD for a generic comparator having three status outputs. In addition to **EQ**, there is now an **LT** (less than) and **GT** (greater than) output to provide more information regarding the relationship between the circuit's two data inputs. **LT** is asserted when **A**<**B**, and **GT** is asserted when **A**>**B**; this relationship is important, but arbitrary. We could generate Boolean equations for **LT** and **GT**, but doing so is not instructive, and it's straightforward to design comparators using an HDL.

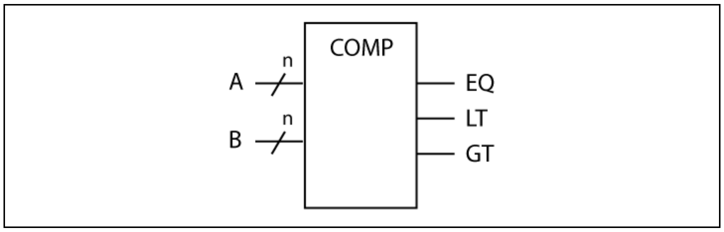
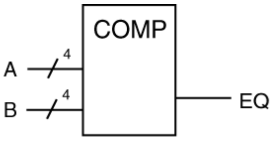
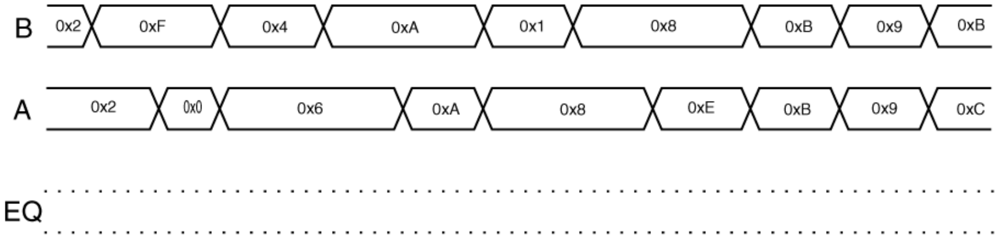


**Figure 18.6: Block diagram for a generic comparator.**

**Example 18.3: Timing Diagrams and the 4-Bit Comparator**

Use the following black box diagram to complete the accompanying timing diagram.





**Solution**: A timing diagram shows the solution to this problem without a significant amount of verbal description. Check out Figure 18.7 for all the gory details.

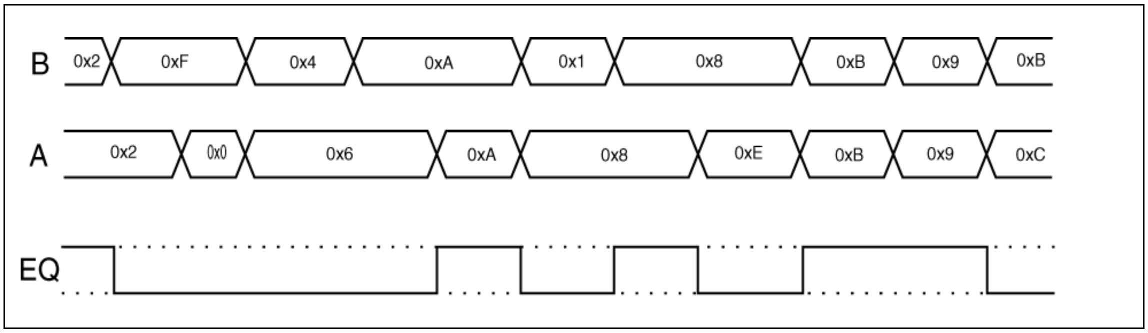


**Figure 18.7: The solution to this example.**

**Example 18.4: SM Magnitude Comparator**

Design a circuit that compares the magnitude of two 8-bit binary numbers in signed magnitude form. The circuit's one indicates when the two inputs have equivalent magnitudes. Minimize your use of hardware. Provide two levels of BBDs for your solution. Also, state what controls this circuit.

**Solution**: The first step is to draw the top-level BBD for the circuit. The circuit has two 8-bit inputs for the two binary numbers and one output. Figure 18.8 shows the final BBD.

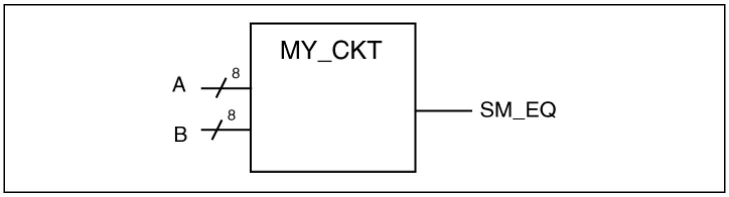


**Figure 18.8: A block box diagram that supports the description of this problem.**

The next step is to make an initial inventory of the circuit's internal modules. The circuit needs to compare two numbers, so we need to include a comparator. The circuit needs to massage the input value so that the comparator is comparing magnitudes, but we deal with that in another step.

Since binary numbers in SM form use all but the sign-bit to represent the magnitude, we only need to compare the magnitude portion of the number, which we do by feeding only the magnitude bits of the two input values to a comparator; the comparator then only needs be a 7-bit comparator. Figure 18.9 shows the final solution to this problem. The internals of this circuit is a comparator. The comparator is a device that has no control inputs, so the final circuit is not controlled.

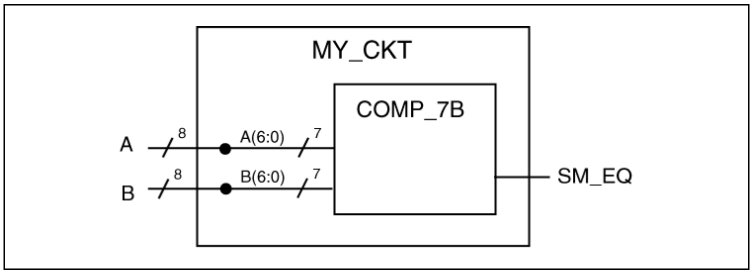


**Figure 18.9: The final solution for this example.**

Note in Figure 18.9 that we "made up" our own terminology for this problem. We put a connection dot on the bundle in an effort to indicate that we are modifying the bundle. Next, we changed the effective width of the bundle and indicated in an arbitrary, but clear manner that we are only inputting the seven lower-order bits (the magnitude bits) of the 8-bit bundle to the comparator. Whenever you do something "different", you need to document it.

**Example 18.5: 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 appears on the **GT** output and the **B** input appears on the **LT** output. Otherwise, the **B** input appears on the **GT** output and the **A** input appears on the **LT** output. Provide the top two levels of BBDs for your solution. Minimize your use of hardware in your solutions. Also, state what controls this circuit.

**Solution**:[2]. Let's start this solution with a top-level BBD; the BBD in Figure 18.10 satisfies the problem's requirements.

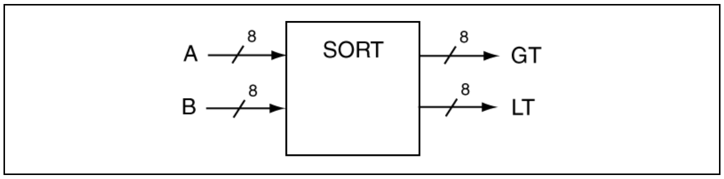


**Figure 18.10: Block diagram for Example 18.5.**

The next step is to create an inventory of the underlying modules our circuit requires. Note that this problem performs a *sort* on the input values; 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. For this problem, we use the version of a comparator that includes the **LT** and **GT** status outputs.

The second key to this problem is that there is some "selection" happening in order to "select" the inputs to feed to the correct outputs. This implies that the design requires a MUX. Since the circuit we're trying to design has two outputs, and both of the outputs need the ability to display either of the numbers, we need two 2:1 MUXes for our solution.

[2] Sorting is a common problem in computer programming, and always makes for a good hardware design problem (it's faster in hardware, anyway). If you want to excite a computer scientist, say the word "sort" to them.

The problem did not state there was external control of the sorting, such as a button, so we know the internal circuitry must provide the control inputs to the MUXes in the circuit. We are interested in 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. However, a more clever way to do this would be to use the **LT** signal on the comparator to directly control the two MUXes.

We need two MUXes, and they always need to 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 is to skip the inverter and connect the data inputs to the MUXes such that the same value from the comparator serves as a control for both MUXes. Figure 18.11 shows a diagram of the final circuit. Any problem using a MUX must also include the number associated with the MUX's data inputs.

This circuit can have different outputs, so "something" is controlling the circuits. This circuit uses internal control because the **LT** output of the comparator connects to the select inputs of the MUXes, which are the control inputs.

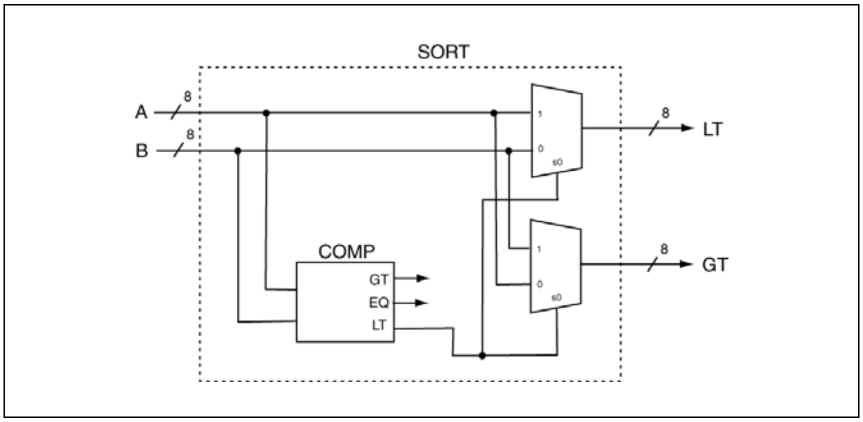


**Figure 18.11: The diagram of the final circuit.**

**Example 18.6: 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 is a '1', otherwise the circuit output is a '0'. Use only standard comparators in this design. Use any support logic you may require but minimize the amount of hardware you use in your solution. Provide two levels of BBD for your solution. Minimize your use of hardware. Also, state what is controlling this circuit.

**Solution**: The main constraint in this problem is that it requires the use of standard comparators in the solution. It's an old math thang to say, "if A = B and B = C then A = C". Your mission is then to translate that intuitiveness to digital hardware. Start with drawing a black box diagram as in Figure 18.12.

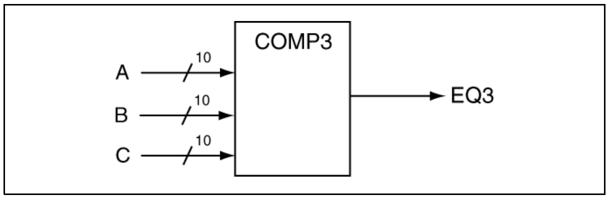


**Figure 18.12: Black box diagram for the solution.**

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, 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 18.13 shows the final block diagram for this problem. Finally, none of the circuit elements in this circuit have control inputs, thus this circuit has no control features.

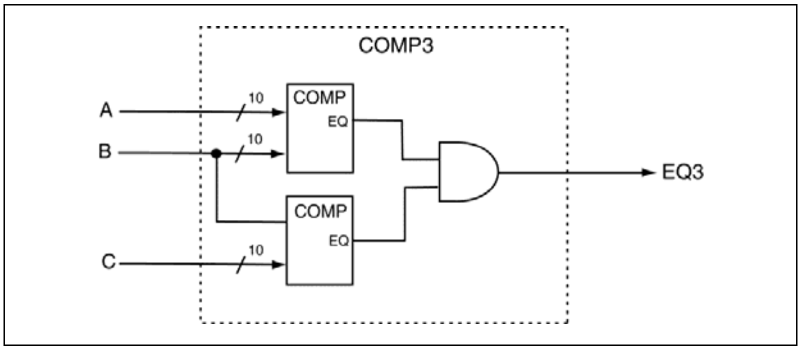


**Figure 18.13: The final circuit for this solution.**

**Example 18.7: 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 never generate a carry out. Provide two levels of BBD for your solution. Minimize your use of hardware for this design. Also, state what is controlling this circuit.

**Solution**: The first step is drawing a block diagram of the final circuit as we show in Figure 18.14.

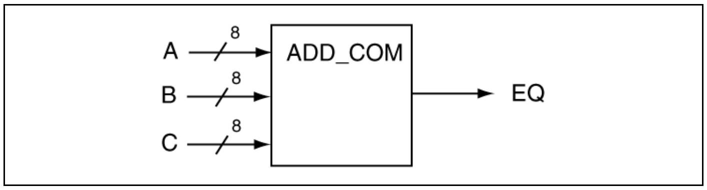


**Figure 18.14: Block diagram of the final circuit.**

The next step is making an inventory of the modules this circuit requires. From the problem statement, you can see that the final circuit to requires two RCAs in order to perform the two required addition operations (**A** + **B** & **B** + **C**). The second clue given in the problem statement is that some things need comparing. In this case, you'll need to check whether the results of the two addition operations are equivalent. For this problem, you'll need two RCAs and one comparator. The problem statement itself provides many of these clues. Figure 18.15 shows the final circuit. None of the circuit elements has control inputs, so this circuit has no control.

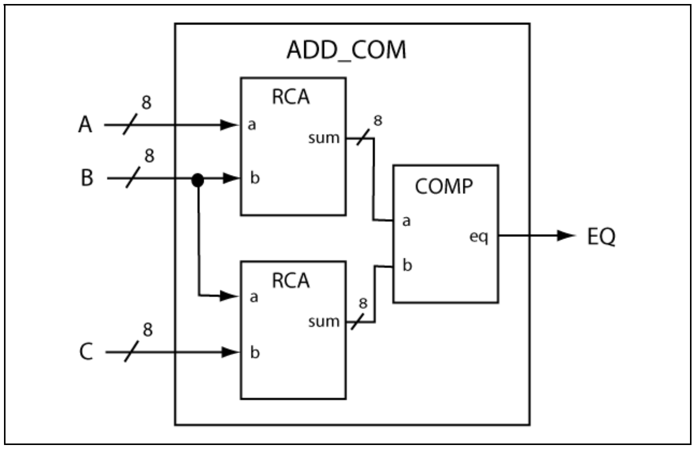
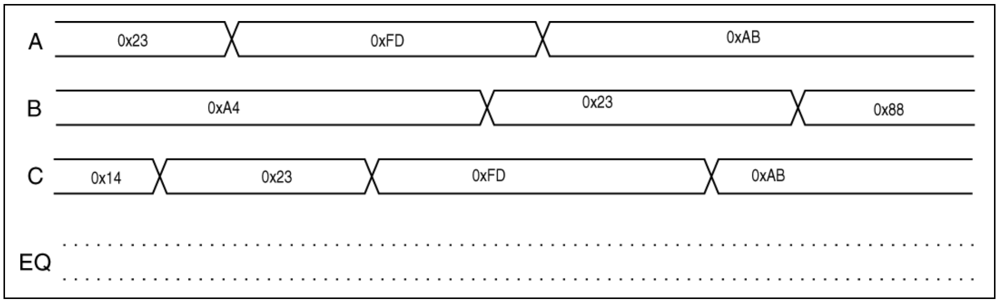


**Figure 18.15: The final circuit solution for this example.**

**Example 18.8: Arithmetic Circuit Timing Diagram**

Based on the solution to the previous example, complete the following timing diagram.



**Solution**: The problem states that the **EQ** output is a '1' whenever A = C. Figure 18.16 shows the final solution to this example. Signal **B** does not affect the answer. Also, the carry-outs from the RCAs do not affect the outputs, as they do not change the value of the RCA sum outputs.

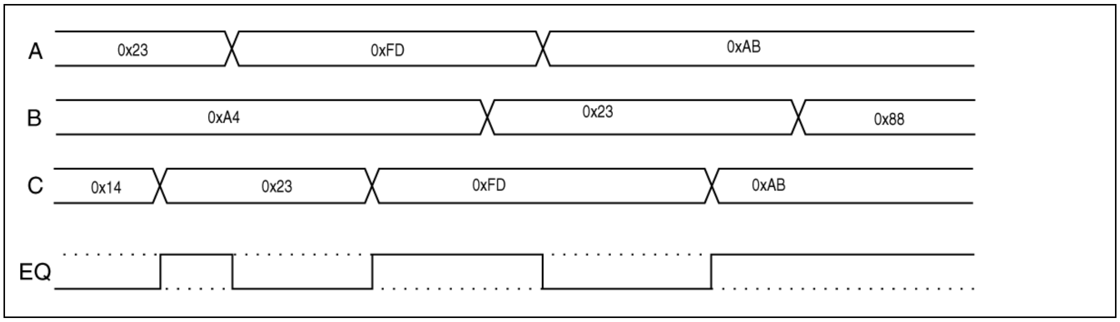


**Figure 18.16: The solution to this example.**

## 18.3 Digital Design Foundation Notation: Comparator

We consider the comparator to be one of our Digital Design Foundation circuits. The comparator is a controlled circuit. Figure 18.17 shows the appropriate digital design foundation notation for the comparator. Comparators always have two inputs, but we can choose between which comparator outputs we want to include in our design (so our comparator module has at least one, but not greater than three outputs). The **LT** output indicates when the **A** input is less than **B** (**A**<**B**), while the **GT** input indicates when **A**>**B**. The EQ output indicates that **A = B**.

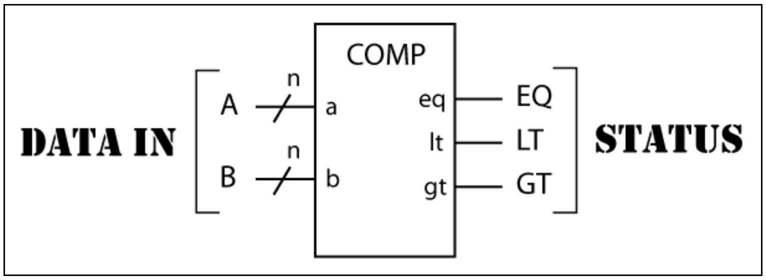


**Figure 18.17: Typical data, and status signals for a comparator.**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **A, B** | Two values to be compared; these values have equivalent data widths. |
| OUTPUT DATA | **n/a** | - |
| CONTROL | **n/a** | - |
| STATUS | **EQ, LT, GT** | Signals that indicate a relation between the two inputs A & B. EQ is asserted when A=B, LT is asserted when A<B, GT is asserted when A>B. |

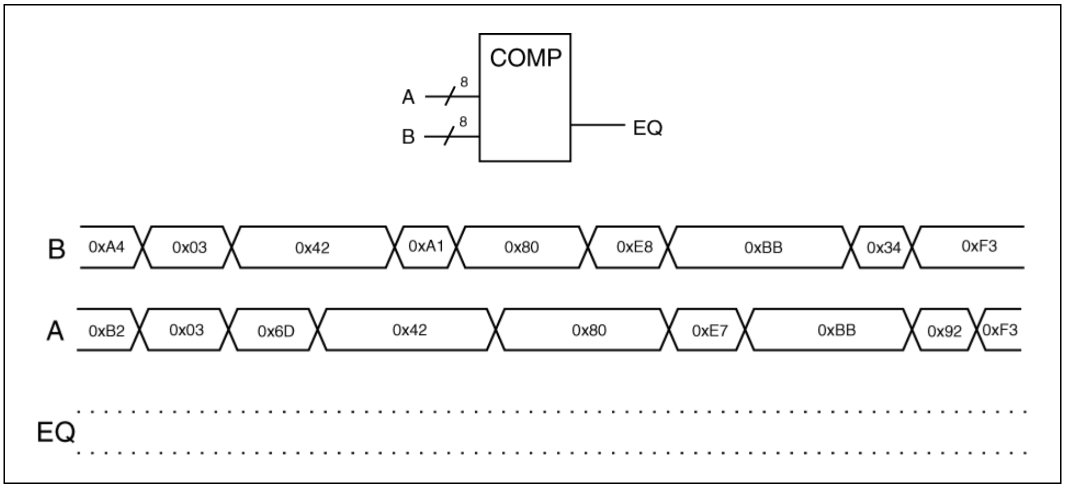
**Table 18.1: The foundation matrix for a comparator.**

## 18.4 Chapter Summary

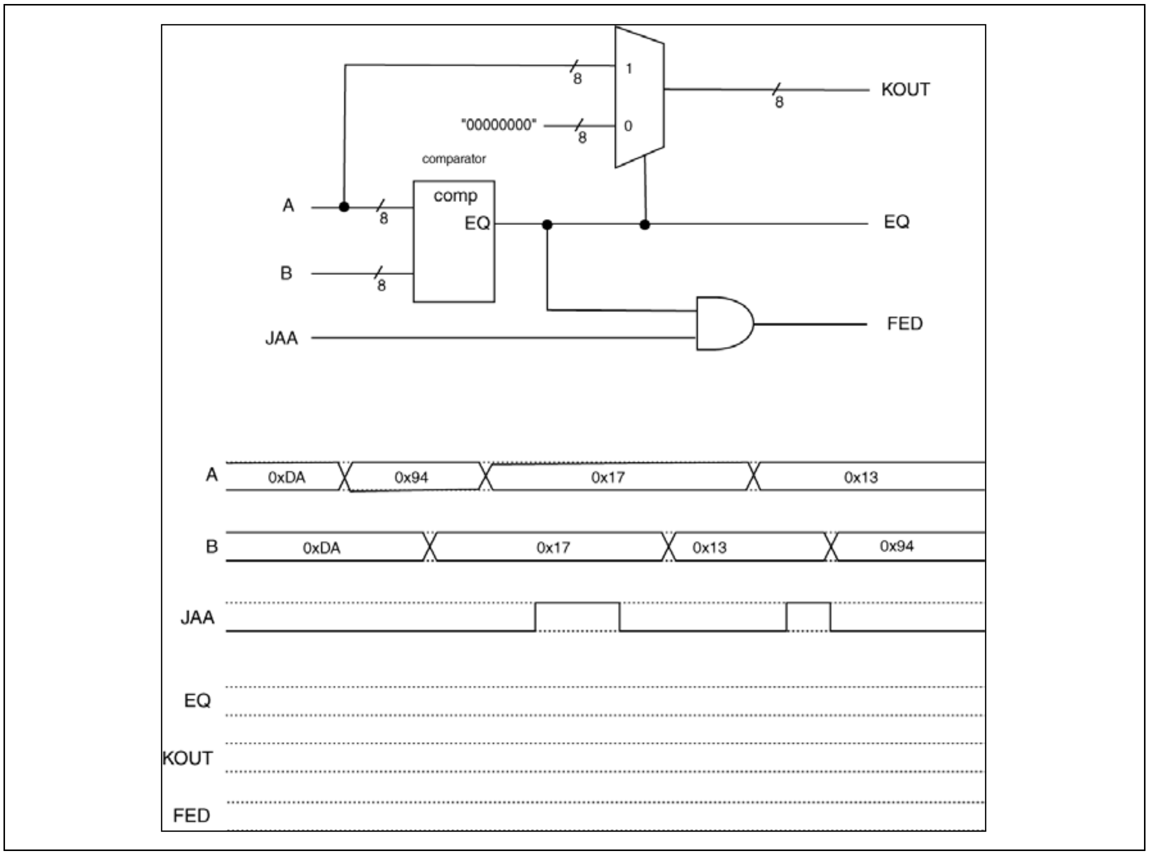
- Comparator: arithmetic circuit used to determine equality of two digital signals of equivalent data widths
- Typical comparator outputs are LT (less than), GT, (greater than), and EQ (equal), which provide information about the mathematical relations between the two inputs.
- We can design basic comparators (comparators with only EQ status outputs) using the IMD approach. When our designs require more complex comparators, we switch to modeling them with an HDL.

## 18.5 Chapter Exercises

**1)** Complete the timing diagram below considering the given schematic symbol.



**2)** Use the following circuit to complete the unlisted signals in the timing diagram. For this problem, assume there are no propagation delays.



**3)** Describe the difference, if any, in comparing RC of unsigned binary numbers.

## 18.6 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”)
- Fully describe any non-foundation modules you use in your design.

**1)** 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.

**2)** 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. For this problem, use a BFD approach, but implement whatever circuit modules you feel necessary using a decoder.

**3)** Modify a high-level BBD of a 10-bit comparator such that the outputs are A=B, A≤B, A≥B.

**4)** 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.

**5)** 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 carry-outs on the output operations.

**6)** Design a circuit that compares the sums of magnitudes of four 16-bit input values in RC format: A, B, C, & D. The circuit has two outputs: one output indicates when the four magnitudes are equivalent. The other output indicates when the sum of magnitudes of A+B equals the sum of magnitudes C+D are equivalent (when a button is pressed), or when the sum of magnitude A+C equals B+D (when button is not pressed). For this problem, assume a button press is associated with a ‘1’.

**7)** Design a circuit that performs as follows: The circuit has six 10-bit 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’.

**8)** 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.

**9)** Design a circuit that outputs one 8-bit value. If the circuit’s two 8-bit inputs, A & B are equivalent, then the sum of A & B are output; otherwise, the value of A + A is output. Use no more than one RCA in your design.

**10)** 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 and 120. Assume inputs and outputs are in RC format.

**11)** Design a circuit that compares the magnitude of two 12-bit signed binary numbers in diminished radix form. Assume the outputs of this circuit are the same as the outputs of a standard comparator.

**12)** Design a circuit that compares three 8-bit values. This circuit has one output that indicates when the three values are equal. Consider the inputs to be in unsigned binary format.

**13)** Design a circuit that acts as a special comparator to two 8-bit unsigned binary values. If a button is pressed, the circuit outputs normal standard comparator outputs. If the button is not pressed, the circuit output GT='1' when B>A and LT='1' when A>B (EQ always indicates equality)

**14)** Design an 8-bit magnitude comparator. This circuit compares the magnitude of two 8-bit signed binary values in RC format. The circuit outputs a single signal to indicate whether the two signals are equivalent or not

**15)** Design a circuit that output A if A≥B; otherwise the circuit output s zero. Consider the A&B inputs to be 10-bit unsigned binary values.

**16)** Design a circuit that does the following: if a button is pressed, the circuit output A when A>B; otherwise the circuit outputs zero. If the button is not pressed, the circuit outputs B when A<B or 0xFF otherwise. Consider A, B, and the outputs to be 8-bit unsigned binary numbers.

**17)** 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. Don't worry about overflow in this design.

**18)** 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.

**19)** 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. Assume the values are all unsigned binary; don't worry about carry-out issues for this problem.

**20)** Design a circuit that outputs one 8-bit value. If the sum of the circuit's two 8-bit inputs, A & B, generates a carry out and the two inputs 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. Don't worry about carry-out issues of 2A & 2B.

**21)** 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 $120_{10}$.

**22)** 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.

**23)** 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. Make this problem work for all possible values.

**24)** 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.

**25)** Design the following circuit. The circuit has three 8-bit unsigned binary inputs A, B, & C. If the result of A + B generates a carry-out and the A & B are not equivalent, the circuit outputs C; otherwise the circuit generously outputs the sum of A + B. This circuit also has an output GT that indicates when the output is greater than $E4_{16}$.

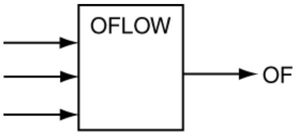
**26)** Design a circuit that does the following. If the circuit's two 2-bit values (A & B) are not equivalent, then the 8-bit value AA will show up on the circuit's output; otherwise, the 8-bit value BB will show up on the output. Consider AA and BB to be inputs to the circuit.

**27)** 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.

**28)** 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'.

**29)** Design a circuit that performs as follows: The circuit has three 4-bit unsigned binary inputs (A,B,C). If the value of A + 2 (addition) equals B and the value of A + 3 (addition) equals C, and neither A+2 or A+3 is greater than 15, the circuit outputs B; otherwise, the circuit outputs C.

**30)** 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 !A; if only an overflow is generated with no carry generated, the circuit outputs !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.



**31)** 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 $120_{10}$.

**32)** 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.

# 19 Parity Generators and Checkers

## 19.1 Introduction

The parity generator and parity checker are two common digital circuits. We consider the parity generator a digital design foundation module, and is another circuit that we design using IMD. Basic parity generators are simple and instructive to design on the gate level and are yet another circuit known for having XOR functions in their design.

**Main Chapter Topics**

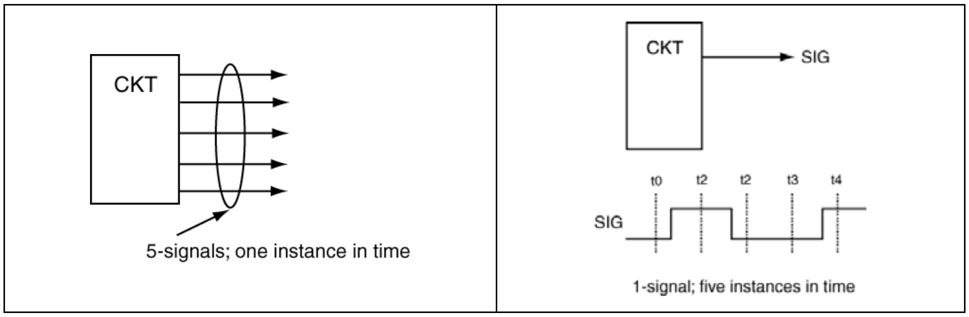
**PARITY GENERATORS:** This chapter introduces the notion of parity and the design of parity generators and parity checking circuits.

**Chapter Acquired Skills**

- Be able to describe the concept of parity
- Be able to describe the most common use of parity in a real-world applications
- Be able to describe a parity generator at the gate level
- Be able to use parity generators and parity checkers in digital circuits

## 19.2 Parity Generators and Parity Checkers

Parity generators and parity checkers are two standard digital circuits that we often use in digital communications. We can apply the concept of parity to a set of bits, which can either exist at one moment in time in a parallel configuration or the bits can exist over several set times in a serial configuration. Figure 19.1(a) and Figure 19.1(b) show an example of both parallel and serial configurations, respectively. In Figure 19.1 (a), the values of the bits in question exist at one instance in time. Figure 19.1(b) shows that we can also apply parity to a single signal over a given time span. The parity concept applies to the set of bits in that are the values of the SIG signal at five different instances in time.


CKT
5-signals; one instance in time
CKT
SIG
t0
t2
t2
t3
t4
SIG
1-signal; five instances in time

(a) (b)

**Figure 19.1: An example of parallel signals and serial signals.**

Once we assemble 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'; modulo-2 addition has no concept of a carry, so we discard any carry resulting from the addition. Thus to perform a modulo-2 addition on a set of bits, you add all the set bits and your result is either '0' or '1'. If the sum of the set of bits is '0', the result of the addition is even (even parity). If the sum of bits is not even, then the sum must be odd (odd parity). The XOR gate inherently performs modulo-2 addition on its two inputs[1]. The concept of odd and even parity has nothing to do with odd and even numbers.

Parity is particularly useful in digital communications; Figure 19.2 shows a simple example of a communication system that uses parity. This example shows four bits that require transferring in parallel across some of medium. The medium is immaterial; the important things is the four data bits that need transferring: three data bits and a parity bit. The Data Generator box generates the data that requires transferring.

The Parity Generator box is a circuit that imposes either an odd or an even parity to the three data lines. The system then includes the parity bit with the data bits that the circuit sends in the communication channel. The Parity Generator 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 (once they are received), the parity needs to be the same as it was prior to transferring the bits. If the bits were sent with even parity and arrive with odd parity (or vice versa), an error occurred during transmission. If the bits were originally sent with odd parity and arrive with odd parity, there is a good chance that there was not an error during transmission[2].

The circuit in Figure 19.2 provides *1-bit error detection* for the data bits sent across the mediums. The circuitry on the receiving end expects either odd or even parity (as designed into the circuitry); if the parity of a received message is different from the parity of the sent message, the circuit indicates an error on the PR output of the Parity Checker[3].

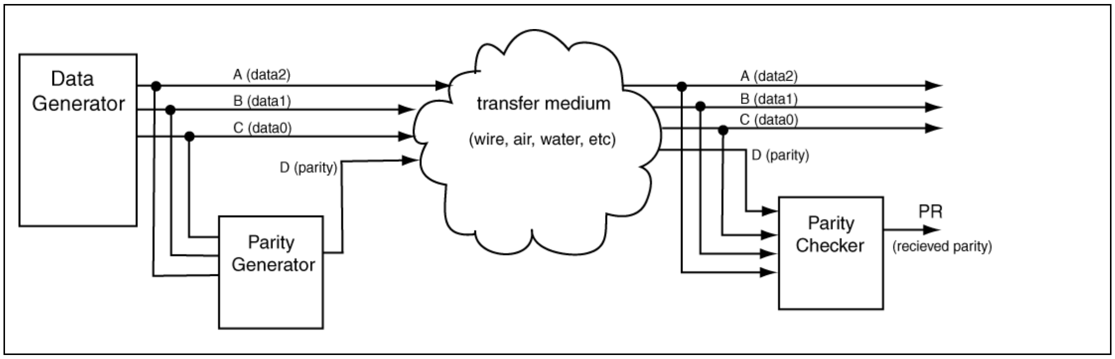


**Figure 19.2: An example of parity generation and checking.**

The circuitry for parity generators and parity checkers is straightforward. The approach we take is to design a small circuit using BFD, then we can use IMD to create a larger circuit.

**Example 19.1: 3-Bit Even Parity Generator**

Use BFD to design a circuit that generates a parity bit that indicates when three bits are even parity. Consider the data bits to be **A**, **B**, & **C**. Consider the parity bit to be **D**.

[1] We consider the condition when zero or no bits that are '1' to be even (even parity).

[2] 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 than the probability that one bit was in error, which is why parity is an effective error detecting measure.

[3] 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.

**Solution**: We need to design a Parity Generator such that the circuit generates even parity based on the data bits **A**, **B**, and **C**. We thus need to assign **D** to ensure that the set of bits **A**, **B**, **C**, and **D** have even parity. Figure 19.3 shows the BBD for our circuit.

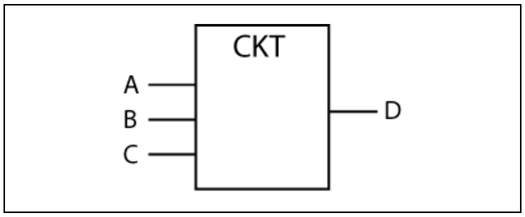


**Figure 19.3: The truth table for a 3-bit even parity generator.**

We start with a truth table 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 is 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', the parity of bits **A**, **B**, **C**, and **D** is even.

The truth table in Figure 19.4 shows the parity concept in tabular form; we assigned the **D** column to ensure that the four bits in each row have even parity. Table 19.1 shows that we can factor the equation generated from Figure 19.4 to simplify the equations, which allows us to extract XOR equations. Figure 19.5 shows the circuit associated with the final equation of Table 19.1.

| 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 |

**Figure 19.4: The truth table for a 3-bit even parity generator.**

| | |
|---|---|
| **(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 19.1: Derivation of the even parity generating circuit.**

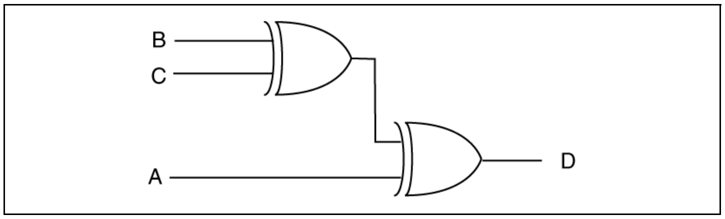


**Figure 19.5: The final circuit for a 3-bit even parity generator.**

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 we can easily do using a truth table. Figure 19.6 shows the required 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 sent with even parity, the arrival of bits having an odd parity indicates that an error occurred in transmission. If you were to grind out the equations for this truth table, Figure 19.7(a) lists the final equation, while Figure 19.6(b) shows the resulting circuit.

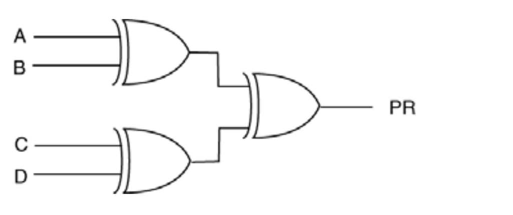
As a final note in this saga of parity generation and checking, you should notice a similarity between the final equation of Table 19.1(d) and equations in Figure 19.7(a). The only difference between a 3-bit even parity generator and a 4-bit even parity generator is the addition of one more XOR term. This similarity allows you to apply IMD to create parity generators and checkers for any number of bits.

| 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 |

**Figure 19.6: The truth table for the 4-bit even parity checker.**

$$PR = (A \oplus B) \oplus (C \oplus D)$$

$$PR = A \oplus B \oplus C \oplus D$$



**(a)** **(b)**

**Figure 19.7: The equations (a) and circuit (b) for the 4-bit even parity generator.**

**Example 19.2: 4-Bit Even Parity Generator**

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 view the parity bit that this circuit generates. One way to view the parity bit is that it indicates with a '1' when the four input bits are odd parity. Another way to view the parity bit is that we assign the parity bit such that the five bits (the four input bits and the parity bit) always exhibit even parity.

Figure 19.8 shows the top-level BBD for this problem. There are four input bits; the output labeled "PR" is the parity bit.

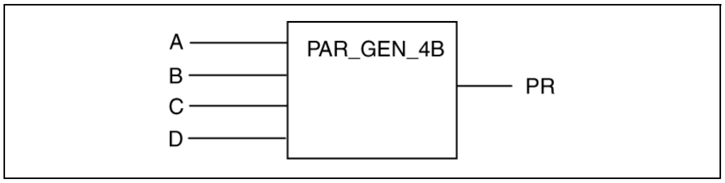
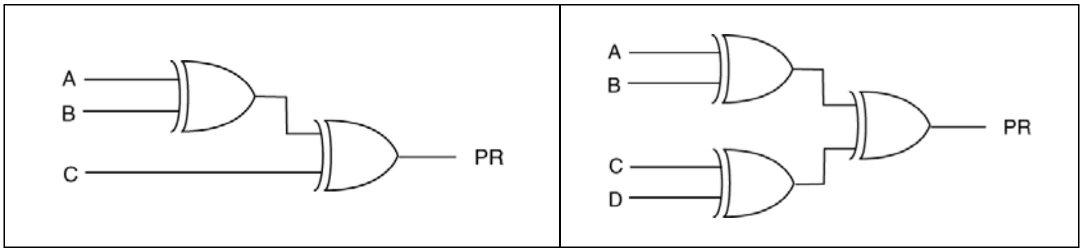


**Figure 19.8: The block diagram for this example.**

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 19.9**(**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 we show in Figure 19.9(b). For this problem, the communications channel would now be sending five signals: A, B, C, D, & PR. The receiving end of the channel examines these five signals in order to verify that the received signal exhibits even parity.

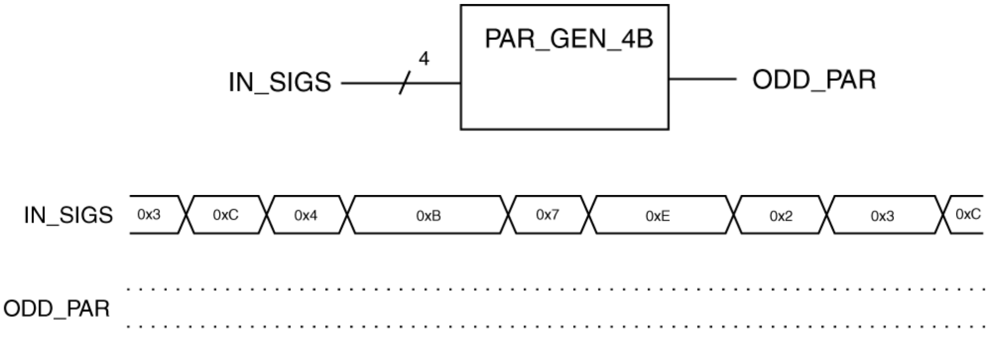


**(a)** **(b)**

**Figure 19.9: The circuit solution for a 3-bit even parity generator (a) and the solution to this example for a 4-bit even parity generator**.

### Example 19.3: Timing Diagrams and an 4-Bit Odd Parity Generator

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 19.10 shows the final timing diagram.

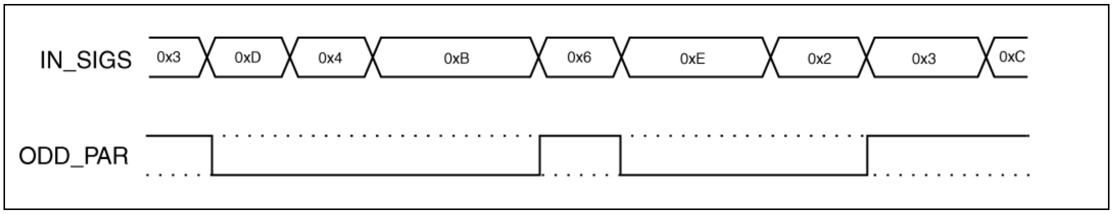


**Figure 19.10: The solution to this example.**

### Example 19.4: Parity Design Problem #1

Design a circuit that inputs one 8-bit value. If the input value is even and has odd parity, the circuit outputs the input value; otherwise, the circuit outputs the 2’s complement of the input value. Consider the input value to be in RC format. Feel free to use BBDs for the 2’s complement and parity circuit without providing descriptions of the underlying implementations, but be sure to label them accordingly. Minimize your use of hardware. Provide the top-two levels of BBD for your solution. State what controls the circuit.

**Solution**: We start this problem by creating a BBD for the solution based on the problem description. Figure 19.11 shows the top-level BBD for this problem. The fun stuff is on the inside of the box.

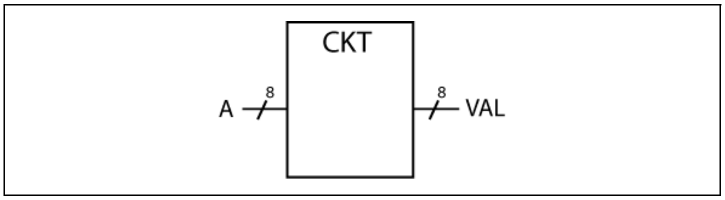


**Figure 19.11: The block diagram for this example.**

The next step is to make an inventory of the modules we need to solve this problem. Note that the circuit outputs one of two values; this means the solution includes a MUX. The circuit chooses between either the **A** input or the 2's complement of the **A** inputs, which means the circuit requires a module to perform a 2's complement. The control input to the MUX is a combination of the LSB and an indicator of the parity of the **A** input. We need an odd parity checker circuit to indicate when the **A** input is has odd parity. We also need to examine the LSB of the **A** input to determine if the value is even. Because the **A** input is even when the LSB of A is '0', we invert that value and then AND it with the output of the parity checker to form the control input to the MUX. Figure 19.12 shows the final circuit diagram for our solution.

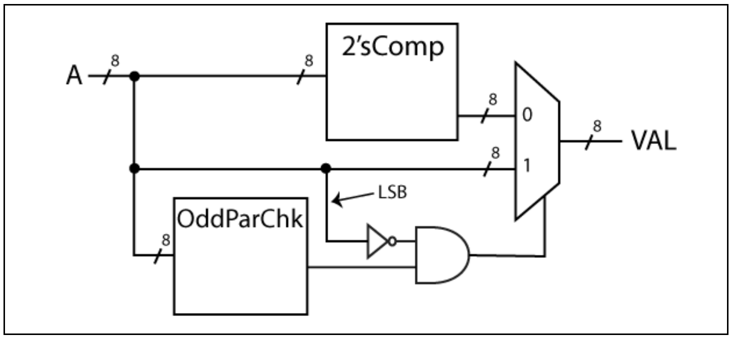


**Figure 19.12: The final lower-level BBD for our solution.**

**Example 19.5: Parity Design Problem #2**

Design a circuit that inputs two 8-bit values (**A** & **B**), both in RC format. If the parity of the two values is different, the circuit outputs **A + B**; otherwise the circuit outputs **A –B**. The result output is an 8-bit value in RC format. The circuit also contains a **VALID** output that indicates when the result of the given operation is valid. For this design, you can use "2's comp" and "valid" modules without providing descriptions of the underlying implementations, but be sure to label them accordingly. Minimize your use of hardware. Provide the top-two levels of BBD for your solution. State what controls the circuit.

**Solution**: We start this problem by creating a BBD for the solution based on the problem description. Figure 19.13 shows the top-level BBD for this problem. The fun stuff is on the inside of the box.

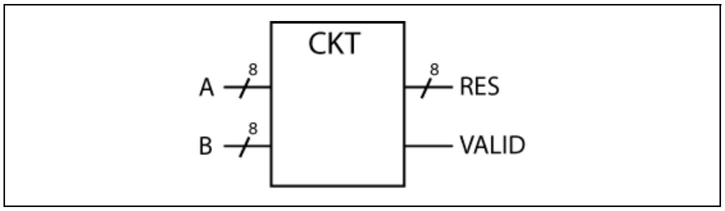


**Figure 19.13: The block diagram for this example.**

The next step is to make an inventory of the modules the final circuit requires in order to solve the given problem. This circuit has a relatively large number of modules; we list them below.

- The circuit indicates a sum or difference of the inputs, so we need an RCA.
- The circuit performs a subtraction, so we need a 2's complement module.
- The circuit chooses between adding or subtracting the B input, so we need at least one MUX.
- The circuit needs to determine the parity of both inputs, so we need two even parity checker modules.
- The circuit adds or subtracts based on the parity of the two inputs, which results in some extra logic. In this case, we need an XOR gate.
- The final circuit output is either the result of the mathematical operation or zero, based on the validity of the chosen operation. This means we need a second MUX, which we control with the output of the validity circuit.

Figure 19.14 shows the final lower-level BBD for this problem. We include annotations with the BBD to indicate information regarding the inputs to the "valid" module. If we omit this note, the circuit would not be complete.

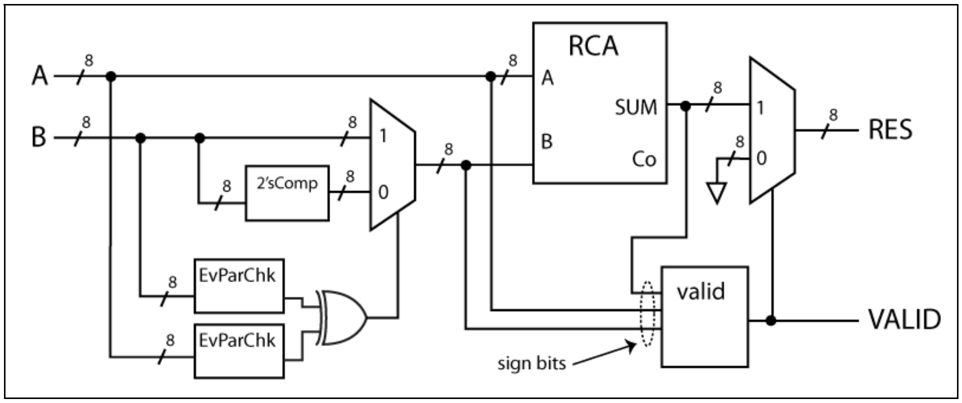


**Figure 19.14: The final lower-level BBD for our solution.**

## 19.3 Extra Parity Details

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 even parity of the signals into odd 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' makes the odd parity of the signals into even parity.

## 19.4 Digital Design Foundation Notation: Parity Generator

We consider the parity generator to be a Digital Design Foundation circuits. The parity generator is a controlled circuit. Figure 19.15 shows the appropriate digital design foundation notation for the parity generator. We only list the data input as a bundle, which implies the parallel version of data rather than serial data. The single status output is the PAR signal, which indicates the parity of the input data. When you use this diagram in your design, the status signal's name should also indicate either odd or even parity.

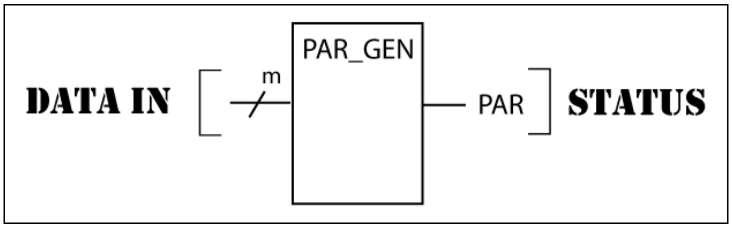


**Figure 19.15: Typical data, control and status signals parity generator.**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **DATA** | The value that the device generates a parity bit for the given "m" input bits. |
| OUTPUT DATA | **n/a** | - |
| CONTROL | **n/a** | - |
| STATUS | **PAR** | The bit that creates the appropriate parity for the DATA & PAR aggregate value. |

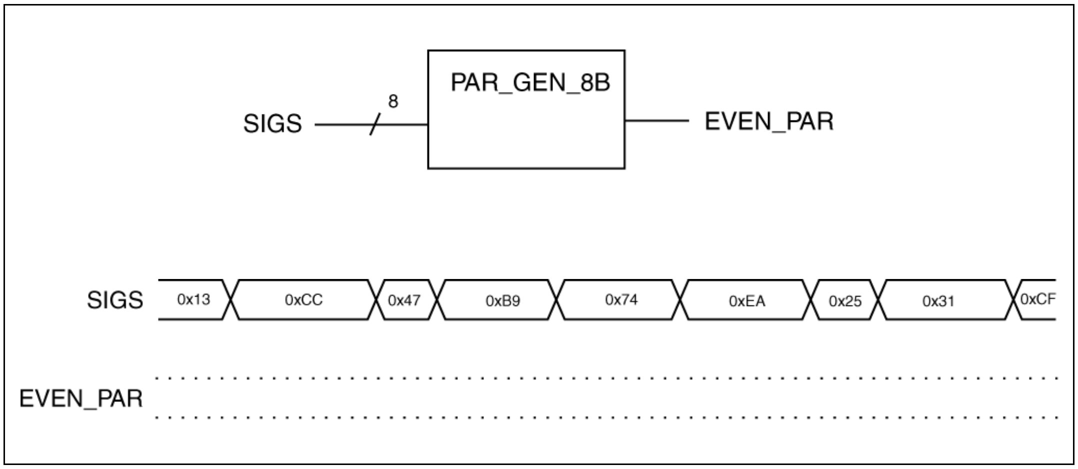
**Table 19.2: The foundation matrix for a parity generator.**

## 19.5 Chapter Summary

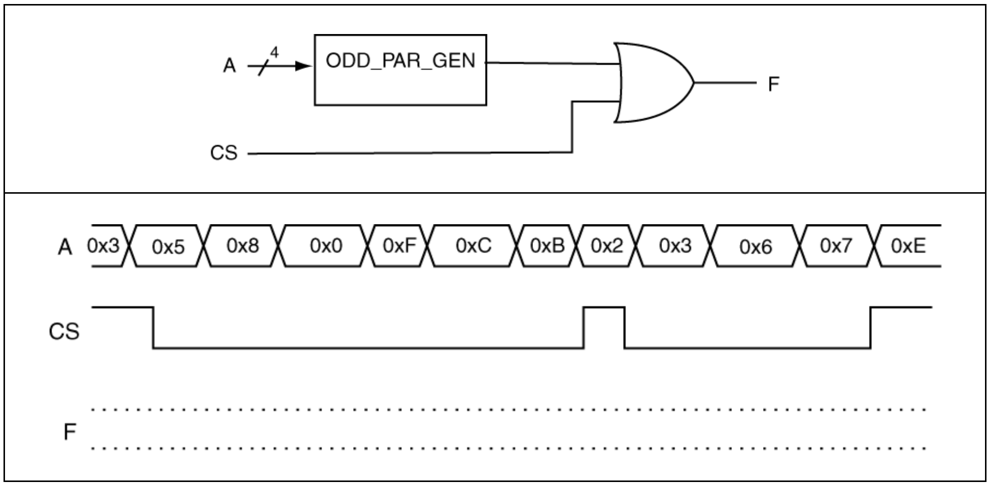
- The notion of parity describes a characteristic of a set of signal or a sequence of signals. Parity is defined as the modulo-2 addition of the ‘1’ bits of the signals in question. Parity can be either even or odd. Parity generators are used to generate a parity bit that ensures a group of signals exhibit even parity or odd parity.
- Parity checkers are essentially the same circuit as parity generators: we implement both circuits on the gate-level using exclusive-OR type gates.
- Odd and even parity has no relation to the numerical attributes of “odd” and “even”.

## 19.6 Chapter Exercises

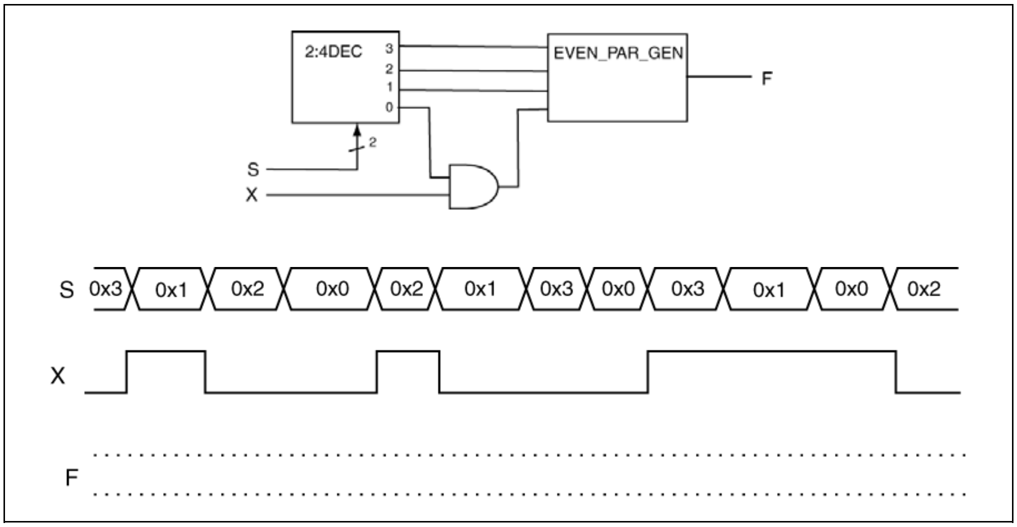
**1)** Complete the timing diagram below considering the given schematic symbol. Consider the circuit to generate even parity for the eight input bits.



**2)** Use the following circuit to complete the listed timing diagram.



**3)** Use the following circuit to complete the listed timing diagram



**4)** Can a given unsigned binary number be even but have odd parity? Briefly explain.

**5)** Does the notion of parity apply equally to unsigned and signed binary numbers? Briefly explain.

**6)** Consider the case where two n-bit unsigned binary numbers are added together using a Ripple Carry Adder (RCA). If the two numbers being added together both have odd parity, will the result necessarily have even parity? For this question, consider the addition will never generate a carry-out. Explain fully but briefly.

## 19.7 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses (“no control”, “internal control”, and/or “external control”).

**1)** Design a 3-bit odd parity generator using BFD. Specifically, this circuit uses the single-bit output to make the combination of the input plus the output even parity. Assume the 3-bit inputs are in a parallel configuration. Use Boolean algebra to reduce the resulting equations.

**2)** Using the design from the previous problem, design an 8-bit even parity generator using IMD. Assume the 8-bit input is in a parallel configuration.

**3)** Design a 3-bit odd parity generator using BFD. Specifically, this circuit uses the single-bit output to make the combination of the input plus output odd parity. Assume the 3-bit inputs are in a parallel configuration. Use Boolean algebra to reduce the resulting equations.

**4)** 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.

**5)** Design a 4-bit even parity checker. This circuit indicates when the parallel 4-bit input is even parity. Use Boolean algebra to reduce the resulting equations.

**6)** Using the design from the previous problem, design an 8-bit even parity checker using IMD. Assume the 8-bit input is in a parallel configuration.

**7)** Design a circuit that has one 8-bit input. If a button is pressed, the circuit’s single output makes the 8-bit input value plus the output even parity; otherwise, the circuit’s single output makes the 8-bit input odd parity. You can use BBDs for the parity generators.

**8)** Design a circuit that has one 12-bit input A in RC format. If a button is pressed, the circuit outputs –A. If the button is not pressed, the circuit outputs !A if the A input has odd parity or A if the input has even parity. Assume the 12-bit output is also in RC format. You can use BBDs for the parity generators.

**9)** Design a circuit that has two 10-bit binary inputs and one 10-bit output, all in RC format. If A-B is both valid and has even parity, the circuit outputs the result of A-B. If the result is odd parity and valid, the circuit outputs A. If the result is not valid, the circuit outputs B if the button is pressed or zero if the button is not pressed. You can use BBDs for the parity generators.

**10)** Design a circuit that has two 8-bit signed binary inputs (RC format) A & B. If the two inputs have the same parity, the circuit outputs A+B if a button is pressed or A-B if the button is not pressed. If the two inputs are of different parity, the circuit outs A if the button is pressed or B if the button is not pressed. Use no more than one RCA in your design. The circuit also has a VALID output to indicate if the result of the mathematical operations is valid. Assume the output is also an 8-bit value in RC format. You can use BBDs for the parity generators.

**11)** Design a circuit that inputs two 10-bit values, A & B, in RC format. If both values are positive and both values exhibit odd parity, the circuit output A-B. If both values are of different sign and both values exhibit even parity, the circuit outputs A+B. If the above two conditions are not met, the circuit outputs zero. Assume the 10-bit output is in RC format and is always valid. You can use BBDs for the parity generators.

**12)** Design a circuit that has one 17-bit input A in RC format. When the input is odd parity but not all 1's, the circuit outputs A. If the circuit is even parity and not all 0's, the circuit outputs –A. If neither of the above two conditions are met, the circuit output !A. Assume the 17-bit output is also in RC format. You can use BBDs for the parity generators.

**13)** Design a circuit that has one 16-bit input in RC format. If the bottom byte if the input is evenly divisible by 8 and has odd parity, one of the circuit's 8-bit outputs has the lower 8-bits of the input; otherwise it outputs zero. If the upper byte of the input positive and evenly divisible by 4, the circuit's other 8-bit output has the 8-bits of the inputs, otherwise it show zero. You can use BBDs for the parity generators.

# 20 Introduction to Sequential Circuits

## 20.1 Introduction

There are two major classes of digital circuits; we've dealt with only one type of circuit: combinatorial, or combinational circuits. The other major type of digital circuits is sequential[1] circuits; most digital circuits use a combination of both circuit types. This chapter introduces the basic concepts behind sequential circuits.

**Main Chapter Topics**

**SEQUENTIAL CIRCUITS**: There are two types of digital circuits: combinatorial and sequential. Previous chapters dealt only with combinatorial circuits. We generally characterize sequential circuits as having the ability to store information. This chapter describes basic latches, which are of basic digital storage elements.

**STATE REPRESENTATIONS**: We characterize circuits with memory by the "state" of the circuit. We define the state of the circuit by the values stored in the circuit's storage elements.

**SEQUENTIAL CIRCUIT REPRESENTATION**: We can represent basic digital storage elements in various ways. This chapter outlines the analysis and representations methods of basic sequential circuits.

**Chapter Acquired Skills**

- Be able to describe the main qualities of combinatorial and sequential circuits.
- Be able to describe the concept of "state" in the context of digital circuits
- Be able to describe what generates memory in a digital circuit
- Be able to describe the basic operation of NOR & NAND latches
- Be able to use state diagrams to discern the operation of NOR & NAND latches
- Be able to use PS/NS tables to describe the operation of NOR & NAND latches

## 20.2 Sequential vs. Combinatorial Circuit

Let's look back at one of the first figures we used in this text. We claimed that Figure 20.1 provided a high-level model of all the circuits that we would use in digital design. Up until now, this definition has been 100% correct: the outputs of all the circuits were direct functions of the circuit inputs. A more accurate description of the circuits we've worked with up until now is that a change in the circuit's input always causes the same reaction (change or no change) in the circuit's output.

[1] The term sequential used in the context of digital circuit should not be confused with sequential statements in the VHDL language: they are completely different concepts.

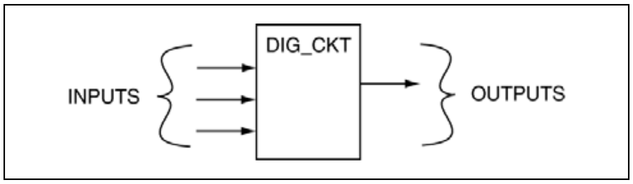


**Figure 20.1: Digital Design in a nutshell**

The input/output relationship in a sequential circuit is different from that of combinatorial circuits. The name sequential hints to the major attribute of a sequential circuit in terms of the input/output relationship: the ***sequence*** of inputs determines the outputs of a sequential circuit. Relative to a combinatorial circuit, this definition means that at one point in time, an input change may cause a certain change in circuit outputs, but at another point in time, the same input change may cause a different change in the circuit outputs. Thus, the output of a sequential circuit is based on the history of inputs and not the inputs themselves. This description implies that sequential circuits have an attribute responsible for its output behavior; we refer to this attribute as *memory*.

If you only remember one thing from digital design, the difference between combinatorial and sequential circuits should be that thing[2]. Table 20.1 shows the true differences between sequential and combinatorial circuits.

| **Sequential Circuits** | **Combinatorial Circuit** |
|---|---|
| Definition: The circuit's outputs are a function of the sequence of the circuit's inputs | Definition: The circuit outputs are a function of the circuit inputs |
| Characteristics: The circuit has at least one single-bit memory element | Characteristics: The circuit does not have memory |

**Table 20.1: The main attributes of sequential and combinatorial circuits.**

In digital design, the notion of "state" has a specific definition. We haven't needed to use the term "state" yet because our circuits up to this point did not have a "state" (they were all combinatorial). We can now refer to the *state* of a sequential circuit. We define the "state" of a digital circuit as the value(s) that the circuit is currently storing in its memory element(s). The notion of state is important because it's virtually impossible to describe sequential circuits without mentioning the state of the circuit.

## 20.3 Sequential Circuits: Low-Level Basics

Let's start examining sequential circuits by analyzing the seemingly simple circuit in Figure 20.2. While this circuit does not appear different from other circuits we've been dealing with, the circuit contains one distinct difference: there is a connection from the circuit's output to the circuit's input, which we refer to as *feedback*. In other words, the **Q** output "feeds back" and becomes a circuit input. This feedback is what ultimately gives circuits the ability to store data.

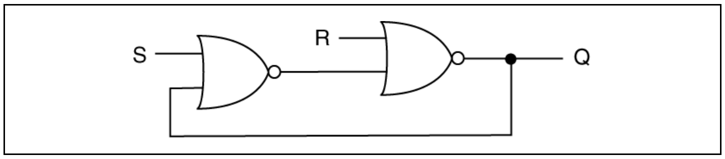


**Figure 20.2: A seemingly simple circuit.**

[2] This is a common interview question and one that is easily asked by a Human Resource person (or someone else who knows nothing about technology or probably anything else for that matter) conducting the interview.

Analyzing the circuit in Figure 20.2 will show it has some interesting properties. In order to analyze this circuit, we'll consider the circuit elements as non-ideal devices. In order to simplify the analysis, let's combine the delays associated with the two NOR gates into one delay; we model the propagation delays with the box labeled **td** in Figure 20.3. The circuit model of Figure 20.3 provides a time delay between two of the circuit's signals, which we arbitrarily refer to as: **Q**$^+$ and **Q**. Even though **Q**$^+$ and **Q** are essentially the same signal, modeling them as different signals simplifies the analysis.

The signal names of **Q**$^+$ and **Q** are special signal names with special symbology. The output that interests us is **Q**, as it's the true output of the circuit. In order to analyze this circuit, we need to consider the values of **Q** as well as the **S** and **R** inputs. However, since we also need to consider how the value of **Q** changes, we need some way to represent the new value of **Q**; we use the "**Q**$^+$" symbology to represent the new value of **Q**, after the **S**, **R**, and **Q** outputs act to alter the circuit output.

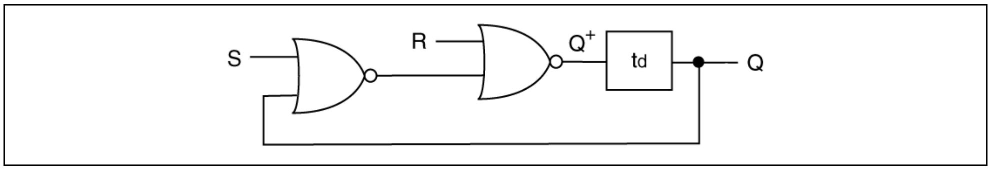


**Figure 20.3: The seemingly simple circuit modeled in such a way as to facilitate analysis.**

Figure 20.3 shows a circuit that has three inputs, **Q**, **S**, and **R**, and one output, **Q**$^+$. Since there are only three inputs, we can analyze this circuit using a truth table. Figure 20.4 shows the empty truth table we use in this analysis. To fill in the **Q**$^+$ column of this truth table, we treat **Q**, **S**, and **R** as independent variables and use them to generate the final value of **Q**$^+$. We'll do a couple of rows and you can do the others at your leisure, if you have any· Figure 20.4(b) shows the completed truth table for the circuit of Figure 20.3. Here are details in analyzing three rows of the table:

- Truth Table Row #0: Since both **Q** and **S** are '0', the output of the first NOR gate is '1'. Anytime there is a '1' input to a NOR gate, the output is '0'. Therefore, don't need to consider the value of the **R** input; the output for this row: **Q**$^+$ is '0'.
- Truth Table Row #2: Since the **S** input is a '1', the output of the first NOR gate must be a '0' and we don't need to consider the **Q** input value. The **R** input is also has a '0' value; since the second NOR gate's inputs are both '0', the **Q**$^+$ output is therefore a '1'.
- Truth Table Row #6: Since one of the inputs to the first NOR gate is a '1', the output of the first NOR gate must be '0'. The **R** input value is also '0', which causes the output of the second NOR gate a '1'. For this row, the **Q**$^+$ output value is a '1'.

| Q | S | R | $Q^+$ |
|---|---|---|---|
| 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)**

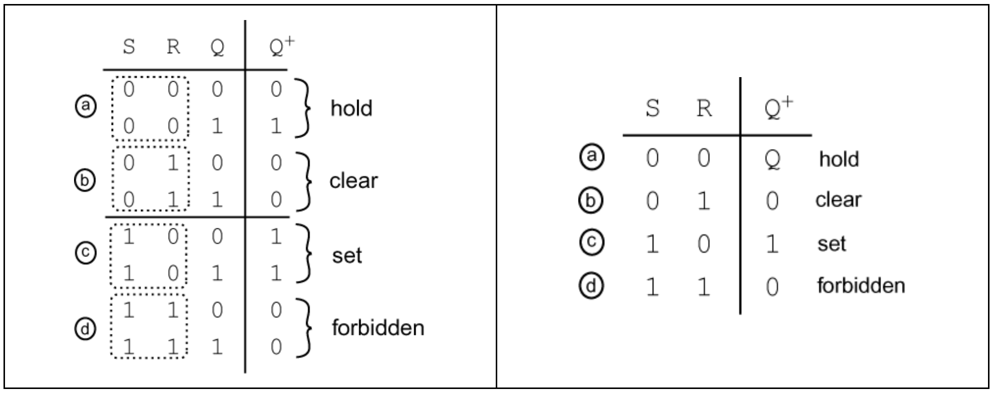
| Q | S | R | Q+ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**(b)**

**Figure 20.4: The empty (a) and completed truth table (b) for the seemingly simple circuit.**

Figure 20.5(a) shows a truth table of Figure 20.4(b) after we translate the truth table into a more usable form. In this new truth table, we re-arrange the independent variables in order to clearly show the relationship between the **Q** and **Q**$^+$. The values of **Q** and **Q**$^+$ in Figure 20.5 represent the output of the circuit but at different times. More specifically, the value of **Q** represents the current state (or output) of the circuit while the value of **Q**$^+$ represents the state of the circuit after a time delay. In sequential circuit terms, the value of **Q** represents the current or *present state* of the circuit while the value of **Q**$^+$ represents the new state of the circuit after a time delay, or the *next state*. The *state changes* in the circuit define the circuit.

The following verbage describes the rows of the truth table in Figure 20.5(a). Comparing and contrasting this analysis to the timing diagram in Figure 20.6 helps you understand the circuit's important attributes.



|  | S | R | Q | Q$^+$ |  |
|---|---|---|---|---|---|
| ⓐ | 0 | 0 | 0 | 0 | hold |
|  | 0 | 0 | 1 | 1 |  |
| ⓑ | 0 | 1 | 0 | 0 | clear |
|  | 0 | 1 | 1 | 0 |  |
| ⓒ | 1 | 0 | 0 | 1 | set |
|  | 1 | 0 | 1 | 1 |  |
| ⓓ | 1 | 1 | 0 | 0 | forbidden |
|  | 1 | 1 | 1 | 0 |  |

|  | S | R | Q$^+$ |  |
|---|---|---|---|---|
| ⓐ | 0 | 0 | Q | hold |
| ⓑ | 0 | 1 | 0 | clear |
| ⓒ | 1 | 0 | 1 | set |
| ⓓ | 1 | 1 | 0 | forbidden |

**(a)** **(b)**

**Figure 20.5: Useful form of the truth table of Figure 20.4 in normal (a) and compressed form (b).**

| Row | Comment |
|---|---|
| (a) | This is the do-nothing state[3], though we refer to this state as the hold condition. A hold condition is evident from examining the **Q** and **Q**$^+$ columns of the two (a) rows of Figure 20.5(a). The output does not change from the present state (**Q**) to the next state (**Q**$^+$), so the present state is being “held”. The next state is dependent on the present state since it’s the next state that is being “held”; the output can be held in either the ‘1’ or ‘0’ state. The state change (**Q** → **Q**$^+$) associated with these input conditions are 0 → 0 & 1 → 1[4]. |
| (b) | This is the clear state, or reset state. In this state, the next state (**Q**$^+$) is always ‘0’ independent of the present state (**Q**), so if the **S** and **R** inputs are equal to ‘0’ and ‘1’, respectively, the next state of the circuit is a ‘0’. The word “clear” is important in digital design; as a noun, it refers to the ‘0’ condition of a circuit output. Therefore, if the circuit output is cleared, the circuit output is currently in a ‘0’ state. As a verb, clear refers to the placing the circuit output into the ‘0’ state. Clearing a sequential circuit refers to the act of making the circuit output a zero. The state changes (**Q** → **Q**$^+$) associated with the **SR** = “01” inputs are 0 → 0 & 1 → 0. |
| (c) | This state is the set state. In this state, the next state (**Q**$^+$) is always ‘1’ and is independent of the present state (**Q**) of the circuit. If the **S** and **R** inputs are equal to ‘1’ and ‘0’, respectively, the next state of the circuit is a ‘1’. The word “set” is another important word in digital design. As a noun, the word set refers to the ‘1’ condition of a circuit output. If a circuit output is set, the output is currently a ‘1’. As a verb, “set” refers to the action of placing the circuit output into the ‘1’ state. Setting the circuit refers to the act of making the output a ‘1’. The state change (**Q** → **Q**$^+$) associated with the inputs conditions are 0 → 1 & 1 → 1. |
| (d) | This is the forbidden state. We soon mention the reason we refer to this state as forbidden. To stay out of the forbidden state, you need to make sure your **S** and **R** circuit inputs do not simultaneously have the values of ‘1’. Nothing dangerous happens if this condition occurs in your circuit but the digital gods will be annoyed. |

**Table 20.2: Detailed explanation of the main points in Figure 20.5.**

The truth table of Figure 20.5(a) becomes clearer by compressing it. Figure 20.5(b) shows the compressed truth table for the circuit of Figure 20.3. The output in the **Q**$^+$ column of Figure 20.5(b) is (a)**Q**, (b)0, (c)1, and (d)0. The **Q** in the (a) row refers to the fact that the next state (**Q**$^+$) is the same as the present state (**Q**). The ‘0’ and ‘1’ in the (b) and (c) rows refer to the fact that the next state will always be ‘0’ and ‘1’, respectively. It does not matter what’s going on in the (d) state since you should not be there.

The true ramifications of Figure 20.5 are not obvious. Recall that we’re describing a sequential circuit, which has the ability to remember a single bit. Instead of speaking of what the circuit is remembering, we refer to the “state” of the circuit. Figure 20.6 shows a timing diagram that tells the whole story; Table 20.3 provides an analysis of that timing diagram.

[3] Unfortunately, academic administrators spend most of their lives in this state.

[4] Generally speaking, we refer to the state changes of 0 → 0, and 1 → 1 as state changes even though the output does not really change.

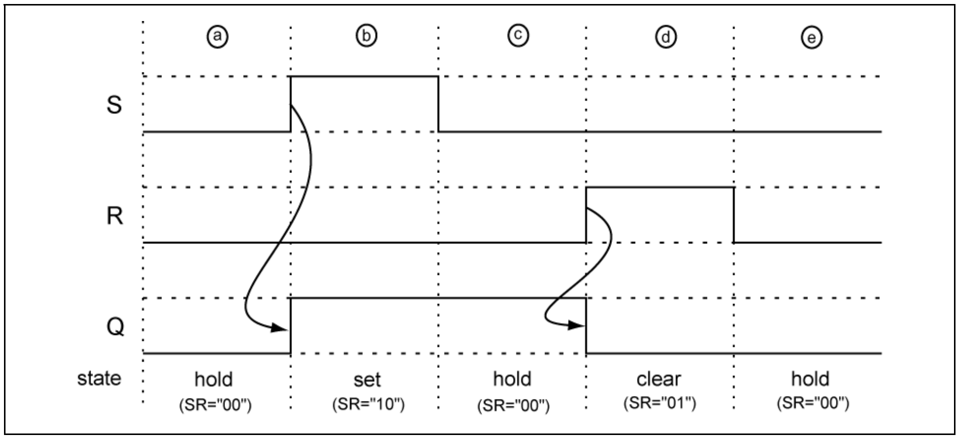


**Figure 20.6: A timing diagram showing the three conditions and two states of the given circuit**.

| time slot | Comment |
|---|---|
| **(a)** | This is a hold condition, which means that the present-state (**Q**) does not change as long as both the **S** and **R** inputs are both ‘0’. The timing diagram provides an arbitrary initial value for **Q**. |
| **(b)** | The **S** input sets at the beginning of the (b) time slots. The values of **S** and **R** are now ‘1’ and ‘0’, respectively. The **SR** = “10” represents the set condition for the circuit and causes the output **Q** to transition from ‘0’ to ‘1’. The output state remains set while **SR** = “10”. |
| **(c)** | The **S** input clears at the beginning of the (c) time slot. The **SR** inputs now equals “00”, which is the hold condition, so the output at the start of the (b) time slots remains set. The fact that the ‘1’ remains on the output after the **SR** inputs are both cleared represents the circuit’s memory. |
| **(d)** | The **R** input sets at the beginning of the (d) time slot. The **SR** is now “01”, which is a clear condition, so the output **Q** transitions from ‘1’ to ‘0’. The output of the device is “cleared” by this action and remains cleared as long as the **SR** = ”01” remains on the circuit inputs. |
| **(e)** | At the beginning of the (e) time slots, the **R** input clears. Since the both the **S** and **R** inputs are once again ‘0’, a hold condition is present on the circuit inputs. This hold condition causes no change from the present circuit outputs; the circuit is thus remembering a ‘0’. |

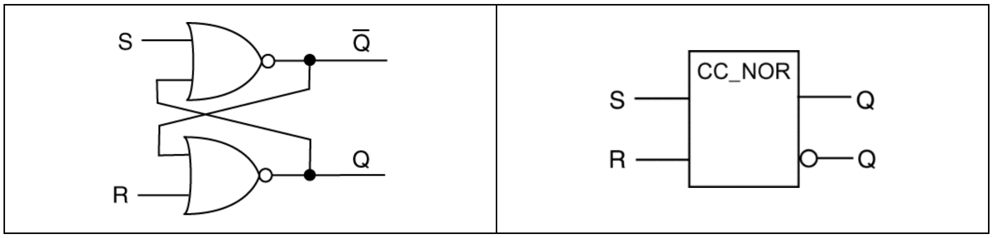
**Table 20.3: Detailed description of the timing diagram in Figure 20.6.**

The timing diagram in Figure 20.6 only provides the **Q** output. Recall that when we first model the original circuit, there was both a **Q** and a **Q**$^+$ value. The concept of **Q** and **Q**$^+$ enables us to model the present and next state of the circuit, respectively, but it wouldn’t provide any useful information to include both a **Q** and **Q**$^+$ in a timing diagram because the **Q** value inherently contains both of these values. For any given time in Figure 20.6, the present state is the state of **Q** at those times. The next state is the state of the circuit after the present state (a concept that is hard to describe in terms of a timing diagram).

## 20.4 The NOR Latch

Figure 20.2 shows a classic circuit in digital design, but Figure 20.7(a) shows the more common depiction of this circuit. While the diagram of Figure 20.2 only shows the **Q** output, the circuit in Figure 20.7(a) has both a **Q** output and **!Q** output. One common name for this circuit is the cross-coupled NOR cell. We also refer to this circuit as a NOR latch, or simply latch.

Once you draw the circuit in Figure 20.7(a) a few times, you'll instead abstract it to a higher-level and draw it as the BBD in Figure 20.7(b). The lower output of the diagram in Figure 20.7(b) contains a bubble on one of the **Q** outputs to indicate that it is active low. The **Q** output of the NOR cell is available in both positive logic and negative logic forms. How convenient.



**(a)** **(b)**

**Figure 20.7: The cross-coupled NOR cell (a) and its black box representation (b).**

### 20.4.1 Latch Terminology

The term "latch" is a common term in digital design. The term latch is similar to "set" and "clear" in that it has two different definitions depending on whether you're using the word as a verb or a noun. As a noun, a "latch" represents a one-bit level-sensitive storage element. As a verb, the notion of "to latch something" means to store a given digital value into a storage element. For the verb version of "latch", the storage element is not limited to a single-bit storage element.

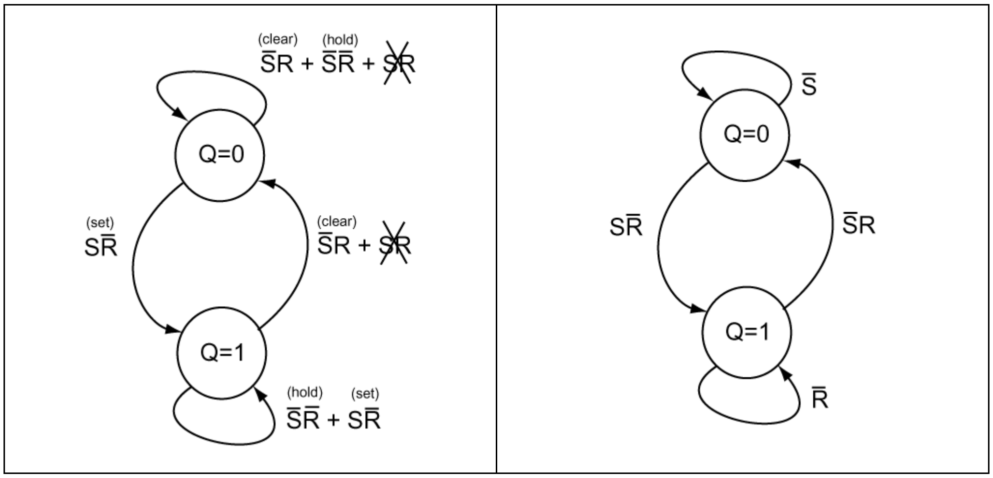
## 20.5 State Diagrams

We can enhance our understanding of sequential circuits using a state diagrams. While tabular representations of sequential circuits are interesting, they can be hard to interpret, especially as circuits become more complex. Truth tables do not present information efficiently; humans are more adept at viewing images such as state diagrams.

State diagrams are often the most useful way to describe the operation of sequential circuits. State diagrams are relatively simple, but they require learning a new terminology and symbology. We use state diagrams extensively, so we first examine one in the context of the simplest sequential circuit: the NOR latch. We present state diagrams in more detail in later chapter.

Figure 20.8 shows two versions of the state diagram associated with the NOR latch. An explanation follows but first we must issue this disclaimer. Unlike syntactical languages such as C or Java, there are no set rules for drawing state diagrams. Here is the one rule you should follow: good state diagrams transfer the most information in the shortest amount of time to the entity examining the state diagram. There are many ways to draw state diagrams. If you use strange techniques to draw your state diagrams, be sure to adequately explain them[5].

[5] In other words, be sure to annotate your state diagrams.


(clear) (hold)
$\overline{S}R + \overline{S}\overline{R} + \cancel{SR}$
Q=0
(set)
$S\overline{R}$
(clear)
$\overline{S}R + \cancel{SR}$
Q=1
(hold) (set)
$\overline{S}\overline{R} + S\overline{R}$
$\overline{S}$
Q=0
$S\overline{R}$
$\overline{S}R$
Q=1
$\overline{R}$

**(a)** **(b)**

**Figure 20.8: Two state diagrams representing the NOR latch.**

The state diagrams in Figure 20.8 completely describes the operation of the NOR latch. This means that the information in Figure 20.8 is the same information in Figure 20.5, but with a different presentation. State diagrams have some important features. Here they are:

- Each circle in state diagram refers to a different state in the circuit; we refer to these circles as *state bubbles*. The NOR cell stores one bit of information, there are two states in the associated state diagram: the **Q**=0 and the **Q**=1 state.
- The singly directed arrows, or just *arrows* in the state diagram represent the state transitions. There are four possible state transitions in the NOR cell: 1) 0→0, 2) 0→1, 3) 1→0, 4) 1→1. We represent the 0→0 and 1→1 transitions by *self-loops* in the diagram (arrows ending in the same state they started from). We represent the other transitions by arrows emanating from one state bubble and ending in another state.
- Each state transition (arrow) includes a list of conditions that allows that state transition to occur. We can represent these conditions in a variety of forms; the forms in the circuit of Figure 20.8(a) happen to be a logic-type form where the "+" symbol represents a logical OR.
- We include the forbidden states of the circuit inputs in Figure 20.8(a), but we cross them out. We include it the state diagram for completeness.
- Figure 20.8(a) shows there are eight product terms, which correspond to the eight rows of the truth table in Table 20.4(b).
- The state diagram of Figure 20.8(b) is functionally equivalent to the state diagram of Figure 20.8(a) but we reduced some Boolean equations and didn't include the forbidden states.

## 20.6 PS/NS Tables

The *present state/next state table* (*PS/NS table*) is another important sequential design element. This table is essentially nothing more than a truth table that lists both the present and next states of the circuit. The PS/NS table is a common tool in describing relatively simple digital circuits such as the NOR latch.

We already worked with a PS/NS table in the design of the NOR latch; the table of Figure 20.4(b) is a basic PS/NS table, which represents a more formal presentation of the table in Table 20.4(b). We often refer to the PS/NS table to as a *characteristic table* since they completely define the set of characteristics of a given device.

| PS/NS Table NOR Latch | | | | |
|---|---|---|---|---|
| | | (PS) | (NS) | |
| **S** | **R** | **Q** | **$Q^+$** | **Comment** |
| 0 | 0 | 0 | 0 | hold condition |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | reset condition |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | set condition |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 0 | x | forbidden |
| 1 | 1 | 1 | x | |

**Table 20.4: The PS/NS table for the NOR latch.**

## 20.7 Excitation Tables

*Excitation tables* are useful for describing the operation of some sequential devices. Excitation table are straightforward in that they represent a rearranging of the columns in a compressed PS/NS table for a given device. The excitation table provides is a list of input conditions that cause a given state transition. We list the state transitions as the change from the present state (**Q**) to the next state (**$Q^+$**); we the list the input conditions that allow those state transitions to occur. Figure 20.9(a) shows a compressed PS/NS table for a NOR latch while Figure 20.9(b) shows the associated excitation table. Table 20.5 provides a detailed description of the excitation table.

| S | R | $Q^+$ |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | X |

**(a)**

| state transitions | | input conditions | | |
|---|---|---|---|---|
| **Q** | **Q+** | **S** | **R** | **Comment** |
| 0 | 0 | 0 | - | (a) |
| 0 | 1 | 1 | 0 | (b) |
| 1 | 0 | 0 | 1 | (c) |
| 1 | 1 | - | 0 | (d) |

**(b)**

**Figure 20.9: A compressed PS/NS table (a) and an excitation table (b) for a NOR latch.**

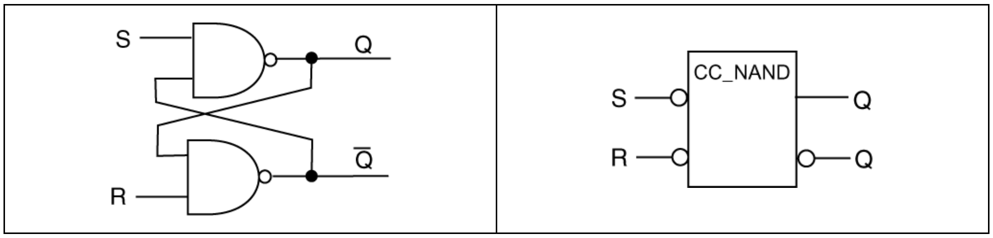
| row | state change<br>$Q \rightarrow Q^+$ | Comment |
|---|---|---|
| **(a)** | 0 → 0 | Two **SR** input conditions cause this state transition: either a hold condition (**SR** = “00”) or a clear condition (**SR** = “01”). Thus, this state transition occurs when the **S** input is ‘0’; the **R** input does not matter because the state change occurs when **R** is either a ‘1’ or a ‘0’. |
| **(b)** | 0 → 1 | Only one **SR** input combination that causes this transition: **SR** = “10”. This is the “set condition” of the NOR latch. This state transition occurs when the **SR** inputs are in the “01” state. |
| **(c)** | 1 → 0 | Only one **SR** input combination causes this transition: **SR** = “01”. This is the clear condition of the NOR latch. We sometimes refer to the clear condition as a “reset condition”. |
| **(d)** | 1 → 1 | Two **SR** input conditions cause this state transition: either a hold condition (**SR** = “00”) or a set condition (**SR** = “10”). This state transition occurs when the **R** input is ‘0’; the **S** input does not matter because the state change occurs when the **S** input is either a ‘1’ or a ‘0’. |

**Table 20.5: An explanation of the NOR cell excitation table of Figure 20.9(b).**

## 20.8 The NAND Latch

Since we’ve gone through the design and description steps for the NOR latch at a detailed level, we won’t to go through the same steps for a similar bit-storage circuit known as the NAND latch. There are many similarities in the derivation of the NOR and NAND latch, so we leave the derivation of the NAND latch as an exercise.

Figure 20.10(a) shows a diagram of the NAND latch. There is one major difference between the NOR and NAND latches: the inputs to the NOR latch are active high while the inputs to the NOR latch are active low. Figure 20.10(b) uses bubbles to show that the inputs to the NAND latch are active low.



**(a)** **(b)**

**Figure 20.10: A circuit diagram of a NAND latch (a), and the associated schematic diagram (b).**

## 20.9 NOR and NAND Latch Summary

Table 20.6 provides the big summary of the various representations of NOR and NAND latches. We refer to both the NOR or NAND latches as a “SR latches” based on the fact that they can “set” and “reset”. The ramifications of the SR latch is that if someone mentions “SR latch”, you’ll know that you’re dealing with a 1-bit storage element but you will not know whether it is a NOR or NAND latch.

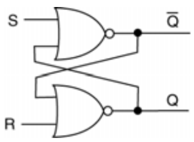
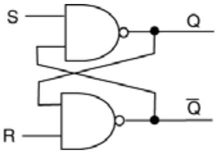
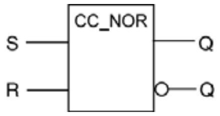
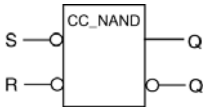
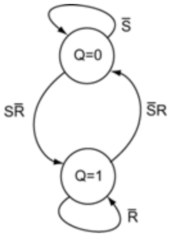
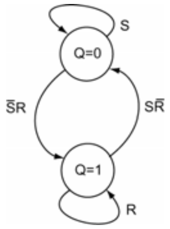
<table>
<tr><th>Item</th><th>NOR Cell</th><th>NAND Cell</th></tr>
<tr><td><b>Circuit Form</b></td><td> </td><td> </td></tr>
<tr><td><b>PS/NS Table</b></td><td>
<table>
<tr><th>S</th><th>R</th><th>Q</th><th>$Q^+$</th><th>Comment</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td rowspan="2">hold</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><td rowspan="2">reset</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><td rowspan="2">set</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>x</td><td rowspan="2">forbidden</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>x</td></tr>
</table>
</td><td>
<table>
<tr><th>S</th><th>R</th><th>Q</th><th>$Q^+$</th><th>Comment</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>x</td><td rowspan="2">forbidden</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>x</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td rowspan="2">set</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td rowspan="2">reset</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td rowspan="2">hold</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td><b>Compressed PS/NS Table</b></td><td>
<table>
<tr><th>S</th><th>R</th><th>$Q^+$</th></tr>
<tr><td>0</td><td>0</td><td>Q</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>x</td></tr>
</table>
</td><td>
<table>
<tr><th>S</th><th>R</th><th>$Q^+$</th></tr>
<tr><td>0</td><td>0</td><td>x</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>Q</td></tr>
</table>
</td></tr>
<tr><td><b>Excitation Table</b></td><td>
<table>
<tr><th>Q</th><th>$Q^+$</th><th>S</th><th>R</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>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>-</td><td>0</td></tr>
</table>
</td><td>
<table>
<tr><th>Q</th><th>Q+</th><th>S</th><th>R</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>-</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>-</td><td>1</td></tr>
</table>
</td></tr>
<tr><td><b>Block Diagram</b></td><td> </td><td> </td></tr>
<tr><td><b>State Diagram</b></td><td> </td><td> </td></tr>
</table>

**Table 20.6: The Big Summary of NOR and NAND latches.**

## 20.10 Chapter Overview

- The two main types of digital circuits include combinatorial circuits and sequential circuits. Sequential circuits have the ability to store bits of information while combinatorial circuits do not. Sequential circuits obtain their memory storage ability circuits by feeding output signals back to the circuit's inputs, a condition we refer to as *feedback*.
- Latches are the most basic storage elements in digital logic. The two main types of latches are the NOR latch and the NAND latch. Although we construct these latches with different logic gates, the only difference between these two latches at a higher-level is the logic levels of the **S** and **R** inputs.
- Since sequential circuits can store information, we consider them as having a *state*. The data a sequential circuit stores determines the state of the circuit. We use the concept of *present state* and *next state* to describe changes in the values stored by the circuit at the present time.
- We typically describe sequential circuits by PS/NS tables, characteristic equations, excitation tables, and state diagrams. Probably the most useful of these representations is the state diagram. Transitioning from any of these representations to any other of these representations is a straightforward process.
- We consider a latch as an level-sensitive device since the outputs can change any time the inputs change. When we add special control inputs to latches, name a clock input, and changes in the state of the circuit can only happen when certain conditions are present on the circuit inputs..

## 20.11 Chapter Exercises

**1)** Briefly describe what is meant by the term “state” in a digital circuit.

**2)** Briefly describe what specific condition gives a digital circuit the ability to have state.

**3)** Briefly describe why the “forbidden” state is considered forbidden.

**4)** Briefly explain what is the worst thing that could possible occur if your circuit finds itself in the forbidden state.

**5)** Briefly define the word “set” as both a verb and a noun.

**6)** Briefly define the word “clear” as both a verb and a noun.

**7)** Derive the tables of Figure 20.5 for a NAND latch.

**8)** Provide an accepted synonym for the word “clear” as it relates to digital design.

**9)** Briefly describe why is it hard to describe the concept of “next state” in a timing diagram.

**10)** Briefly describe the main attribute of a good state diagram.

**11)** Briefly the physical circuit characteristic that created the notion of “memory” in a circuit.

**12)** Briefly describe why it makes no sense to describe the “state” of a combinatorial circuit.

## 20.12 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware in your solution
- State the types of control your circuit uses ("no control", "internal control", and/or "external control").

**1)** Design a circuit that contains a NOR latch and only allows the outputs to change when the button is pressed. The circuit's inputs and outputs should be the same as a NOR latch except for the addition of the button input. Assume a button press equals a logical '1'.

**2)** Design a circuit that contains a NAND latch and only allows the outputs to change when the button is pressed. The circuit's inputs and outputs should be the same as a NAND latch except for the addition of the button input. Assume a button press equals a logical '1'.

**3)** Design a circuit that adds two 8-bit inputs in RC format. The circuit outputs A+B or A-B. The outputs A+B when BTN1 is pressed and keeps outputting that value until BTN2 is pressed. When BTN2 is pressed, it outputs A-B and keeps outputting that value until BTN1 is pressed again, at which time it outputs A+B again. Assume BTN1 and BTN2 will never be pressed simultaneously. Assume the result of the mathematical operations will always be valid.

# 21 Flip-Flops

## 21.1 Introduction

The previous chapter's introduction to sequential circuits entailed the simple latch. This chapter presents the notion of a one-type of flip-flop[1], which is nothing more than a latch with added control features. The notion of flip-flops is somewhat "dated" as modern digital design no longer uses all flavors of flip-flops.

**Main Chapter Topics**

**The Flip-Flops:** This chapter describes the basic operation of a D flip-flop, which is nothing more than an edge-sensitive latch.

**Chapter Acquired Skills**

- Be able to describe the basic terminology associated with clocked digital circuits
- Be able to describe the basic operation of a D flip-flop
- Be able to describe the difference between synchronous and asynchronous flip-flop inputs
- Be able to describe the associated features of D flip-flops as they apply to timing diagrams

## 21.2 Clock Vernacular

In order to understand this chapter, we first must toss out some definitions regarding "clock" signals. We fill in more details in a later chapter.

- Unless stated otherwise, clocks are *periodic* signals, which means that the output waveform of the signal repeats itself after a finite amount of time.
- We refer to the time it requires for periodic signals to repeat themselves as the *period*.
- A periodic signal by nature some amount of time high and some amount of time low. We refer to the transition from low-to-high as the "rising edge" of the signal, and refer to the transition from high-to-low as the "falling edge" of the signal.
- We refer to the ratio of the time the signal is high to the time the period of the signal as the *duty cycle*. The duty cycle is unit-less value.

[1] There are three standard types of flip-flops out there; this text only deals with one type: the D flip-flop.

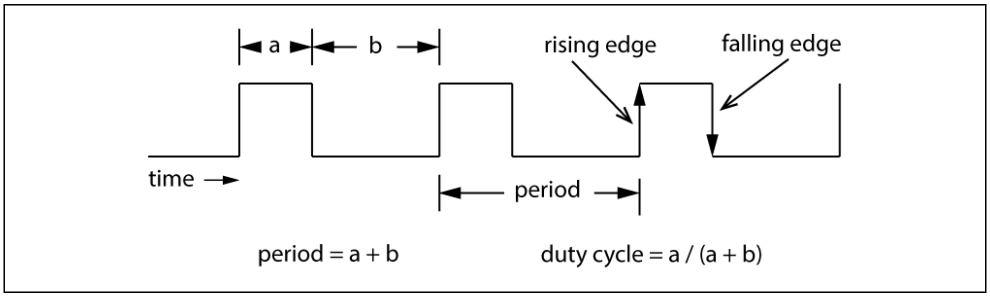


**Figure 21.1: Everything you wanted to know about periodic signals but were afraid to ask.**

## 21.3 Flip-Flops

A flip-flop is essentially an edge-sensitive latch, which means that the flip-flop's outputs can only change simultaneously to an active edge of a particular input signal. The primary difference between flip-flops and latches is that fact that latches are level-sensitive (state can change anytime inputs change) while flip-flops are edge sensitive (state changes are synchronized with an active edge of a control signal).

Most sequential devices have an input that we refer to as a *clock*; a flip-flop bases its edge-sensitivity on that clock edge. Additionally, we refer to flip-flops as synchronous circuits, which means that the changes to the state of the circuit are synchronized with an active clock edge. The active clock edge can be either the rising or falling edge of the clock. We refer to devices whose state changes on the rising clock edge as "rising-edge-triggered" devices (RET) and devices whose state changes on the falling clock edge as "falling-edge-triggered" devices (FET)[2].

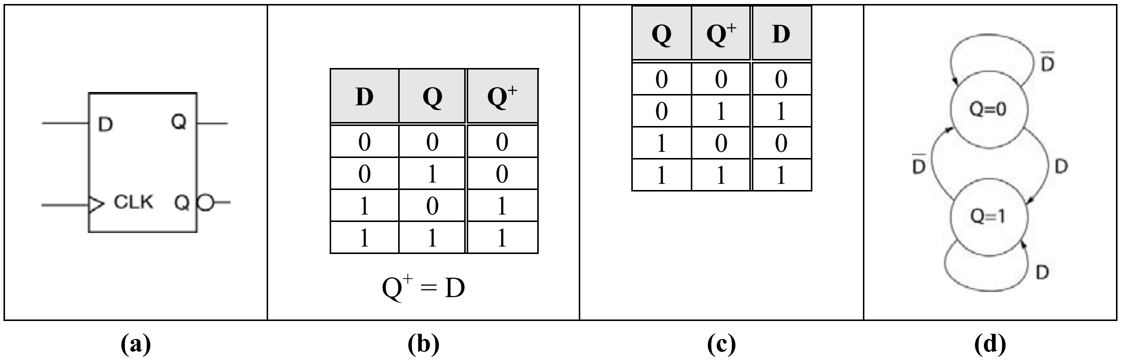
## 21.4 The D Flip-Flop

The most common flip-flop is the D flip-flop. The **D** stands for *data*, so the D flip-flop is a data flip-flop. The characteristic of a **D** flip-flop is that the output of the flip-flop follows the **D** input. Figure 21.2(a) shows a schematic symbol for a simple **D** flip-flop. The new feature to notice about the **D** flip-flop symbol is the triangular shape on the **CLK** input, which means that the device is edge-triggered. 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, we would consider this device a falling-edge-triggered (FET) device. The standard **D** flip-flop outputs both the positive and negative logic versions of the flip-flop's state (**Q** & **!Q**).

Figure 21.2(b) shows the characteristic table of the **D** flip-flop, which shows that the next state (**$Q^+$**) of the flip-flop follows the **D** input to the flip-flop. By inspection of the characteristic table, you can generate the characteristic equation in Figure 21.2(b). Figure 21.2(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 (**$Q \rightarrow Q^+$**) to occur.

Figure 21.2(d) shows the state diagram that models the **D** flip-flop. This state diagram looks very similar to the state diagram for the latch presented earlier; however, there is one significant: because the **D** flip-flip is a synchronous device, the state of the flip-flop can only change on clock edges. This means that the singly directed arrows in Figure 21.2(d) are *implicitly associated with the clock edge*. Unless stated otherwise, from now on, state-to-state transitions in state diagrams are synchronized with the active clock edge.

[2] Most digital logic texts include the derivation of the actual circuitry that creates the "edge sensitivity". The circuitry that creates edge triggering is somewhat complicated, so we omit it in favor of keeping things abstracted to higher levels.



| 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 |

**Figure 21.2: The schematic symbol (a), characteristic table and characteristic equation (b), and excitation table (c), and the state diagram for the D flip-flop.**

The timing diagram in Figure 21.3 demonstrates the operation of the RET D flip-flop in Figure 21.2(a); here's a list of the items to note in the timing diagram:

- Outputs can only change on the rising-edge of the clock signal. Figure 21.3 uses vertical dotted lines to show the rising edges of the clock across the signals.
- 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', which transfers to the output and becomes the official "state" of the flip-flop. At the second rising edge, the D input is high again so no state change occurs.
- During the interval between the first and second rising edges, the **D** input changes twice. The circuit ignores these changes because the output can only change on the 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.

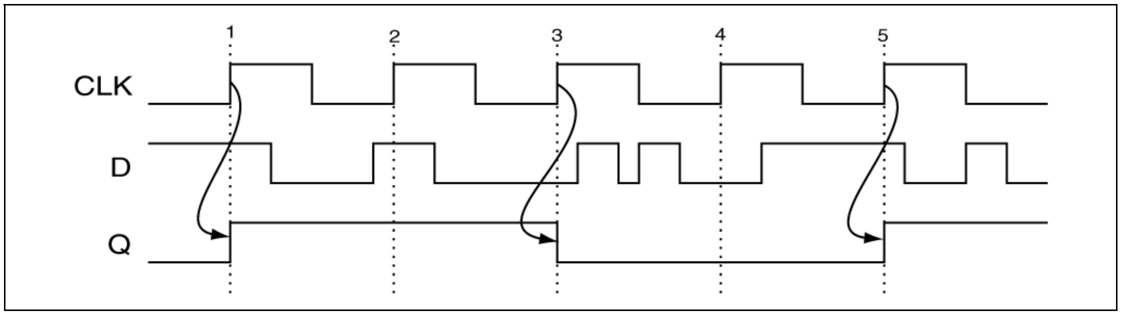


**Figure 21.3: An example timing diagram for the D flip-flop.**

## 21.5 Synchronous and Asynchronous Flip-Flop Inputs

The flip-flops we've described up to this point were 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. Many flip-flops have the ability to change state either synchronously (synchronized to the clock input) or "asynchronously". Asynchronous inputs cause state changes that are not synchronized with the clock edge.

There are two different things you can do to a flip-flop's output, namely *set it* or *clear it*. Not surprisingly, there are usually two different asynchronous control 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, the flip-flop can change state. Flip-flop diagrams use a bubble to indicate the logic level of the inputs. Flip-flops use an **S** and **R** inputs to represent signals that asynchronously set and clear the state of the flip-flop, respectively.

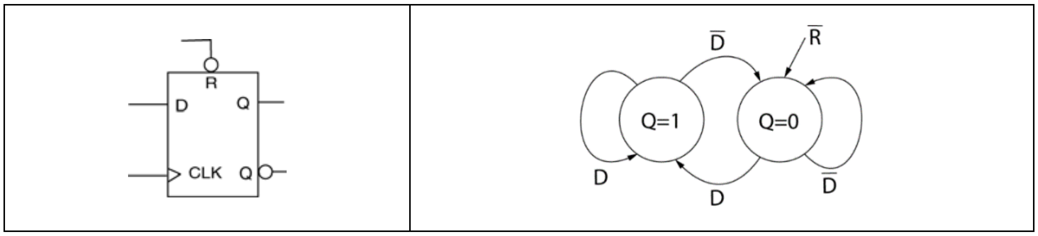
### 21.5.1 D Flip-Flop with Reset

A D flip-flop with an asynchronous input is relatively simple, so we model it with a state diagram in Figure 21.4(b). The schematic in Figure 21.4(a) shows most of what we need to know about the D flip-flop, but it does not show whether the **R** input is asynchronous or not; someone or something needs to state this fact. Here are a few other things to note.

- The flip-flop has complementary outputs, which the diagram in Figure 21.5(a) indicates with two **Q** outputs; the **Q** with the bubble is the active low version of **Q**.
- We generally assumed the **R** input to be a "reset" control input
- The **R** input has a bubble, which indicates that the input is active low
- While the **R** input is active low, someone needs to tell you whether it is synchronous or asynchronous. For this example, we assume the **R** input is asynchronous.

The state diagram in Figure 21.5(b) is nearly identical to the state diagram in Figure 21.2(d), but with one major difference: how we represent the **R** input. Here is a complete description of that difference.

- We represent asynchronous inputs with a new type of arrow. We represent synchronous state transitions with "state-to-state" arrows (starts in a state and ends in a state), while we represent asynchronous transitions with "coming-out-of-nowhere-to-state" arrows (starts nowhere, and ends in a state).
- We represent the fact that the **R** input is active low in the schematic by using a bubble on the input and by placing an overbar on the signal in the state diagram.



(a) (b)

**Figure 21.4: Timing diagram associated D Flip-flop with asynchronous active low reset.**

Figure 21.5 shows a timing diagram we use to model the operation of the flip-flop. For this timing diagram, assume the **R** input as precedence of the **D** input. Here are things to note about the timing diagram.

- Because the **R** input is low at the start of the timing diagram, the output of the flip-flop is in the reset state.
- On the first rising clock edge (1) , the D flip-flop acts as you expect. In this case, the model ignores the **R** input because it is a '1' (its non-active state); the model then evaluates the other inputs. Because the D input is '1' on this clock edge, '1' becomes the new state of the flip-flop.
- On the second clock edge, the flip-flop does not change state. The **D** input is at '1' on this clock edge, the flip-flop output does not change because the flip-flop is currently in the **Q**=1 state.

- Between the second and third clock edges, the **R** input asserts (goes low); so the flip-flop immediately resets[3], as the causality arrows indicate. The **R** input returns to its non-active state (the '1' state) soon afterwards. Returning the **R** input to the non-active state has no effect on the state of the flip-flop, meaning that the flip-flop stays reset.
- The pulses on **D** between the third and fourth clock edges have no effect because they did not occur on an active clock edge.
- On the fifth clock edge, the flip-flop sets as the causality arrows indicate. Soon after that clock edge, the flip-flop resets are a result of the **R** input becoming active (**R**='0').

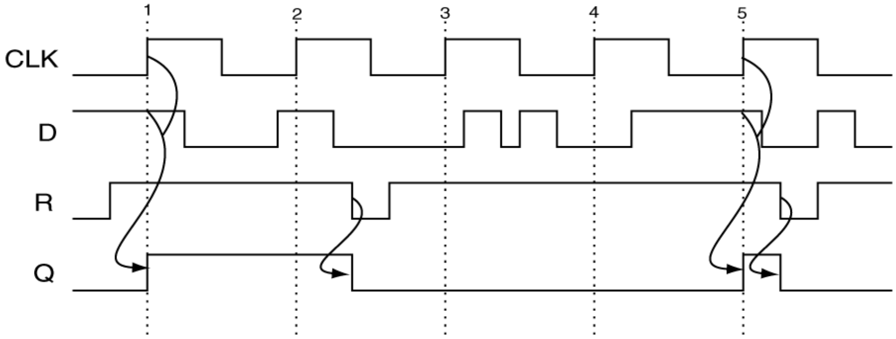


**Figure 21.5: Timing diagram associated D Flip-flop with asynchronous active low reset.**

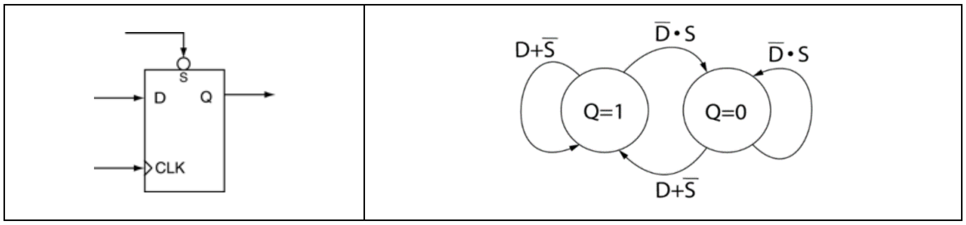
### 21.5.2 D Flip-Flop with Set Input

D flip-flops often have a control input that allows the flip-flop to be set. The **D** flip-flop in Figure 21.6(a) has an **S** input as a control signal that sets the flip-flop. We also need to state that the **S** input is synchronous. Figure 21.6(a) shows that the **S** input is active low and that the device is rising-edge-triggered.

The state diagram in Figure 21.6(b) is different from the state diagram in Figure 21.4(b) because the set input is synchronous. Because it's synchronous, the **S** control input is now associated with the state-to-state-type transition arrows rather than the coming-out-of-nowhere arrows. There are a few new issues to describe in the state diagram of Figure 21.6(b); these issues involve the presence of logic and the **S** control input in the state diagram. There are now two conditions associated with the state transitions: the **D** and the **S** inputs.

- The logic shows that we can attain the 1 → 1 transition in two different ways: when the **D** input is asserted (**D**='1') or when the S input is asserted (**S**='0').
- The logic shows that we can attain the 0 → 0 transition when the D input is not asserted (**D**='0') at the same time as when the **S** input is not asserted (S='1').
- The logic shows that we can attain the 0 → 1 transition in two different ways: when the D input is asserted (**D**='1') or when the **S** input is not asserted (**S**='0').
- The logic shows that we can attain the 1 → 0 transition when the **D** input is not asserted (**D**='0') at the same time as when the **S** input is not asserted (**S**='1').

[3] There is actually an associated propagation delay associated with this state transition but we're still modeling these flip-flops using an ideal model.



**(a)** **(b)**

**Figure 21.6: Timing diagram associated D Flip-flop with asynchronous active low set.**

Figure 21.7 shows a timing diagram associated with Figure 21.6. Here are a few things to note:

- The problem provides the starting state of **Q**, which the problem description did not state.
- The flip-flop ignores the **S** pulse between the first and second rising clock edge because the **S** input in this example is synchronous. 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.

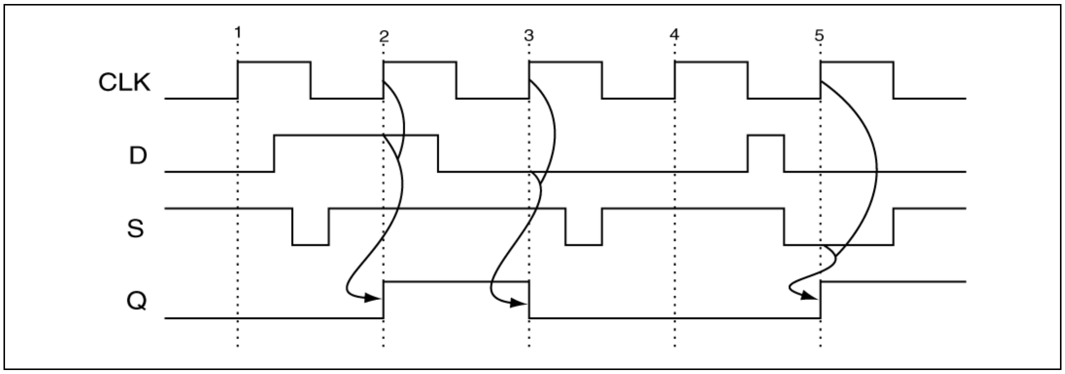


**Figure 21.7: Timing diagram associated with this example.**

## 21.6 Flip-flops with Multiple Control Inputs

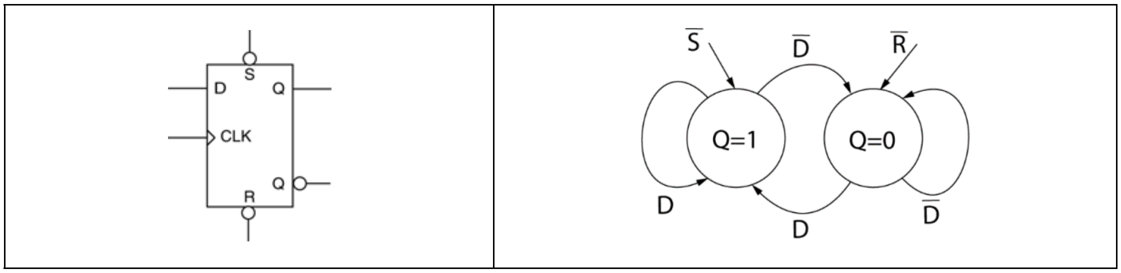
Flip-flops can also have multiple control inputs, such as the flip-flop in Figure 21.8(a). This flip-flop has active-low asynchronous set and reset inputs. Figure 21.8(b) shows the state diagram modeling the flip-flop's operation. Here are a few things to note about the state diagram.

- Both the **S** and **R** inputs are asynchronous and active low; the complemented signal names indication they are active low while the "arrow from nowhere" indicate the asynchronicity of the inputs.
- Problems such as this must state what happens when both the **S** and **R** inputs assert simultaneously; the state diagram does not make sense unless we provide this information. For this problem, we declare that the **S** and **R** inputs won't be simultaneously asserted.

We base the timing diagram in Figure 21.9 on the schematic diagram of Figure 21.8(a). Here are a few items of interest in the timing diagram.

- The output of the D flip-flop goes to the '1' state with the initial 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. When **R** returns to the ‘1’ state, the output of the device remains in the ‘0’ state.
- The second low pulse on **R** does not affect the state of the flip-flop since the flip-flop is current in a ‘0’ state.
- The final low pulse on the **S** signal sets the output of the flip-flop.



(a) (b)

**Figure 21.8: Timing diagram associated D Flip-flop with asynchronous active low clear.**

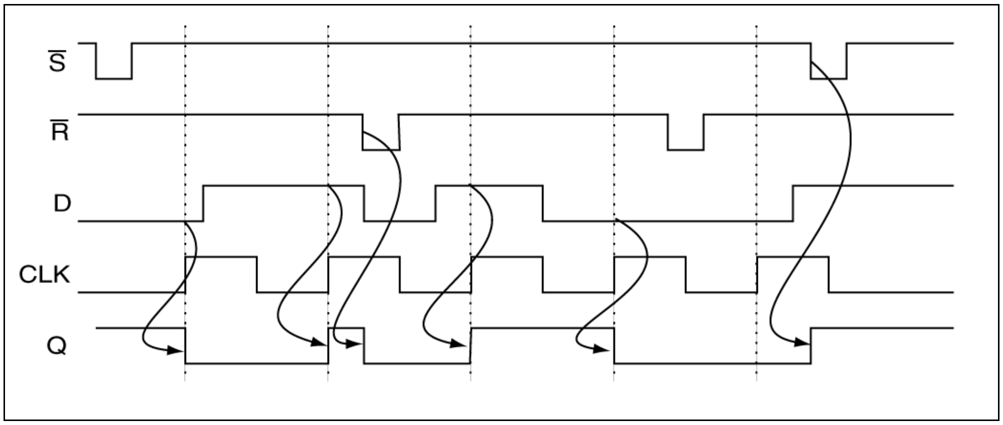


**Figure 21.9: Example timing diagram for a RET D flip-flop with active low asynchronous preset and clear for this example.**

## 21.7 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 a flip-flop. There are three main types of flip-flops: the D, T, and JK flip-flops; this text does not consider 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; we refer to these inputs to as asynchronous inputs.
- We use state diagrams to represent the operation of D flip-flops, which provide a visual description describing the operation of the device.
- State diagrams don't include clock signals, as we understand most transitions to be associated with the device's active clock edge. We represent asynchronous transitions with singly directed arrows emanating from nowhere and ending up in a state in the state diagram.

## 21.8 Chapter Exercises

For the following problems, assume all inputs and outputs are positive logic, unless stated otherwise.

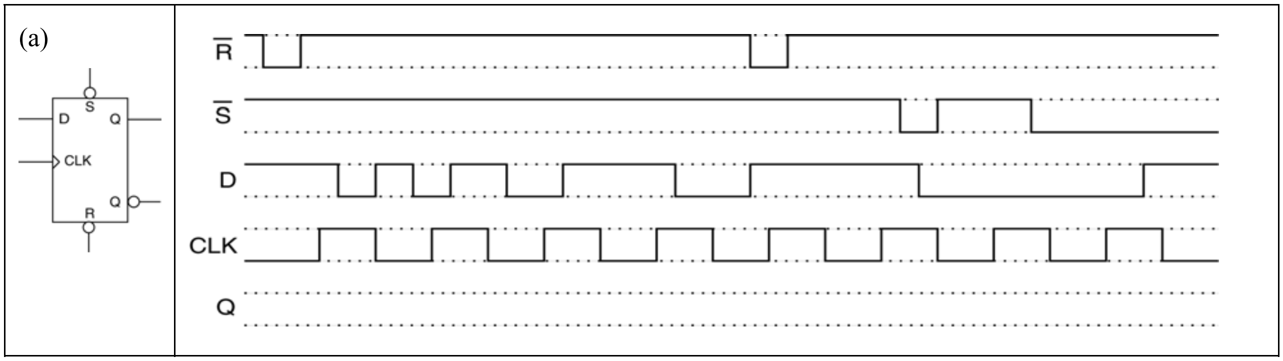
**1)** Briefly describe the difference between a flip-flop and a latch.

**2)** Briefly describe the difference between synchronous and asynchronous inputs on a D flip-flop.

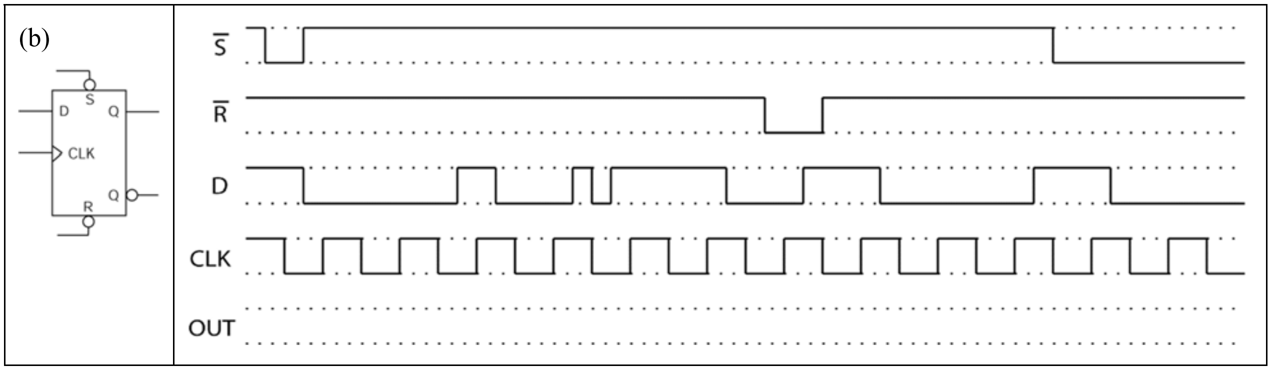
**3)** Briefly describe the what the “D” in D “flip-flop” stands for.

**4)** 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.

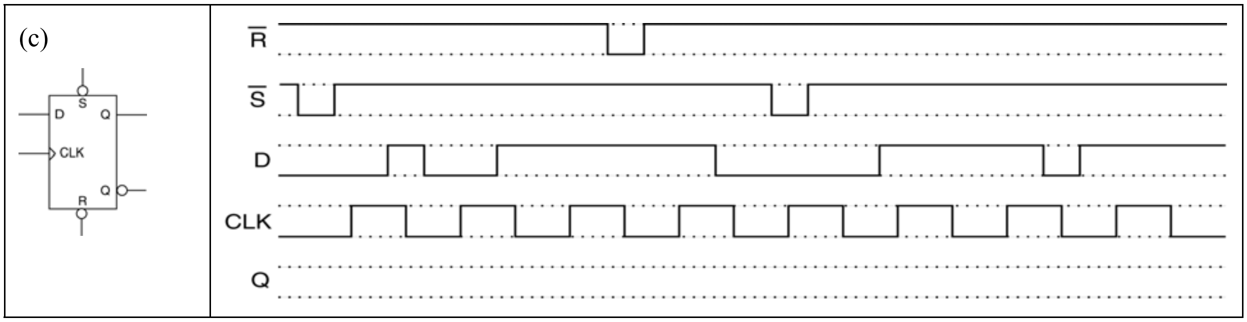
(a)

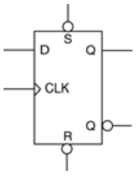
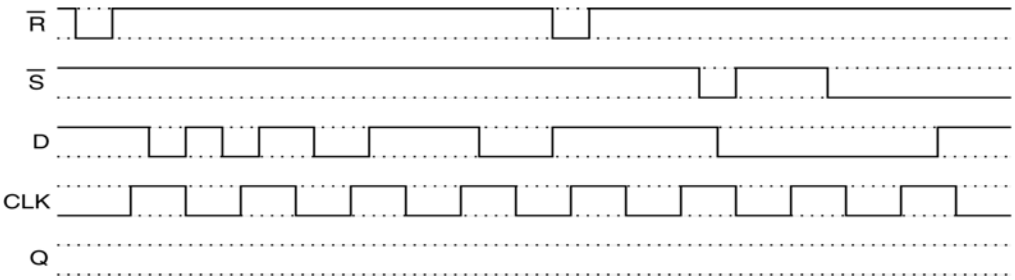
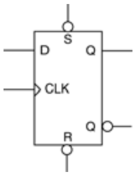
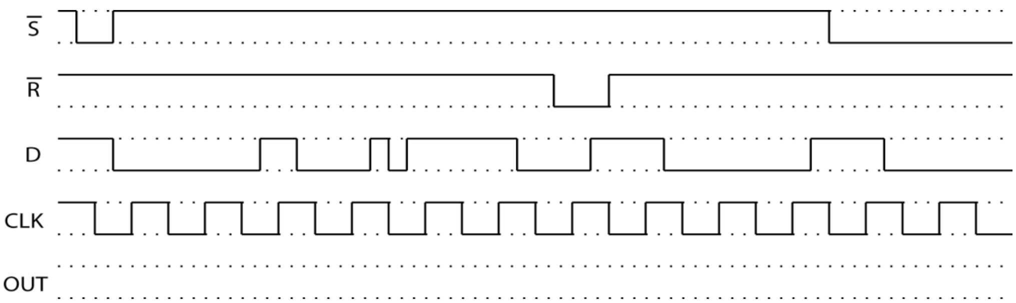
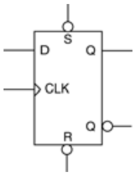
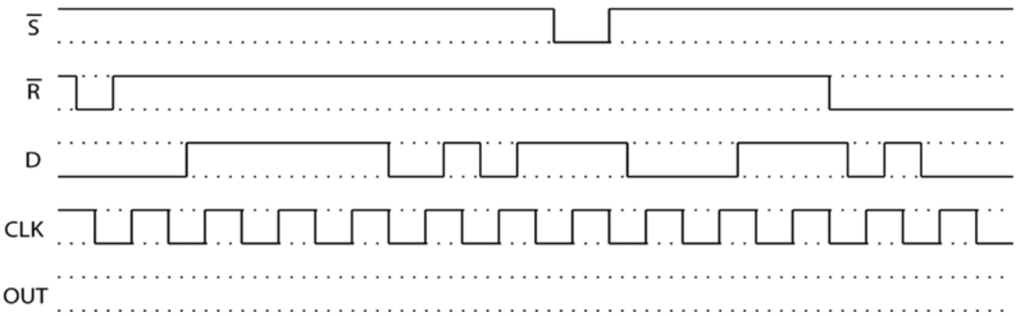


(b)

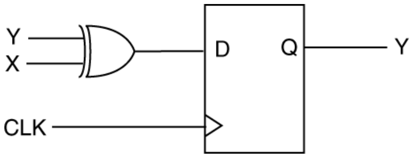


(c)



| | |
|---|---|
| (d)  |  |
| (e)  |  |
| (f)  |  |

**5)** Provide a state diagram and a PS/NS table that describes the following circuit.



**6)** Briefly describe the karmic potential of a D flip-flop.

**7)** Briefly describe the distinct relationship between D flip-flops and popsicles.

## 21.9 Design Problems

**1)** Using only one extra device, use a D flip-flop to blink an LED at half the D flip-flops clock frequency.

**2)** Design a circuit with two LED outputs. Both outputs blink at half the input clock frequency, but the two LEDs are never simultaneously on.

**3)** Use a D flip-flop to blink an LED at half a D flip-flop's clock frequency. This circuit also had a button that holds (prevents the outputs from changing) the circuit's outputs. Assume a pressed button outputs a '1'.

**4)** Design a circuit with two LED outputs. Both outputs blink at half the input clock frequency, but the two LEDs are never simultaneously on. This circuit also has a positive logic control input that turns off both LEDs when asserted. Assume a pressed button outputs a '1'.

**5)** Design a circuit with two LED outputs. Both outputs blink at half the input clock frequency, but the two LEDs are never simultaneously on. This circuit also has a negative logic control input that toggles both LEDs when asserted.

**6)** Design a circuit that shows the previous three values present on the VAL input on the rising edge of the circuit's clock input.

**7)** Design a circuit that shows when the previous three values present on the VAL1 and VAL2 inputs on the rising edge of the circuit's clock input are equivalent.

**8)** Design a circuit that has two data inputs: VAL1 & VAL2. The circuit also has a control input show chooses whether to display the previous three values on the VAL1 input, or the previous three values on the VAL2 input. The control signal is positive logic; a value of '0' on this signal directs the circuit to display the VAL1 associated sequence.

# 22 Registers

## 22.1 Introduction

Registers could be the most widely used circuit in digital design. The concept of registers is straightforward, particularly since you have already been working with a 1-bit register (the D flip-flop). This chapter describes multi-bit registers; we work with other common types of registers in later chapters.

**Main Chapter Topics**

**SIMPLE REGISTERS AND REGISTERS "WITH FEATURES"**: This chapter defines and describes basic multi-bit registers and their common features.

**Chapter Acquired Skills**

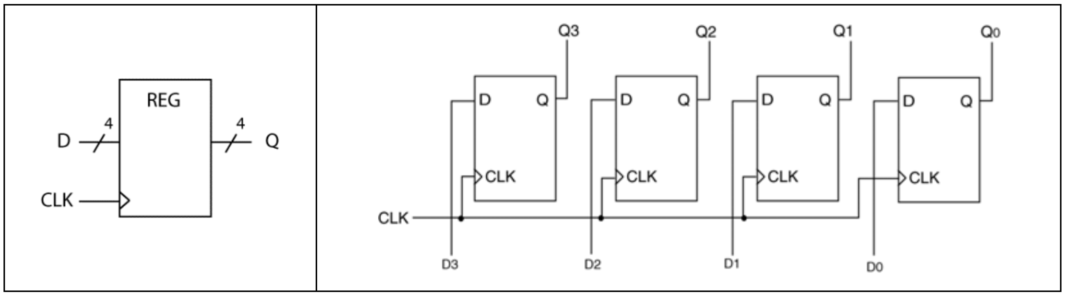
- Be able to describe the construction and function of a register in terms D flip-flops
- Be able to describe the basic synchronous nature of flip-flops.
- Be able use basic register control features such as load and clear
- Be able to describe the basic operation of registers in timing diagrams
- Be able to use simple registers in solutions of digital design problems

## 22.2 Registers

Registers are multi-bit storage elements modeled as a parallel configuration of D flip-flops that share a common clock signal. When we refer to "registers", we refer to simple registers; we refer to other common register types by their names: counters and shift registers (topics for later chapters).

Figure 22.1 shows four D flip-flops assembled to act as a simple multi-bit register. In particular, Figure 22.1(a) shows the block diagram for a 4-bit register and Figure 22.1(b) shows the underlying circuit. Here are a few things to note about Figure 22.1:

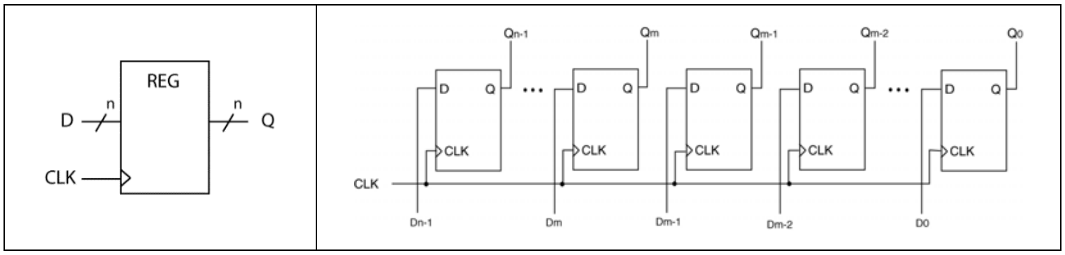
- The block diagram in Figure 22.1(a) shows that this register is rising-edge triggered. This means that all changes in the state of the register are synchronized with the rising clock edge.
- Figure 22.1(b) shows that each flip-flop in the register shares a common clock signal, which allows all flip-flops to simultaneously latch their data.
- We label the four input signals, Dx & Qx, respectively, with numbers. We often consider the left-most bit the MSB and the right-most bit the LSB.



(a) (b)

**Figure 22.1: A block diagram for a 4-bit register (a), and the associated lower-level model (b).**

Figure 22.2(a) shows the block diagram for a generic n-bit register; Figure 22.2(b) shows the underlying details. We typically model the LSB with an index of '0', and the MSB with an index of "n-1".

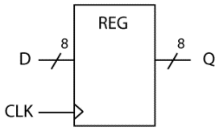


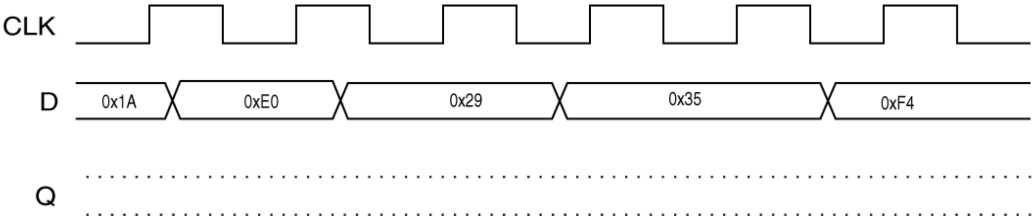
(a) (b)

**Figure 22.2: A block diagram of an n-bit register (a), and the underlying circuitry, (b).**

**Example 22.1: A Simple Register**

Using the block diagram on the right to complete the timing diagram provided below. 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. We only need to examine only the portions of the timing diagram aligned to the rising edge of the clock, which is when the register's input data latches into the register, thus allows the data to appear on the output. Figure 22.3 shows the solution for this example; here are a few items worth noting.

- The solution adds dotted vertical lines on the rising clock edges
- The problem did not provide an initial value (state) of the register, which is why the first value on the **Q** line contains question marks.

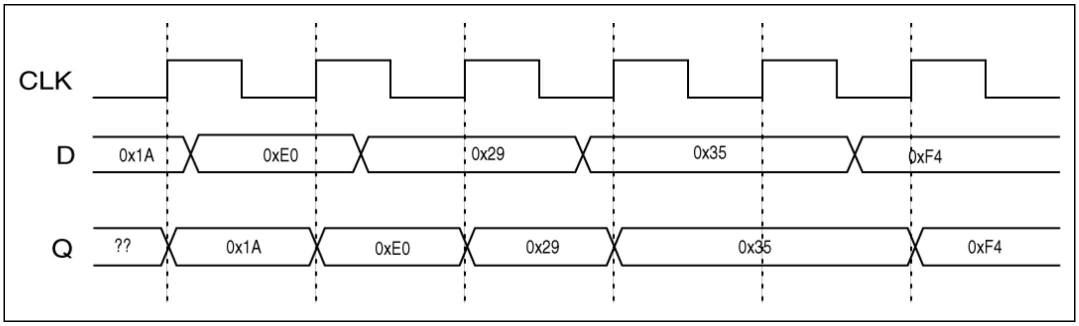


**Figure 22.3: The solution for this example.**

In real digital circuits, you rarely see registers as simple as the register in the previous example as they lack the "control" to make them useful. Useful registers that are more useful contain a signal that controls when the register latches the data, so that the register is not loading data on every active clock edge. Control signals for such registers are associated with the word "load" (and the acronym **LD**); the vernacular is registers "load" the input data into the register. Figure 22.4 shows an example of a register with a **LD** control. In order for this register to latch the input data into the register, it requires both an active clock edge and an asserted **LD** signal.

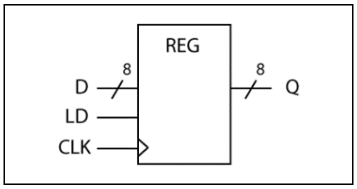
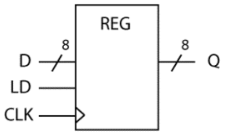
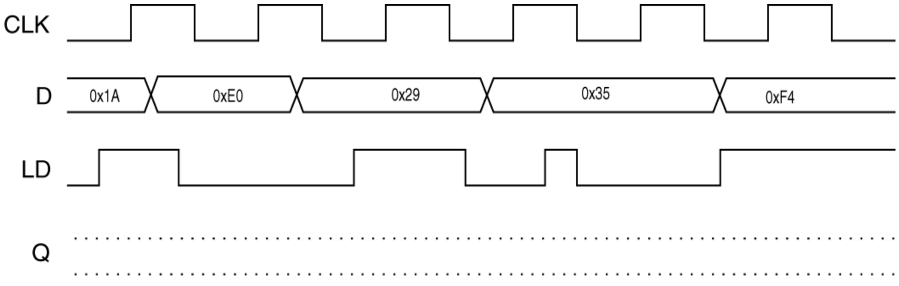


**Figure 22.4: A register with a load control (LD).**

**Example 22.2: A Register with Load Control**

Using the block diagram on the right to complete the timing diagram provided below. Ignore all propagation delay issues.





**Solution**: We need to examine the times where the both the rising edge of the **CLK** signal occurs and where the **LD** signal is asserted. Figure 22.5 shows the solution for this example; some interesting things to note surely follow as well. Here are a few more items of interest in Figure 22.5

- The **LD** signal is "level sensitive", so the register can only load the input data when the **LD** input is asserted at the same time there is an active clock edge.

- The first rising clock edge latches the data on the **D** input to the register because of the **LD** input being asserted.
- On the second rising clock edge, the state of register does not change because the **LD** input is not asserted.
- At the time marked with the circled "1", the **LD** signal asserts and then de-asserts, which has no effect on the register because there was no rising clock edge when the **LD** signal was asserted.

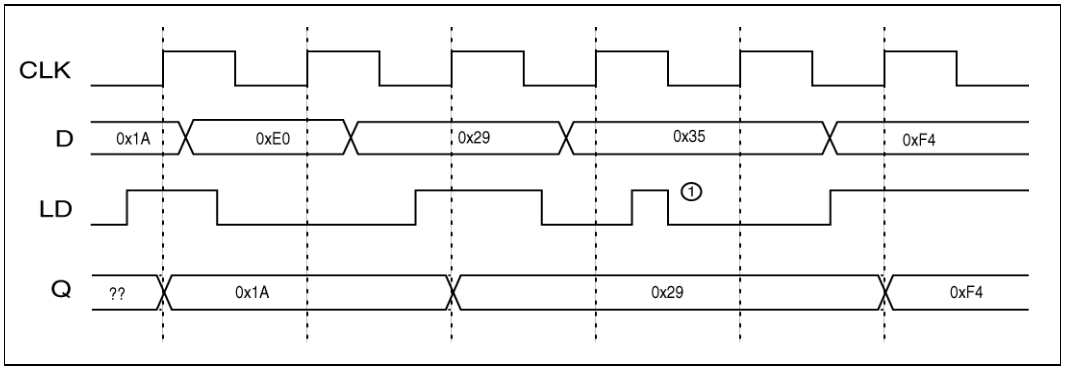


**Figure 22.5: The solution for Example 22.2.**

Registers can have other control inputs as well. Figure 22.6 shows a register that has both a load and a clear input. The **CLR** input, like the **LD** input, is a control signal. We can generally assume the register's **LD** signal be synchronous, while signals such as **CLR** are usually asynchronous. The **CLR** input, as Figure 22.6 shows, is active low as the bubble on the input indicates. We don't know from looking at Figure 22.6 whether the **CLR** signal is asynchronous or not; circuits you work with need to provide that information.

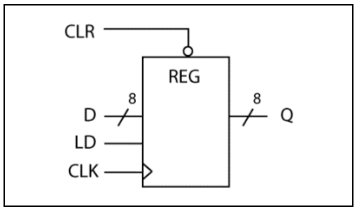
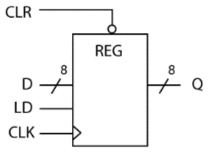
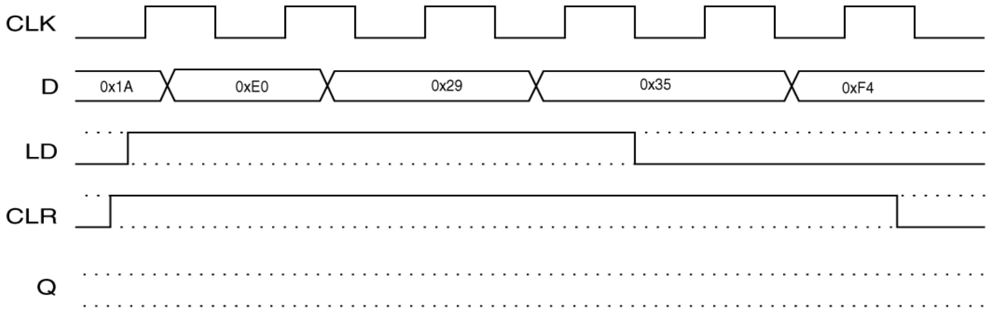


**Figure 22.6: A register with a load control and clear input (CLR).**

**Example 22.3: A Register with Synchronous and Asynchronous Control**

Using the block diagram on the right to complete the timing diagram below. 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. Ignore all propagation delays.





**Solution**: Although this solution is straightforward, it provides a few new tidbits of information regarding the operation of registers.

- The asserted **CLR** signal at the beginning of the timing diagram makes the register's state known; the register is initially storing zero, or "cleared".
- Though you can't tell from the first instance of the asserted **CLR**, the second instance shows that the **CLR** signal is asynchronous. We know this because the clearing of the output register occurs when the **CLR** signal asserts.

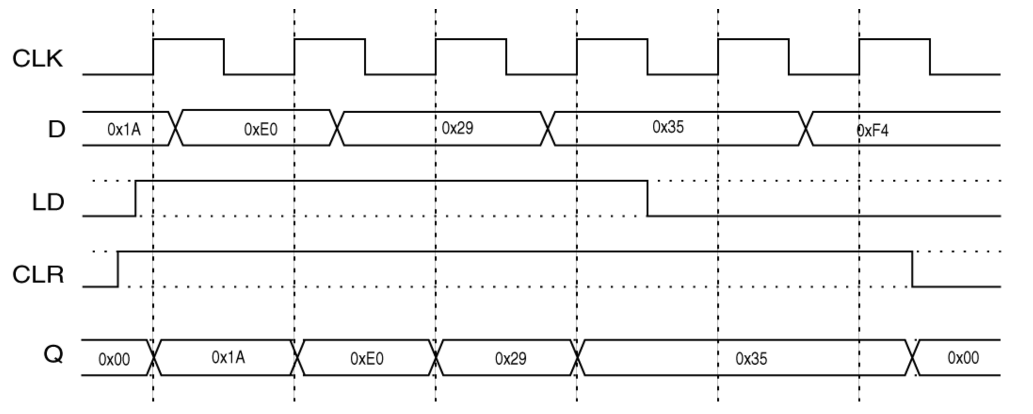


**Figure 22.7: The final solution to Example 22.3.**

### Example 22.4: Three-Value Serial Equivalency Detector

Design a circuit that detects when three consecutive values are equivalent. The circuit examines the circuit's 8-bit input value on each rising clock edge. If three consecutive values are equivalent, the circuit's **EQ** output is a '1; otherwise the **EQ** output is '0'. Also, state what controls the circuit's operation.

**Solution**: The first step in this example is to discern the inputs and outputs from the problem description and draw a top-level BBD. Figure 22.8 shows this first step in our solution.

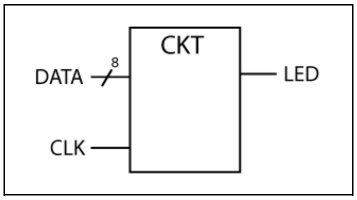


**Figure 22.8: The top-level BBD for this example.**

The next step is to create an inventory of modules the solution requires. We need to compare three 8-bit values, yet the circuit only receives one value per rising clock edge. This means that we need to store two previous values and compare them to the current value, which requires two registers. We also need to compare two different pairs of values, so the circuit requires two comparators. The circuit's **EQ** output is asserted when both comparators indicate their input values are equal, which requires an AND gate.

The key to making this circuit work is ensuring the two registers are always holding the two previous data input values. We accomplish this by connecting the data input of the first register to the incoming data; we then connect the output of this register to the input of the second register. One comparator the compares the incoming data with the previous data, while the other comparator compares the data latched on the previous clock edge to the data latched two clock cycles previously.

The next step in this solution is to assemble the modules we previously identified and connect them to make the circuit satisfy the problem description. Figure 22.9 shows the final BBD for this problem.

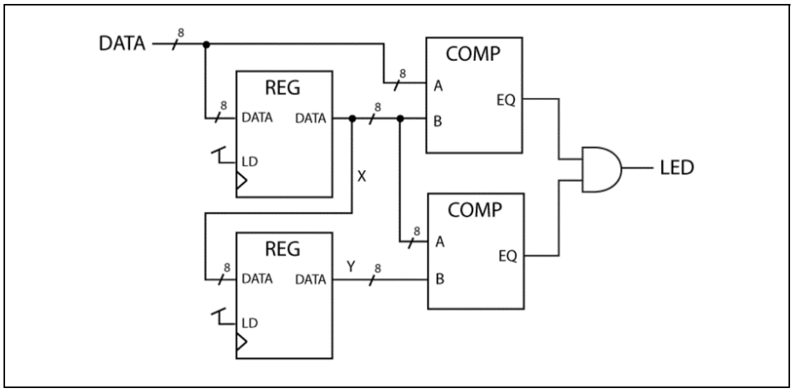


**Figure 22.9: The lower-level BBD for this example.**

The two registers are the only devices in this circuit containing control inputs. The circuit hardwires these control inputs to '1', which provides internal control for the circuit. Additionally, Figure 22.10 shows an example timing diagram for this solution.

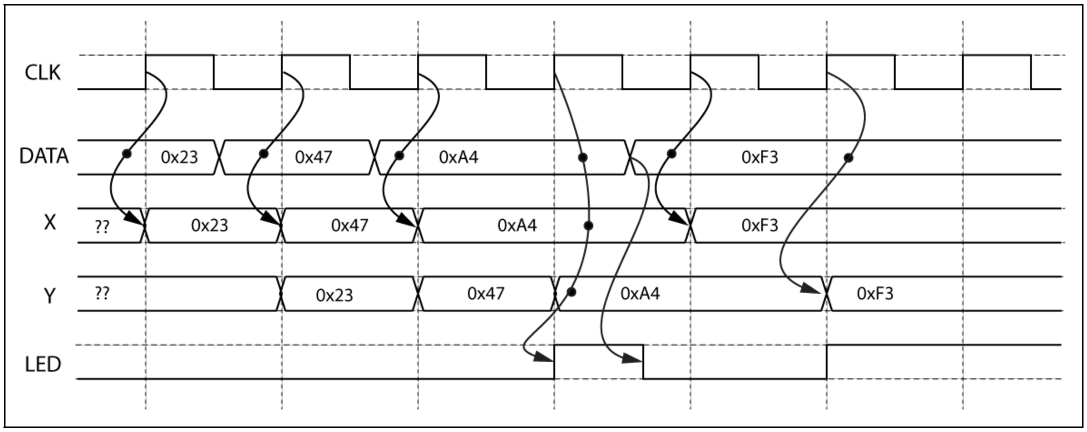


**Figure 22.10: An example timing diagram for this problem.**

## 22.3 Special Register Circuits: The Accumulator

The accumulator is a useful and common circuit in digital design. We present accumulators in this chapter because it is a relatively simple combination of a register and an RCA.

The accumulator does what its name implies: it accumulates. In digital design is that we can only add two numbers at a time, but often we need to add more than two numbers. In this case, we still can only add two numbers at time, but we add the successive values to a "running total". The resulting circuit is relatively simple: we need a device to store the running total (a register) and a device to do the adding (an RCA). Since we have flexibility in the features we add to the register, when we design an accumulator, we need to make sure of the following items:

- We need to ensure we can clear the register, as anytime we're accumulating something, we typically start accumulating with a register value of zero.
- We need to ensure the width of the register is wide enough to hold the maximum possible value based on the width of the values we're adding and the maximum quantity of values we need to add. For the sake of simplicity, the width of the accompanying RCA generally has the same data widths as the register, which requires bit-stuffing of the input RCA's data-widths.

Figure 22.11 shows a diagram of a generic accumulator. Note that some other entity needs to issues control signals to the counter (**CLR**, **LD**, & **CLK**). For this example, we're not connecting these signals, but we do in later examples that use finite state machines (FSMs). Here are some important details.

- The register is a synchronous circuit; we indicate this with the triangle on the register module; we don't route the clock line in order to keep the diagram readable.
- The register has a **CLR** control input so that we can clear the value stored in the circuit before we commence accumulating
- The circuit also has a **LD** control input, which some other entity provides
- We list the output data width as "n" bits and the input data width as "m" bits. The notion here is that we'll be adding a bunch of numbers of width "m". In doing this we need to do two things:
    1. Ensure the output data width "n" is wide enough to handle the maximum possible value of the accumulation
    2. Bit-stuff the "m" width input data to match the "n" width of the output. We do this because we expect both inputs of the RCA to have the same data width. Figure 22.11

indicates this bit-stuffing with the square containing the "+". For this diagram, we are stuffing (n-m) bits to the DATA input.

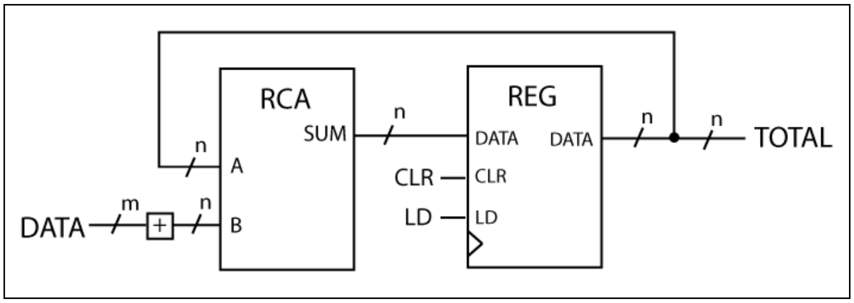


**Figure 22.11: Generic circuit for an n-bit accumulator.**

**Example 22.5: Data Width Expansion: #1**

A given circuit must accumulate eight unsigned 10-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the eight input values? Show your calculations for this problem.

**Solution**: We need to add eight 10-bit values, so we need to consider how many bits we need to represent that number. The maximum value associated with a 10-bit unsigned value is $2^{10}$-1, but in order to simplify this problem, we consider the maximum value to be $2^{10}$. We can have eight 10-bit values, so here is the final calculation:

**8 (number of inputs) * $2^{10}$ (max value on any one input)= $2^3$ * $2^{10}$ = $\underline{2^{13}}$**

Thus, in order to accurately represent the sum of the eight values, the accumulator's register requires 13 bits.

The previous problem was set up nicely in that the number of values we accumulated was an exponential factor of two. This won't always be the case. For those of you who are searching for a formula of how to calculate the width of the output based on the number of inputs, we can provide one.

**Number bits required to represent a given unsigned binary value: $\lceil \log_2(max\ value) \rceil$**

**Figure 22.12: The number of bits required to represent a decimal value.**

**Example 22.6: Data Width Expansion: #2**

A given circuit must accumulate 12 unsigned 16-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the 12 input values? Show your calculations for this problem.

**Solution**: Instead of reasoning this one out, we use the formula in Figure 22.12. Here is the calculation:

**Width of accumulator output = $\lceil \log_2[12 * (2^{16} - 1) \rceil = 20$**

**Example 22.7: Data Width Expansion: 3**

A given accumulator has a 16-bit output. What is the maximum number of unsigned 6-bit inputs values this circuit can accurately represent? Show your calculations for this problem.

**Solution**: This is a similar problem but we take a different approach to the solution and use basic algebraic manipulation (and other magic) to arrive at the solution. We solve the following equation for VALUE.

$$\mathbf{16} = \lceil \mathbf{log_2}[\boldsymbol{VALUE} * (\mathbf{2^6} - \mathbf{1})] \rceil \textbf{ ; remove the ceiling function}$$

$$\mathbf{16} = \mathbf{log_2}[\boldsymbol{VALUE} * (\mathbf{2^6})]$$

$$\mathbf{2^{16}} = (\boldsymbol{VALUE} * \mathbf{2^6})$$

$$\boldsymbol{VALUE} = \frac{\mathbf{2^{16}}}{\mathbf{2^6}} = \mathbf{2^{10}} = \mathbf{1024}$$

## 22.4 Registers: The Final Comments

The use of registers is quite common in digital design; this chapter presented only a basic register. Other popular flavors of registers include shift registers and counter, which are the main topics in upcoming chapters. The Venn diagram in Figure 22.13 shows how the various flavors of registers relate to each other.

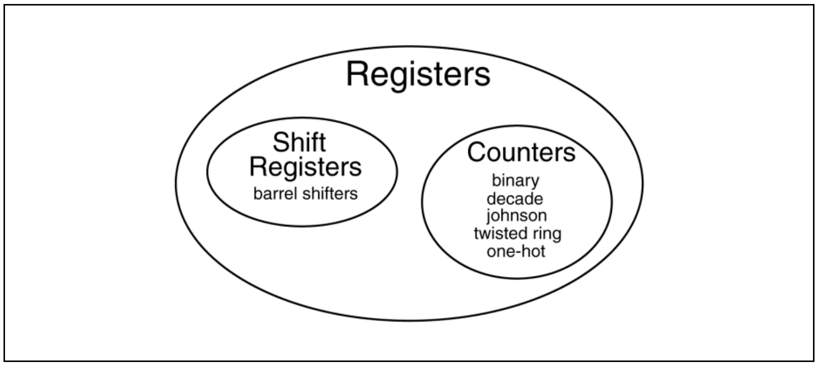


**Figure 22.13: Venn diagram for the register family.**

## 22.5 Digital Design Foundation Notation: Registers

The register is a controlled circuit and is one of our Digital Design Foundation modules. Figure 22.14 shows the appropriate digital design foundation notation for the register with a basic set of control features. Registers typically have both data inputs and data outputs. The typical set of controls for a register includes synchronous load signals (LD) and an asynchronous clear input. Table 22.1 show a complete description of the registers input and output signals.

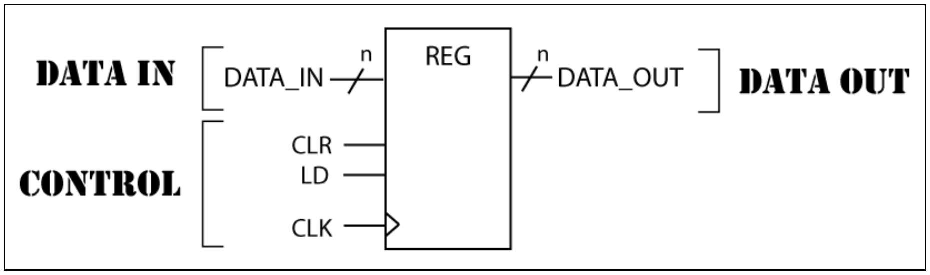


**Figure 22.14: Typical data and control signals for a register.**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **DATA_IN** | The data that can be latched into the register’s storage elements.. |
| OUTPUT DATA | **DATA_OUT** | The DATA_OUT signal is the data currently being stored in the counter’s storage elements. |
| CONTROL | **CLK** | Registers are synchronous circuits, in that the loading of data to the register happens on the clock edge. |
| | **LD** | Allows the latching (loading) of the DATA_IN signal to the counters storage elements. This signal is always synchronous. |
| | **CLR** | Latches 0’s into each of the register’s storage elements; can be synchronous or asynchronous. |
| STATUS | **n/a** | - |

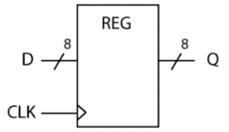
**Table 22.1: The foundation description for a simple register.**

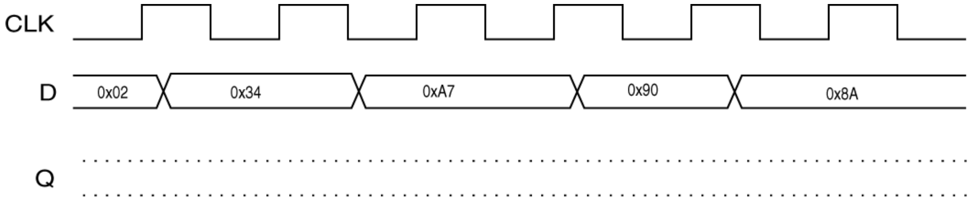
## 22.6 Chapter Summary

- A register is a sequential circuit that we can model as a parallel combination of single-bit storage elements. We model these storage elements as a specific number of D flip-flops that share a common clock signal and possibly other control signals typically associated with D flip-flops (such as clear signals). We typically the register to “latch” (and thus remember) an n-bit wide set of data on the active clock edge of the device.
- When we refer to the state of the register, we are referring to the data currently stored in the register’s underlying memory elements.
- We consider register inputs such as **CLR** (clear), **CLK**, and **LD** (load) to be control signals, in that they control the operation of the register.
- A common use for registers is in accumulators. An accumulator is a combination of a register and an RCA configured in such a way as to add a list of numbers. Digital circuits can only add two values at a time, so we use an accumulator to add lists of number, which it effectively does by keeping a running total of the numbers being summed. The key to obtaining the correct answer with an accumulator is to make sure you clear the underlying register before the summing operations commence.

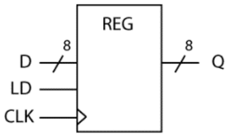
## 22.7 Chapter Exercises

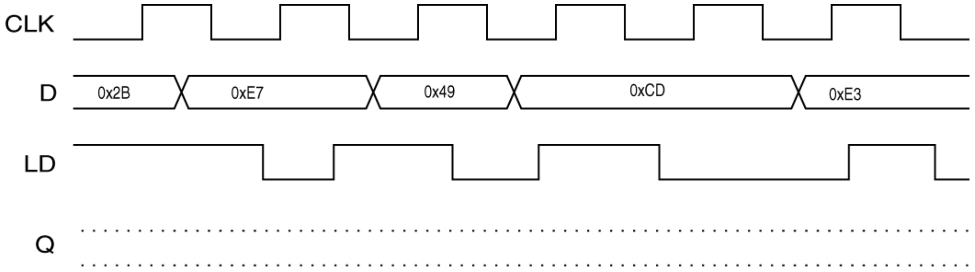
**1)** Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



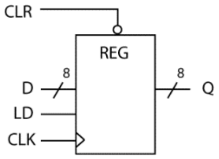


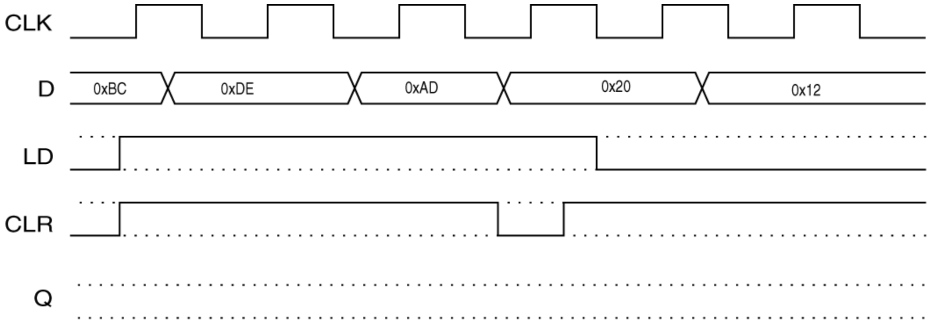
**2)** Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



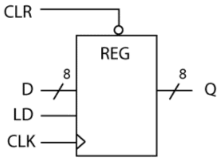


**3)** Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. The CLR input is an asynchronous input that clears the register when asserted and has a higher precedence than the LD input. Consider the register to be rising-edge triggered and ignore all propagation delay issues.





**4)** 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 16-bit device.



**5)** Briefly explain why a register is a major component of an accumulator.

**6)** We often refer to accumulators that only have RCAs (and no registers) as a "random number generator". Briefly explain why this is the case.

**7)** The registers associated with accumulators always have the ability to either clear the memory in the register, or load the register with zero. Briefly explain why the registers in accumulators require clear control signals.

**8)** A given circuit must accumulate four unsigned 20-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the given number of input values? Show your calculations for this problem.

**9)** A given circuit must accumulate 16 unsigned 18-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the given number of input values? Show your calculations for this problem.

**10)** A given circuit must accumulate 13 unsigned 11-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the given number of input values? Show your calculations for this problem.

**11)** A given circuit must accumulate 17 unsigned 7-bit values. What is the minimum width of the accumulator output such that the accumulator can accurately represent the sum of the given number of input values? Show your calculations for this problem.

**12)** A given accumulator has a 13-bit output. What is the maximum number of unsigned 5-bit unsigned binary input values this circuit can accurately represent? Show your calculations for this problem.

**13)** A given accumulator has a 20-bit output. What is the maximum number of unsigned 8-bit unsigned binary input values this circuit can accurately represent? Show your calculations for this problem.

**14)** A given accumulator has a 39-bit output. What is the maximum number of unsigned 16-bit unsigned binary input values this circuit can accurately represent? Show your calculations for this problem.

## 22.8 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the use of hardware when problem require extra hardware
- Assume all inputs and outputs are positive logic unless stated otherwise
- State all forms of control for your solution.

**1)** Design a circuit that stores an 8-bit value when the circuit's three single-bit inputs (A, B, and C) are asserted. The circuit has an 8-bit output that shows the current value that is stored in the circuit. The loading of input data is synchronized with a rising clock edge. Assume the clock is periodic.

**2)** Design a circuit that stores one of two 8-bit inputs X & Y. The circuit loads the value when one and only one of the A & B inputs (single-bit) are asserted. The circuit loads the X value under those conditions if the SEL input is asserted; otherwise, it loads the Y input. Assume the SEL input is positive logic. The loading of input data is synchronized with a rising clock edge. Assume the clock is periodic.

**3)** Design a circuit that stores an 8-bit input when only one of three single-bit signal A, B, & C is asserted. Assume the three inputs are negative logic. The circuit also clears asynchronously when two single-bit inputs T & U are both asserted. Assume the T input is positive logic while the U input is negative logic. The loading of input data is synchronized with a rising clock edge. Assume the clock is periodic.

# 23 Finite State Machines

## 23.1 Introduction

If you look-up the definition of a finite state machine (FSM) on Wikipedia, you'll find a description that is based in abstract mathematics. While this is all good and fine, I've never understood those types of definitions when dealing with digital devices. All the digital devices I know about are simple, intuitive, and straightforward to understand. A FSM is a relatively simple, but incredibly useful circuit.

This chapter presents FSMs using low-level circuity and in an intuitive manner so that you can build a strong understanding of their operation. The primary use of FSMs in digital design is as a digital circuit that controls another digital circuit; it's a controller circuit. We present FSMs as true controller circuits in a later chapter. An FSM is a combination of other circuits that we already looked at: registers and decoders.

**Main Chapter Topics**

**FSM CONSTRUCTION**: We describe the basic digital modules that form a FSMs

**FSM OPERATION**: We describe how the basic construction of FSMs determines the characteristics of how they operate.

**FSM MODELING USING STATE DIAGRAMS**: We can describe the operation of FSM using state diagrams. This description includes representing state transitions, input representations, and both Moore and Mealy-type outputs.

**FSM ILLEGAL STATE RECOVERY:** This chapter describes the notion of hang states and provides techniques on how to avoid this unwanted behavior in FSM.

**Chapter Acquired Skills**

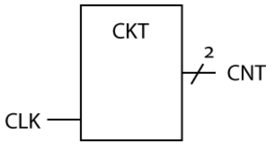
- Describe the basic subsystems of a FSM
- Describe the attributes of Moore and Mealy outputs on an FSM
- Design FSM-based counters at a low level using the basic subsystems of a FSMs
- Design FSMs with built-in illegal state recovery
- Describe the four basic parts of a state diagram

## 23.2 FSM Design: Start with What You Know

There are many different approaches to understanding FSMs; the approach we take is to develop simple FSMs from what we already know and then build up our knowledge using more feature-laden examples. This chapter provides everything you need to know about FSMs using simple examples that leverage hardware and topics we previously discussed. This approach gives you a solid foundation for understanding FSMs, and helps you realize that FSMs are relatively simple devices. FSMs are most interesting when we use them as controller circuits, but this chapter develops non-controller FSMs; we move to controller-type FSMs in a later chapter.

**Example 23.1: FSM Design #1: 2-Bit Up Counter**

Using the diagram on the right, design a FSM that implements a 2-bit binary counter. The counter's increments are synchronized with the rising-edge of the CLK input. This counter counts endlessly using this sequence in binary: …0,1,2,3,0,1,… Provide a circuit diagram and a state diagram.



**Solution**: A counter is a device that counts in a repeating sequence. There are many types of counters; this problem deals with a 2-bit binary up counter. The counter is binary because all digital circuits are inherently binary. The counter is a 2-bit counter because that is the minimum number of bits we need to represent the given sequence in binary. We refer to this as an "up counter" because the sequence is always counting up by one, which is another way of saying the counter always increments on each rising edge of the clock.

A counter is a simple register with some added external circuitry that makes it into a counter. This circuit requires memory elements because the output is dependent on the past inputs. There is only one input, the **CLK** signal, but the outputs change every clock cycle (synchronized with every rising-edge) as the counter steps through the count sequence.

Since this is a FSM, we can model it using a state diagram. Figure 23.1 shows a state diagram that describes the FSM we're creating to solve this problem. This state diagram completely describes the functionality of the given FSM; here is a list of the important features of the state diagram:

- The state diagram has four states: one state for each unique value in the given count sequence
- We could represent the count with two or more bits, but it makes the most sense to represent the counter with using two bits; we can arrange two bits in four unique combinations.
- Each state bubble includes both a descriptive label (the top part of the bubble) and a value for the output (the body of the bubble). The labels are descriptive, which makes the state diagram self-commenting, which in turn makes the diagram more understandable to humans.
- The state diagram has singly directed arrows indicating state transitions. The "-" character associated with each arrow indicates the given state transition is unconditional. In state diagrams, all state transitions must explicitly list the conditions under which the transfer occurs, or list a "-" if the transition is unconditional.
- Although the circuit has a **CLK** input, there is no mention of the **CLK** signal in the state diagram. This implies the state transitions in the diagram are synchronized to the active clock edge of the circuit.

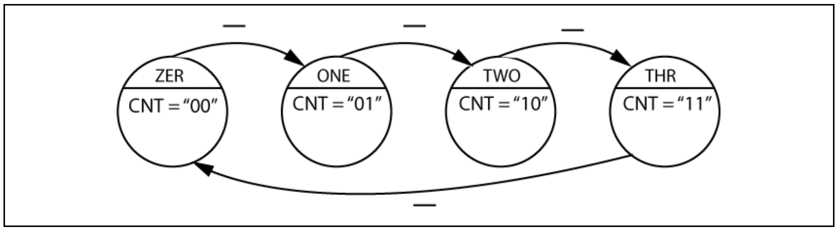


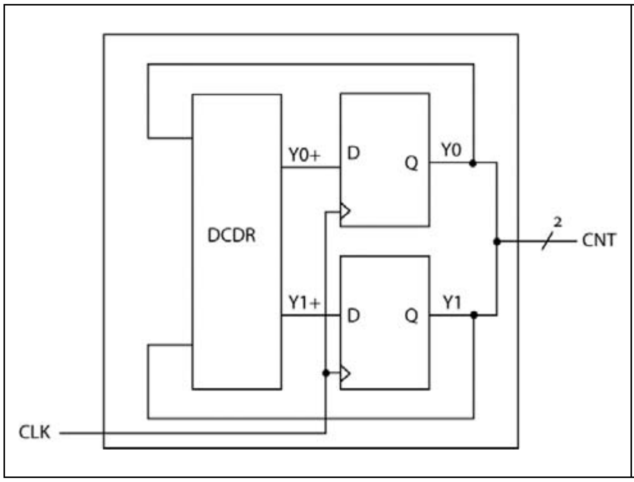
**Figure 23.1: The state diagram to support our solution.**

The circuit for this problem requires two rising-edge triggered (RET) D flip-flops. These D flip-flops are memory elements, which we use to represent the "state" of the circuit, which means the D flip-flops hold the 2-bit binary count that the problem is asking for.

We need to make the D flip-flops sequence through the 2-bit binary count. That means, for example, when the state of the circuit (thus the output) is "10", we want the next state to be "11". We accomplish this by placing some circuitry in front of the D flip-flop inputs. Since we don't exactly know what this circuitry is, we hedge our bets and use a decoder. Figure 23.2(a) shows the circuit we've described up to this point.

We now need to define the transitions we need to happen in order to make the count sequence appear on the circuit's outputs. The best way to do this is to define the circuit transitions in tabular format; we need to show the present state of the circuit and the circuit's desired next state. The accepted vernacular for this is to create what we refer to as a PS/NS table, where PS and NS stand for *present state* and *next state*, respectively. The present state of the circuit is the value currently on the outputs of the circuit (which is the same value as stored in the circuit); the next state of the circuit are the values on the **D** inputs to the circuit's two D flip-flops. Figure 23.2(a) uses the term **Y0** & **Y1** for the present state (PS) of the circuit and **Y0**$^+$ & **Y1**$^+$ for the next state (NS) of the circuit.

We now need to define the PS/NS table. Once we describe the required transitions from present state to next state (PS→NS) in tabular format, we have ourselves a generic decoder that we can use to complete the circuit. Figure 23.2(b) shows the PS/NS table we're looking for, as this table lists the present states of interest and their corresponding next states. Note that this PS/NS table does not include any mention of the **CLK** signal because we assume all state transitions in the table are synchronized to the rising-edge of the **CLK** signal.



| PS<br>Y1 Y0 | NS<br>$Y1^+$ $Y0^+$ |
|---|---|
| 0 0 | 0 1 |
| 0 1 | 1 0 |
| 1 0 | 1 1 |
| 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.2: A block diagram for our solution (a), and the associated PS/NS table (b).**

Figure 23.3 shows a timing diagram for this circuit; this diagram lists the outputs in both an aggregate form (CNT) as well as its constituent parts (**Y1** & **Y2**). The **CNT** output arbitrarily shows the output count in decimal format. Note in this diagram that the rising-edge of the clock synchronizes changes in the **CNT** output because of using RET D flip-flops for the storage elements, which we refer to as *state registers*. Wildly interesting.

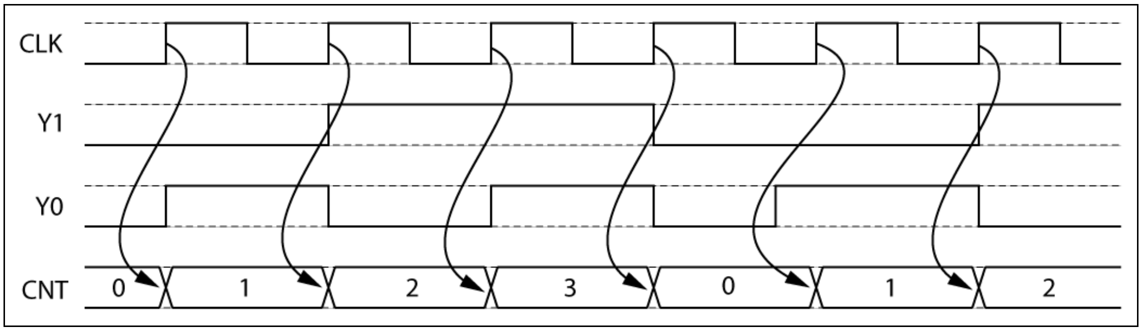
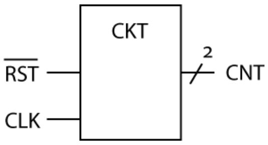


**Figure 23.3: A timing diagram showing an example of the circuit's operation.**

**Example 23.2: FSM Design #2: 2-Bit Up Counter with Asynchronous Reset**

Using the diagram on the right, design a FSM that implements a 2-bit binary counter. The counter's increments are synchronized with the rising-edge of the **CLK** input. This counter counts endlessly using this sequence: …0,1,2,3,0,1,… The counter has an active-low asynchronous **RST** input that resets (makes the count zero) the counter when asserted. Assume the **RST** input takes precedence over the normal count operations. Provide both a circuit diagram and state diagram for your solution.



**Solution**: This problem is similar to the previous problem, but now has an added reset feature. The **RST** signal is an active-low signal that causes the FSM to immediately transition to the "00" state when the **RST** signal is asserted. The state transition associated with the asserted **RST** signal is asynchronous, which means the state transition happens whenever the signal is asserted and is not synchronized with the active clock edge.

Figure 23.4 shows that we do not model the **RST**-based transition as a state-to-state transition; we model it as a "nowhere-to-state" transition. Also note in Figure 23.4 that the signal **RST** signal includes an overbar to indicated that the signal is an active low.

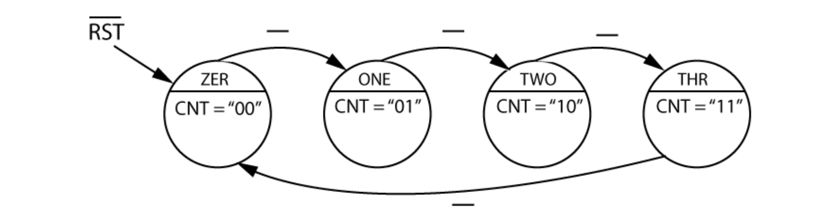
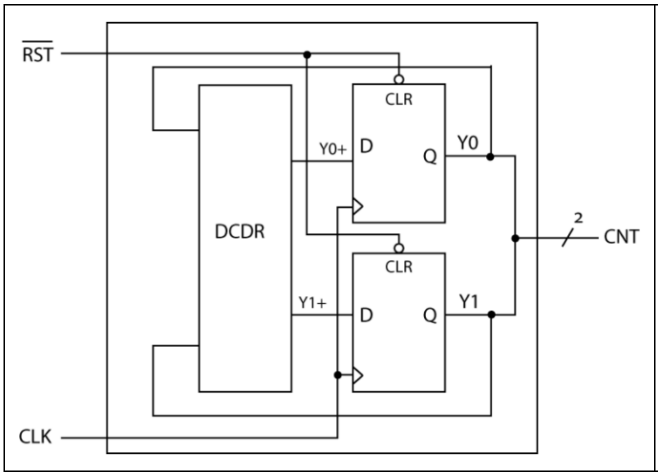


**Figure 23.4: The state diagram to support our solution.**

Our next concern is to draw a lower-level BBD for our solution. We opt to replace the simple D flip-flops in the previous example with D flip-flops that include asynchronous active-low clear inputs. Figure 23.5(a) shows our new solution. This solution is similar to the previous problem's solution, the only difference being that we include a **RST** input that connects to the **CLR** inputs of each D flip-flop. From Figure 23.5(a), we know that the D flip-flop's **CLR** inputs are active low (as indicated with the bubble on the input), but for a complete solution, we must somewhere state that the D flip-flop's **CLR** inputs are asynchronous.

Figure 23.5(b) shows the PS/NS table for the solution. Note that the table does not include the **RST** input or the **CLK** input. We assume state-to-state transitions are synchronized with the **CLK** edge, so we omit it from the PS/NS table. We omit the **RST** signal from the PS/NS table for two reasons. First, because asynchronicity is hard to indicate in PS/NS tables because the other signals in the table are synchronous. Second, because we already directly handled the **RST** input via the **CLR** control input to the D flip-flops on the underlying hardware (see Figure 23.5(a)).



(a)

| PS | NS |
|---|---|
| Y1 Y0 | Y1$^+$ Y0$^+$ |
| 0 0 | 0 1 |
| 0 1 | 1 0 |
| 1 0 | 1 1 |
| 1 1 | 0 0 |

(b)

**Figure 23.5: A block diagram for our solution (a), and the associated PS/NS table (b).**

Figure 23.6 shows a timing diagram for this circuit; we arbitrarily list the **CNT** output in decimal. Here are a few important things to note about Figure 23.6.

- We list the **RST** signal with an overbar, which indicates it's an active-low signal.
- **RST** signal is initially asserted when puts the FSM into the ZER state. We do this in so that we don't need to provide an *initial state* of the circuit.
- The **CNT** value represents the current content of the state registers (D flip-flops); we arbitrarily list the count in decimal.
- The second and third time the **RST** signal is asserted causes an immediate transition the ZER state (see Figure 23.4), which in turn causes the **CNT** to be "00".
- The third assertion of the **RST** signal overlaps an active clock edge. Asynchronous inputs have precedence over synchronous inputs in sequential circuits. You can design D flip-flops either way, or read the spec sheet if you're using an off-the-shelf flip-flop.

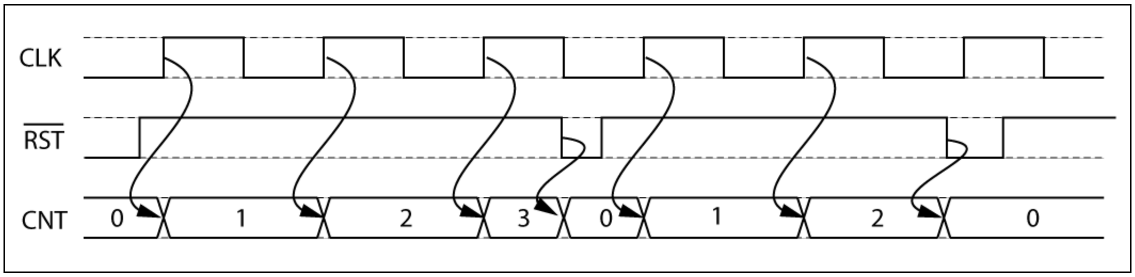
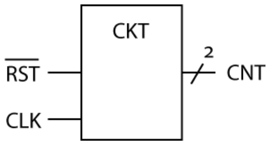


**Figure 23.6: A timing diagram showing an example of the circuit's operation.**

**Example 23.3: FSM Design #3: 2-Bit Up Counter with Synchronous Reset**

Using the diagram on the right, design a FSM that implements a synchronous 2-bit binary up counter. The counter has an active-low synchronous **RST** input that resets the counter when asserted. The **RST** input takes precedence over the counter's basic count operation. Consider the rising edge to be the **CLK** signal's active clock edge. Use D flip-flops with only **D** and **CLK** inputs for your FSM's storage elements. Provide both a circuit diagram and state diagram for your solution.



**Solution**: This problem is similar to the previous problem, but the synchronous reset signal changes how we approach the problem. Because the reset signal is synchronous, it only takes effect on the circuit's active clock edge. Additionally, the problem needed to state whether the count operation or the reset operation has precedence when the **RST** signal is asserted on the **CLK's** active edge. Finally, the problem states that we must use only plain D flip-flops for the state registers, which means we can't rely on a D flip-flop's **CLR** input to complete this problem. The first place to start with this problem is the by drawing a state diagram, which we show in Figure 23.7. Here are some important features from the state diagram.

- Most of the signal transitions are no longer unconditional; the state-to-state transitions only occur when the **RST** signal is unasserted, which is when **RST**='1'.
- The ZER state is the only state that contains a self-loop; in all other cases, either the FSM returns to the ZER state or transitions to the next state in the count sequence.
- The transition from the THR state to the ZER state is unconditional because the FSM always transition from the THR state to the ZER state on the next clock edge.
- Each state has two condition arrows leaving the state: one transition for each value of the **RST** signal. The "don't care" effectively implements the Boolean equation: (**RST** +**!RST**), which says the expression is always true, thus the transition happens unconditionally.

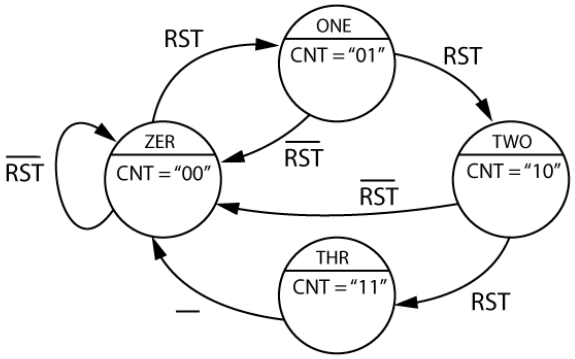
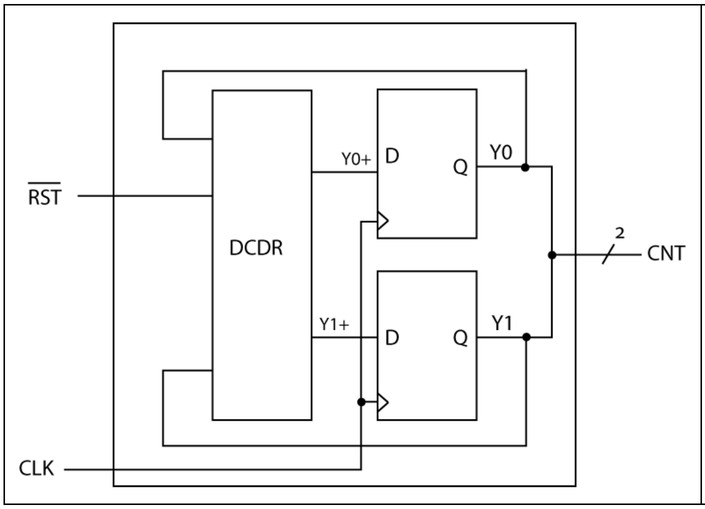


**Figure 23.7: The state diagram to support our solution.**

Figure 23.8(a) shows the underlying circuit schematic for the solution. The **RST** signal is now an input to the decoder. We must explicitly state that the **RST** signal takes precedence over the circuit's active clock edge. Figure 23.8(b) shows that the PS/NS table for this solution is also significantly different from the previous solution. Because the **RST** is synchronous, we can represent the signal in the table. Here are some other notable items regarding the PS/NS table:

- We're using an exclamation mark ("bang" notation) to indicate that RST is negative logic.
- When the RST signal is low (active), the next state is always the "00" state; the first four rows of the table show this characteristic.

- The bottom four rows of the PS/NS table show the normal binary count sequence where the count increments on each active clock edge.
- The fact that transitions indicated in the fourth and eighth rows of the table are always from the "11" state to "00" state essential make that transition unconditional, which means the transition does not depend on the **RST** signal. We indicate this condition what the "-" associated with the THR to ZER transition in the state diagram.



(a)

| PS | NS |
|---|---|
| **!RST Y1 Y0** | **$Y1^+$ $Y0^+$** |
| 0 0 0 | 0 0 |
| 0 0 1 | 0 0 |
| 0 1 0 | 0 0 |
| 0 1 1 | 0 0 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

(b)

**Figure 23.8: A block diagram for our solution (a), and the associated PS/NS table (b).**

Figure 23.9 shows an example timing diagram associated with the FSM. Here is the fun stuff to note in the timing diagram.

- The **CNT** arbitrarily starts at "2"; representing the **CNT** in decimal is also arbitrary.
- The **RST** signal is asserted at the start of the timing diagram. Because the signal is synchronous, it does not take effect until the first active clock edge.
- The third time the **RST** signal is asserted is between active clock edges, and thus has no effect on the state of the FSM.

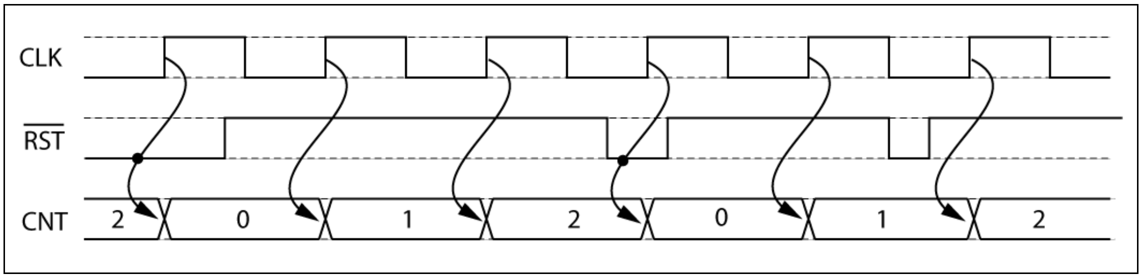
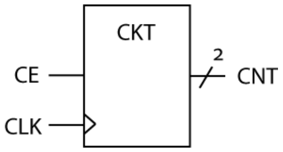


**Figure 23.9: A timing diagram showing an example of the circuit's operation.**

**Example 23.4: FSM Design #4: 2-Bit Up Counter with Counter Enable**

Using the diagram on the right, design a FSM that implements a synchronous (RET) 2-bit binary up counter. This counter counts up when the CE (count enable) input is a '1' and holds the previous count when the CE signal is '0'. Provide both a circuit diagram and state diagram for your solution.



**Solution**: The best place to start is defining the state diagram. This is a standard 2-bit binary up counter with the added feature of a control input that allows the counter to progress through its count sequence (**CE**). The problem description states that the counter is synchronous, which means changes in the count output are synchronized with the active clock edge. The black box diagram from the problem description indicates that the device is rising-edge triggered. Figure 23.10 shows the state diagram for our solution. Here are a few important things to note about the state diagram.

- The state diagram indicates that the **CE** input controls whether the FSM transitions to different state or stays in the current state. The **CE** with an overbar indicates the **CE** signal is not asserted, and thus the counter output does not change. The FSM accomplishes the "no change" in the counter output by not changing state, as the state diagram indicates with the "self-loop".
- Three items determine the state transitions: 1) the **CE** input, 2) the current state, and 3) the **CLK** signal.
- The FSM has one control input: **CE**. That means we must account for two arrows leaving each state: one for both the **CE** asserted and **CE** not asserted. If we did not list both options, the state diagram would not be completely specified and thus be incorrect.

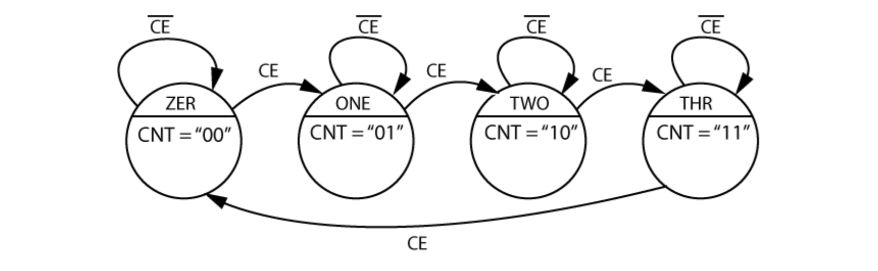
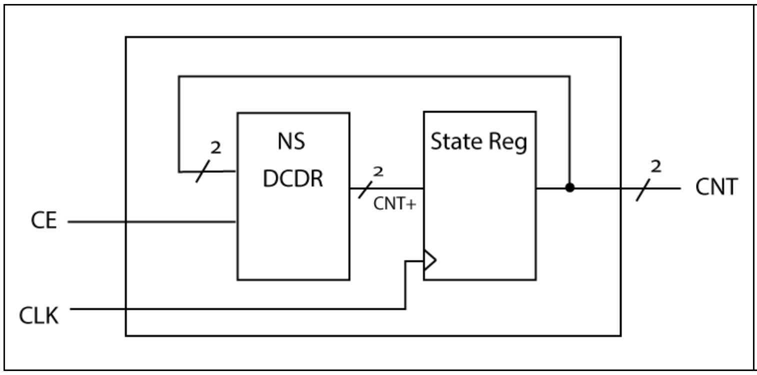


**Figure 23.10: The state diagram to support our solution.**

The next step is to layout the underlying hardware. Figure 23.11(a) shows a set of hardware that does the job for us. Here are a few things to note about this hardware.

- We essentially abstracted the hardware to a higher level by replacing the two flip-flops from a previous example with a "state reg" model. This module is officially the *state registers* for the FSM, and is responsible for holding the state of the FSM. This module is a simple a 2-bit register that does not contain a **LD** input.
- The next state decoder (NS DCDR) explicitly shows that the next state of the FSM is a function of the **CE** input and the current state. The decoders two output bits (**CNT**$^+$) provide the data inputs to the 2-bit state registers. The **CE** input controls whether the FSM does a self-loop (**CE** = '0') or transitions to the next state (**CE** = '1'). The current state determines which state the FSM transitions to when **CE** is asserted.

At this point, we described everything there is to know about this FSM. But wait, there's more. Figure 23.11(b) shows the associated PS/NS table for the FSM. The most important thing to note from Figure 23.11(b) is that the present state (PS) is the output of the state registers, while the next state (NS) is the output of the next state decoder (NS DCDR).



| PS<br>CE CNT | NS<br>$CNT^+$ |
|---|---|
| 0 0 0 | 0 0 |
| 0 0 1 | 0 1 |
| 0 1 0 | 1 0 |
| 0 1 1 | 1 1 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.11: A block diagram for our solution (a), and the associated PS/NS table (b).**

Figure 23.12 shows an example timing diagram for the FSM. Things to note about this timing diagram include the following fun information.

- We need to provide a starting state, which is the "2" on the left side of the **CNT** signal. Unlike a previous example that had a reset input, this FSM has no way to force the FSM to a given state.
- When the **CE** input is not asserted (**CE** = '0'), the output transitions back to the current state. We opt to leave the signal changing symbology on the first clock edge, but the signal does not change (it remains at '2'). This symbology saves time when drawing the **CNT** signal.
- When the **CE** signal is asserted between the third and fourth rising clock edge, the circuit ignores the **CE** signal. The only time the FSM considers the **CE** signal is on the active edge of the **CLK** signal.

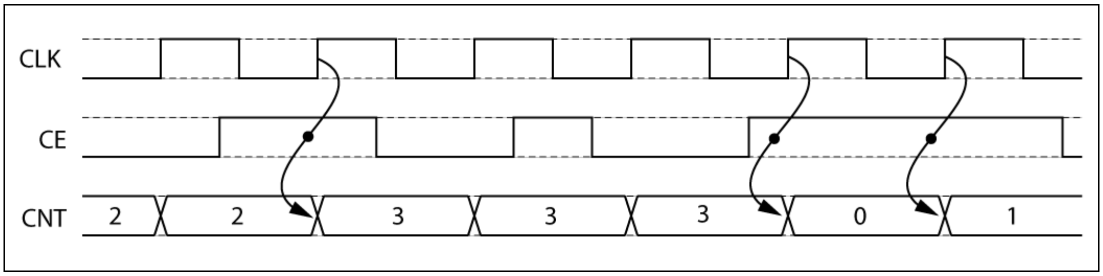
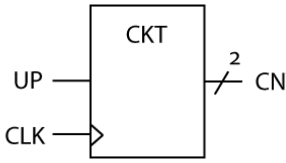


**Figure 23.12: The solution for this example.**

**Example 23.5: FSM Design #5: 2-Bit Up/Down Counter**

Using the diagram on the right, design a FSM that implements a 2-bit synchronous binary up/down counter. This counter counts up when the **UP** input is an asserted, and down when the **UP** input is not asserted (both on the rising clock edge). Assume the **UP** input is positive logic. Provide both a circuit diagram and state diagram for your solution.



**Solution**: This is yet another version of the 2-bit counter. This counter can count up or down based on the UP input. Because the **UP** input controls the count direction of the counter, we consider it a control input the module. Although the problem uses the one control signal (**UP**) to control the direction of the count, we could have used two signals for the same effect, specifically, we could have both "UP" and "DOWN" inputs to the circuit. Figure 23.13 shows the state diagram describing a solution this problem. Here are the important things to note from the state diagram.

- We consider the counter to be "circular" because it automatically rolls over and under. The counter rolls over from 3→0 when counting up and from 0→3 when counting down.
- The FSM has one control input: **UP**, which means that we must account for two arrows leaving each state: one for both the **UP** asserted and not asserted cases. Each state bubble has two transition arrows exiting it.

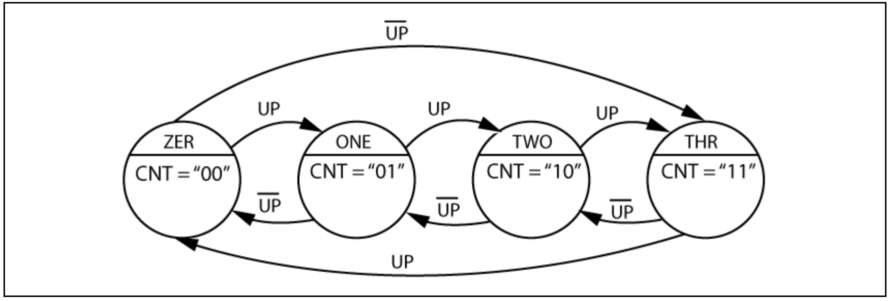
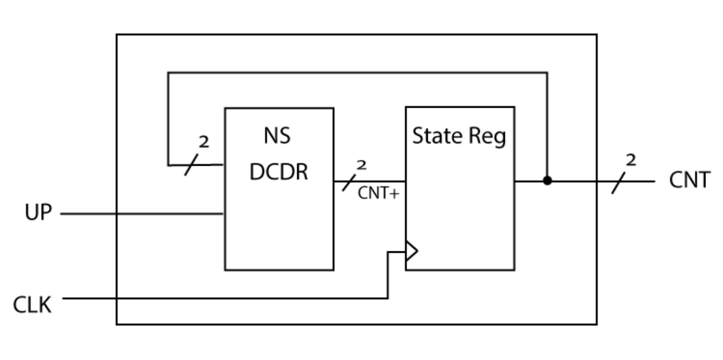


**Figure 23.13: The state diagram to support our solution.**

Figure 23.14(a) shows the underlying block diagram for the solution. In this diagram, that we use the output of the state registers as the desired count output (**CNT**). Figure 23.14(b) shows the PS/NS table. The PS/NS table lists the operation of the FSM in tabular format. All the information in Figure 23.14(b) is the same information found in the associated state diagram, but in a different format.



| **PS**<br>**UP CNT** | **NS**<br>**$CNT^+$** |
|---|---|
| 0 0 0 | 1 1 |
| 0 0 1 | 0 0 |
| 0 1 0 | 0 1 |
| 0 1 1 | 1 0 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.14: A block diagram for our solution (a), and the supporting PS/NS table.**

Figure 23.15 shows an example timing diagram for the FSM of this example. A few worthy things to note about this diagram include the following:

- We arbitrarily started the **CNT** signal at '2'. This represents the state of the circuit; we had to provide this information explicitly as the circuit has no other control inputs to put the FSM into a known state.
- All state changes (changes in the **CNT** signal) are synchronized to the rising edge of the **CLK** signal.

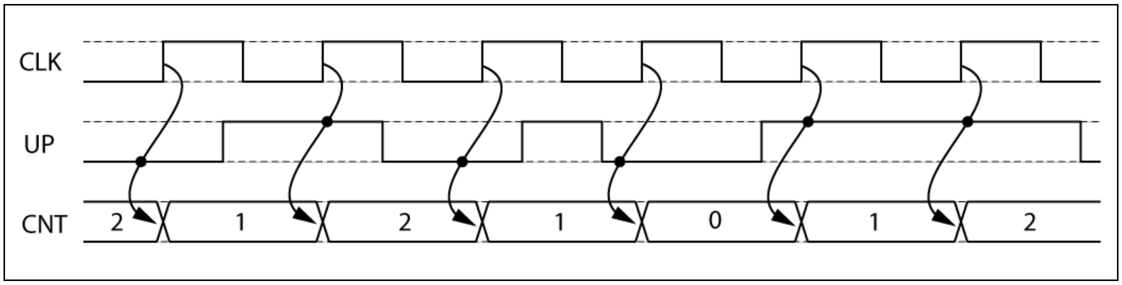
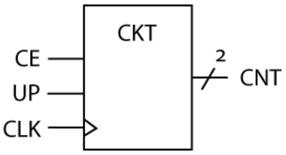


**Figure 23.15: The solution for Example 4.8.**

**Example 23.6: FSM Design #6: 2-Bit Up/Down Counter with Count Enable**

Using the diagram on the right, design a FSM that implements a synchronous 2-bit binary up/down counter. This counter counts up when the **UP** input is a '1', and counts down when the **UP** input is '0'. The counter also has a **CE** input (control enable), which allows the counter to "count" when asserted. Assume the **CE** input is positive logic. Provide both a circuit diagram and state diagram for your solution.



**Solution**: From examining the BBD for the problem, we see that changes in the circuit's output are synchronized with the rising edge of the clock. This problem also differs from previous problems in that there are now two control inputs (**CE** and **UP**), which means there are four conditions that determine the next state of the FSM: **CE**, **UP**, and the 2-bit value for the present state of the FSM (which is also the **CNT** signal for this problem). Figure 23.16 shows the state diagram associated with this solution. Some other things to note include the following.

- When the **CE** is not asserted the FSM does not change state. We indicate this condition is indicated with **!CE** (the complemented, or unasserted **CE** signal) associated with the self-loop.
- Because there are two signals associated with the state transitions (not including the PS), there should be four arrows leaving each state. The state diagram only shows three arrows leaving each state, which is because we opted to not include the **UP** signal in the self-loops. We can do this because if **CE** is not asserted, the **UP** input does not matter (it's a "don't care"); omitting it from the state diagram does not alter the function of the state diagram and make the diagram more readable. In this FSM, !**CE** is equivalent to: !**CE·UP** + **!CE·!UP**, which factors to: !**CE**.
- The conditions associated with each state-to-state transition are now more than a signal; they are now a logic expression. The dot means AND, so both conditions need to be true in order for the transition to occur. For example, to transition from state ONE to state TWO, both the **CE** and **UP** signal must be asserted. Similarly, to transition from state THR to state TWO, both **CE** signal needs to be asserted and the **UP** signal needs to be unasserted.

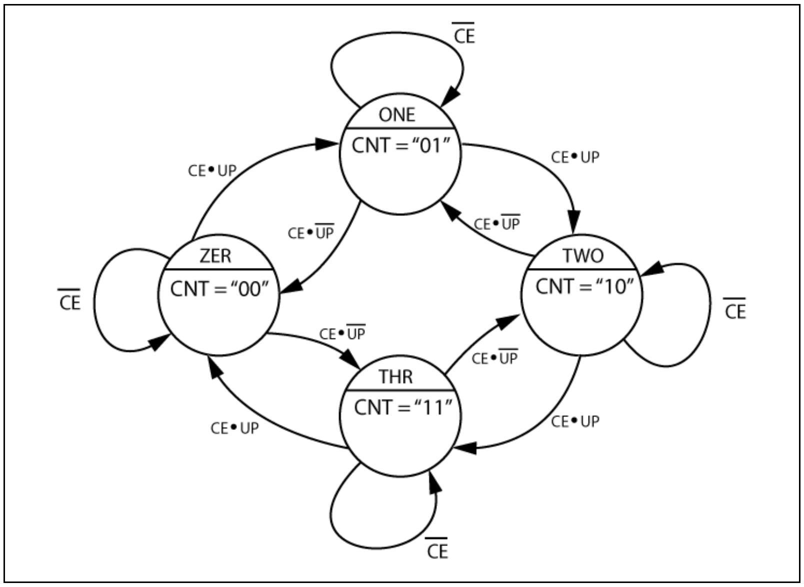
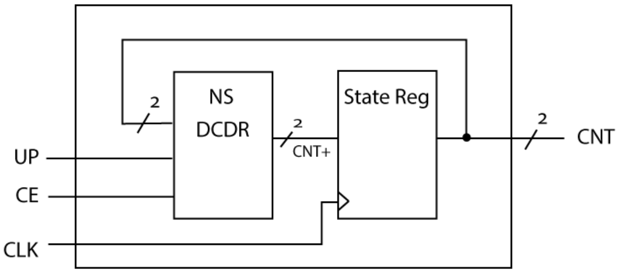


**Figure 23.16: The state diagram to support our solution.**

Figure 23.17(a) shows the schematic associated with this solution while Figure 23.17(b) shows the associated PS/NS table. The PS/NS table has four inputs: **CE**, **UP**, and the 2-bit **CNT** signal (the present state). The next state is dependent upon each of these four signals, which is why the PS/NS table is a 4-input table. The PS/NS table has four inputs and thus 16 rows. The NS column of the PS/NS table shows the inputs to the state registers, which become the next state on the next rising clock edge.



| PS<br>CE UP CNT | NS<br>CNT+ |
|---|---|
| 0 0 0 0 | 0 0 |
| 0 0 0 1 | 0 1 |
| 0 0 1 0 | 1 0 |
| 0 0 1 1 | 1 1 |
| 0 1 0 0 | 0 0 |
| 0 1 0 1 | 0 1 |
| 0 1 1 0 | 1 0 |
| 0 1 1 1 | 1 1 |
| 1 0 0 0 | 1 1 |
| 1 0 0 1 | 0 0 |
| 1 0 1 0 | 0 1 |
| 1 0 1 1 | 1 0 |
| 1 1 0 0 | 0 1 |
| 1 1 0 1 | 1 0 |
| 1 1 1 0 | 1 1 |
| 1 1 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.17: A block diagram for our solution (a), and the associated PS/NS table.**

Figure 23.18 show and example timing diagram for the circuit. All state transitions only occur on the active clock edges when **CE** is asserted.

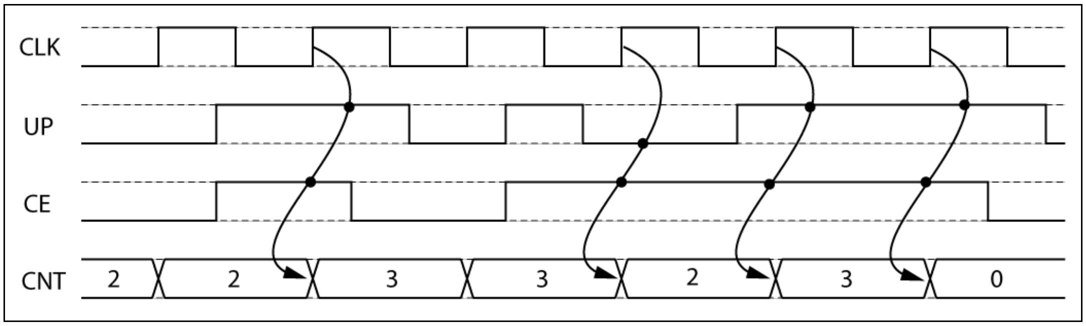
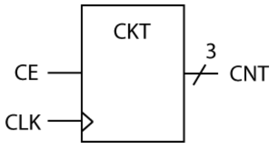


**Figure 23.18: The solution for Example 4.8.**

**Example 23.7: FSM Design #7: 3-Bit Stoneage Unary Up Counter**

Using the diagram on the right, design a FSM that implements a synchronous 3-bit stoneage unary up counter. The counter also has a control enable (**CE**) input that allows the counter to count. Assume **CE** is a positive logic signal. Do not use more than two bits of storage in your FSM design. Provide both a circuit diagram and state diagram for your solution.



**Solution**: This problem seems similar to previous examples, but the problem constrains the memory usage to two bits. In the context of FSMs, the memory is associated with the state registers. In addition, it’s still a synchronous binary up counter, but the count is now something special: the infamous stoneage unary count. Recall that a 3-bit stoneage unary count sequence is: …“000”, “001”, “011”, “111”, “000”…. The first part of this problem involves generating the state diagram, which we show in Figure 23.19. The state diagram is similar to previous state diagrams for the 2-bit binary up counter with a count enable, but the individual states list the **CNT** output in the requested stoneage unary format. The state diagram supports the answer because there only four states, we represent using two bits of memory. The trick here is to convert the two bits of state memory to the desired 3-bit output.

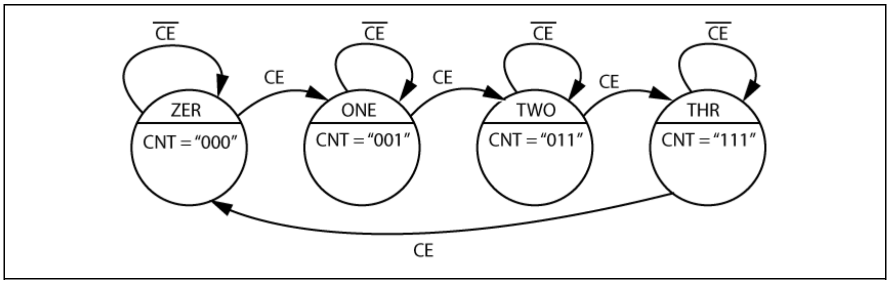
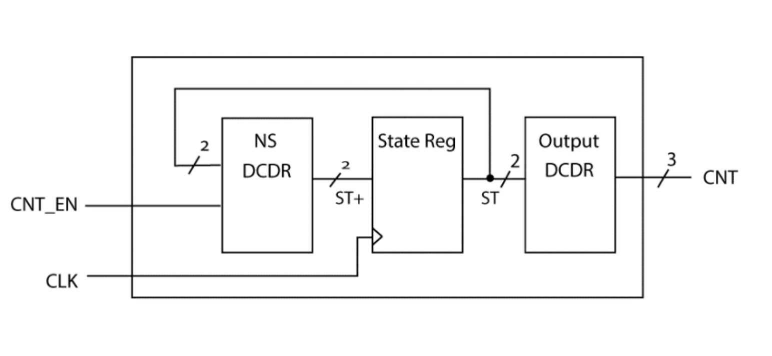


**Figure 23.19: The state diagram to support our solution.**

Next we need to deal with is converting a 2-bit value into a three bits value in the circuit. This is not a big deal once you realize we’ve been doing this with the NS decoder in all the problems up until now. Anytime you have an issue such as this, you should think: “decoder”. Thus, the solution is to include what we refer to as an “output decoder”. This output decoder essentially translates the state of the FSM (the present state, which is the value of the state registers) to our desired output. It’s a decoder; it’s not a big deal.

Figure 23.20(a) shows the circuit schematic for our solution, including the special placement of the output decoder. Figure 23.20(b) shows the PS/NS table, which is associated with the next state decoder.



| PS<br>CE ST | NS<br>$ST^+$ |
|---|---|
| 0 0 0 | 0 0 |
| 0 0 1 | 0 1 |
| 0 1 0 | 1 0 |
| 0 1 1 | 1 1 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

(a) (b)

**Figure 23.20: A block diagram for the circuit (a), and the associated PS/NS table. (b).**

The new item we need to define for this problem is the output decoder. Since it's a decoder, we only need to place the required information in a tabular format. Figure 23.21 shows the table we so eagerly seeking. In the table, the ST column is the output of the state registers, which makes it the present state of the FSM, while the **CNT** output is the stoneage unary output that the problem is looking for.

| ST | CNT |
|---|---|
| 0 0 | 0 0 0 |
| 0 1 | 0 0 1 |
| 1 0 | 0 1 1 |
| 1 1 | 1 1 1 |

**Figure 23.21: The definition of the output decoder for this solution.**

The outside world only sees the **CNT** output; the outside world does not know or care about the particular values of the state variables. This problem needs a binary count on the circuit's output. Thus, the particular choice of state variables is not important, so long as we keep it to no more than two bits of storage. This being the case, the output decoder definition in Figure 23.22 is equally as valid as the output decoder definition of Figure 23.21

| ST | CNT |
|---|---|
| 1 0 | 0 0 0 |
| 0 0 | 0 0 1 |
| 1 1 | 0 1 1 |
| 0 1 | 1 1 1 |

**Figure 23.22: Another possible definition of the output decoder for this solution.**

Figure 23.23 shows an example timing diagram for this solution. You've seen most of this information before. This only thing worth noting here is that the **CNT** output arbitrarily starts at "001" for no apparent reason.

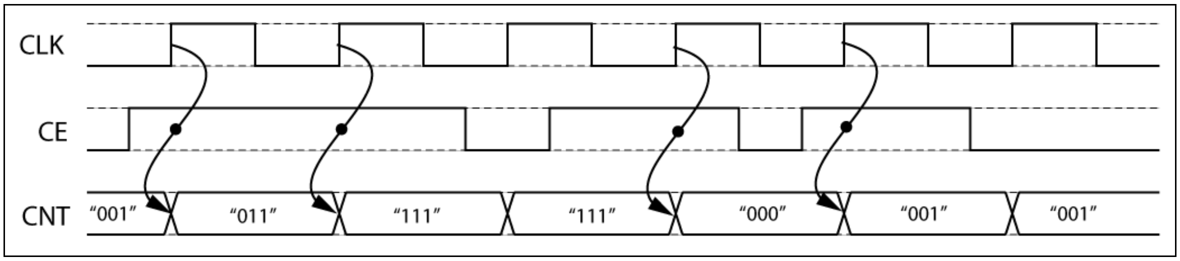
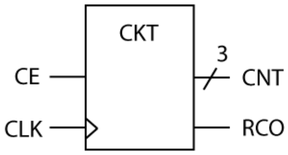


**Figure 23.23: An example timing diagram for this solution.**

**Example 23.8: FSM Design #8: 3-Bit Stoneage Unary Up Counter with Status Output**

Using the diagram on the right, design a FSM that implements a synchronous 2-bit stoneage unary up counter. The counter also has a control enable (**CE**, positive logic) input that allows the counter to count up. The counter also contains a "ripple carry out" output signal that indicates when the counter is currently outputting its maximum count value. Do not use more than two bits of storage in your FSM design. Implement the positive logic **RCO** signal with the next-state decoder and not with logic external to the FSM. Provide both a circuit diagram and state diagram for your solution.



**Solution**: This problem is similar to a previous problem, but now there is an **RCO** output, which states when the counter output is at its maximum count value. The use of **RCO**-type signals are common with counters as they are a status output, which provides useful information about the counter's state. If this counter did not have an **RCO** output signal, the digital designer would need to add logic external to the counter to generate the signal.

The problem constrains the solution to using only two bits of storage, which forces us to use an output decoder to generate the proper 3-bit stoneage unary output count (**CNT**). The new issue with this problem is how we represent the **RCO** signal in the state diagram. Figure 23.24 shows the state diagram for our solution. The **RCO** signal is a function of the count only; the count is a function of the present state. In Figure 23.24 we choose to represent **RCO** as a Boolean logic variable, rather than write an equation such as "**RCO**='1' ". While either approach is acceptable, the approach in Figure 23.24 makes the state diagram more readable.

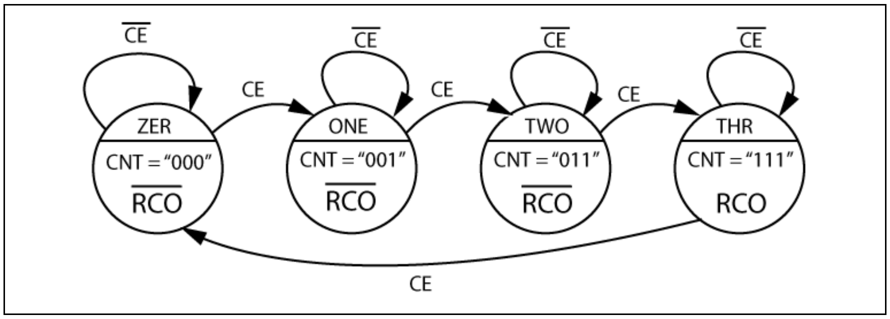
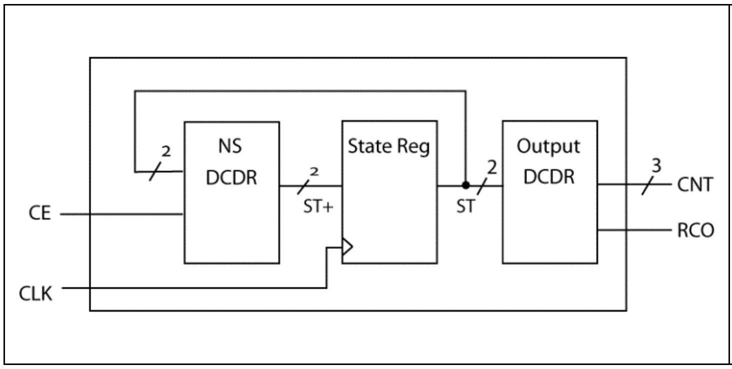


**Figure 23.24: The state diagram to support our solution.**

The next step is to generate a circuit diagram, which we show in Figure 23.25(a). The state registers module consists is two bits wide as the problem specifies. The PS/NS decoder does not change from previous problems in that this problem only added a new output signal. Figure 23.25(b) shows that the output signal is an output from the output decoder, which makes sense because **RCO** is only dependent upon the present state.



| PS<br>CE ST | NS<br>$ST^+$ |
|---|---|
| 0 0 0 | 0 0 |
| 0 0 1 | 0 1 |
| 0 1 0 | 1 0 |
| 0 1 1 | 1 1 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.25: A block diagram for our solution (a), and the associated PS/NS table.**

The problem requires an output decoder that outputs the correct **CNT** and **RCO** values. Figure 23.26 only shows the output decoder for our solution. The two outputs are dependent upon the present state. The **CNT** column shows the stoneage unary outputs for the corresponding state (which is a 2-bit binary sequence); the **RCO** column shows that the **RCO** signal is only asserted the **CNT** value is "111", which is the maximum count value for this counter.

| ST | CNT | RCO |
|---|---|---|
| 0 0 | 0 0 0 | 0 |
| 0 1 | 0 0 1 | 0 |
| 1 0 | 0 1 1 | 0 |
| 1 1 | 1 1 1 | 1 |

**Figure 23.26: The decoder definition for this solution.**

Figure 23.27 shows an example timing diagram for this solution. This timing diagram is similar to previous solutions but there is now a **RCO** signal. The timing diagram shows that the **RCO** signal asserts when the count is at its maximum value, which for this counter is "111".

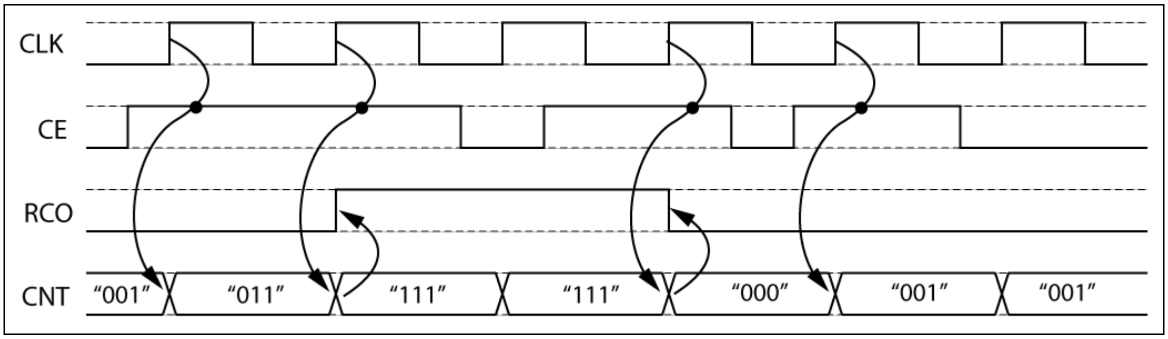
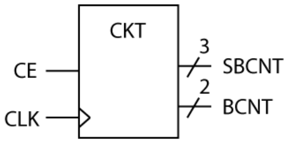


**Figure 23.27: An example timing diagram for this solution.**

**Example 23.9: FSM Design #9: Specialty Up Counter**

Using the diagram on the right, design a FSM that implements a synchronous counter that has both a 2-bit binary (**BCNT**) and a 3-bit stoneage unary (**SBCNT**) output. The FSM also has a count enable input (**CE**), which serves two functions. When the **CE** is asserted, both counter outputs increment of the active clock edge. When **CE** is not asserted, the neither count is incremented. Additionally, the binary count (**BCNT**) outputs “00” when **CE** is not asserted, but returns to its original value when **CE** is reasserted. The **SBCNT** output does not increment, but does not go to zero when the **CE** input is not asserted (as does the **BCNT** output). Provide both a circuit diagram and state diagram for your solution.



**Solution**: The counter has two different count outputs: binary and stoneage unary. In terms of counting, they both react to the same to the **CE** input, meaning that they both increment their counts on the active clock edge when **CE** is asserted. There is, however, a major difference in how the two count outputs react to the **CE** input.

The **SBCNT** essentially does not react to the **CE** input, and is thus similar to the count outputs in previous examples. The **BCNT** output needs to react to the **CE** input: when the **CE** input is asserted, it outputs its count value; when the **CE** input is not asserted, it outputs zero (“00”). What this is officially describing is that the **SBCNT** output is a function of only the state (state variables) of the FSM, while the **BCNT** output is a function of the state and the **CE** input. The ramifications of this are the **SBCNT** output can only change when the state changes, which is synchronized with the clock, while the **BCNT** output can change when **CE** changes as well as when the state changes. Thus, changes in the **BCNT** output are not necessarily synchronized with the circuit’s active clock edge because it is also a function of the **CE** input. We address these special output characteristics more formally later in this chapter.

Because the two outputs in this example have significantly different characteristics, we would expect some new symbology associated with state diagrams to model those output characteristics. Figure 23.28 shows this new symbology along with a description below.

- The value of the **SBCNT** output only depends on the present state of FSM, so we can express the **SBCNT** output inside of the state bubbles. We can do this because the **SBCNT** output is only a function of the present state of the FSM.
- The value of the **BCNT** output depends on both the **CE** input as well as the state. Because it is not a strict function of the present state of the FSM, we can’t list it inside the state bubble as we did for **SBCNT**. The **CE** signal has two functions in this FSM: 1) it controls the state transitions, and 2) it controls the **BCNT** output. Because we already used the **CE** signal in conjunction with the transition arrows to describe the state-to-state transitions, we need to then include the **BCNT** output next to the **CE** input that are associated with the state transitions, which is what we did in the state diagram of Figure 23.28. This symbology requires getting used to, but the underlying approach is straightforward. The main item you must realize when working with these state diagrams is differentiating between the inputs and outputs. Because the output (**BCNT**) is dependent upon the input (**CE**), we list the output with the input. When we list outputs next to the state transition arrow, that output is then associated with the state the transition arrow is leaving. Finally, the **BCNT** output can change whenever **CE** changes; but a change in **CE** does not necessarily cause a state transition because state transitions can only happen on active clock edges.

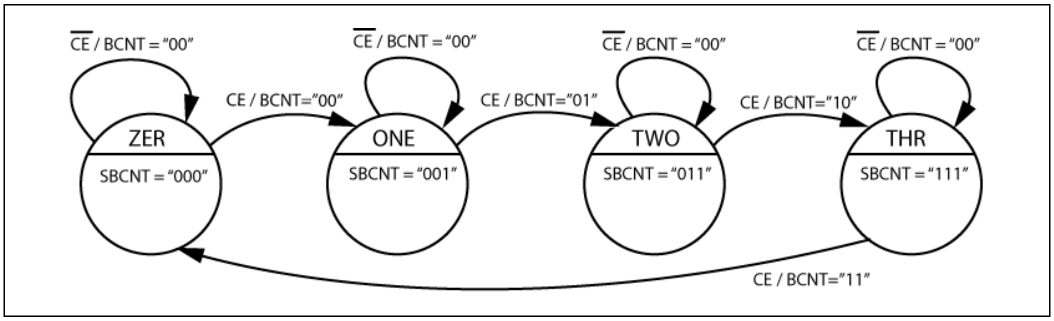
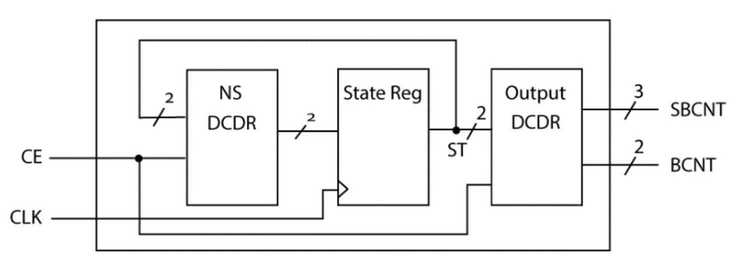


**Figure 23.28: The state diagram to support our solution.**

Figure 23.29(a) shows the block diagram for our solution. The block diagram of is similar to the previous example, the only difference being the output decoder now includes the **CE** signal as an input. The output decoder must include **CE** as an input because that inputs controls whether the **BCNT** is outputting the current count value, or outputting zeros. The **CE** input to output decoder changes the structure of the output decoder, but the next-state decoder remains unchanged. The next-state decoder of Figure 23.29(b) is the same as a previous example even though the output of the FSM is now responsible for two different count outputs.



| PS<br>CE ST | NS<br>ST+ |
|---|---|
| 0 0 0 | 0 0 |
| 0 0 1 | 0 1 |
| 0 1 0 | 1 0 |
| 0 1 1 | 1 1 |
| 1 0 0 | 0 1 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 1 |
| 1 1 1 | 0 0 |

**(a)** **(b)**

**Figure 23.29: A block diagram for the FSM (a), and a description of the next-state decoder (b).**

The next part of this solution is to model the output decoder. This is a decoder, so we need to generate a table that describes the output in such a way as to solve the problem. Looking at the output decoder module in Figure 23.29(a) shows that the decoder has two input signals: ST (a 2-bit bundle) and **CE**; the output decoder has two outputs **BCNT** (a 2-bit bundle) and **SBCNT** (a 3-bit bundle). Figure 23.30 shows the final decoder definition. This table looks a bit different. We formatted the table slightly different in order to show some important details, which we describe below.

- ST in Figure 23.30 is the present state (not the next state); ***the output decoder always bases its output on the present state.***
- The decoder has three inputs: two bits of state information (ST) and the **CE** input. We place the state values before the **CE** values so that the state values don't change with every row change. The **CE** input changes with every row in the table because we list it as the LSB. The arrangement of the signals in the table is arbitrary, but we always list them in the most understandable manner.
- The states now come in pairs; the first two rows are associated with the "00" state, the next two rows are associated with "01" state, etc. The table uses double lines to emphasize the changes in state variables (ST).

- The **SBCNT** changes only when the state changes. This is another way of saying the value of the **SBCNT** output is only dependent upon the state: it's not dependent upon the **CE** input. You can see that for each of the pair between the double horizontal lines in the table, neither the state (ST) nor the **SBCNT** output changes. Because **SBCNT** is a function of only the present state of the FSM, the **SBCNT** can only change when the state changes. In other words, changes in the **SBCNT** output are synchronized with the active edge of the state register's clock signal.

- The value of the **BCNT** output changes for each of the state pairs (except for the ST="00" case). This is another way of saying that the **BCNT** output is dependent upon both the state of the FSM (ST) and the **CE** input. Recall that when the **CE** signal is not asserted (**CE**='0'), the FSM outputs "00" for **BCNT**; otherwise when **CE** is asserted, the FSM outputs the associated count (which happens to be the same as the state variables, or ST). Because the **BCNT** output is a function of both the present state and the **CE** input, it can change when either the state changes, or when the **CE** input changes. Changes in the **BCNT** output are not necessarily synchronized with the state register's clock; because it is function of the **CE** input, **BCNT** can change anytime **CE** changes.

| ST CE | SBCNT | BCNT |
|---|---|---|
| 0 0 0 | 0 0 0 | 0 0 |
| 0 0 1 | 0 0 0 | 0 0 |
| 0 1 0 | 0 0 1 | 0 0 |
| 0 1 1 | 0 0 1 | 0 1 |
| 1 0 0 | 0 1 1 | 0 0 |
| 1 0 1 | 0 1 1 | 1 0 |
| 1 1 0 | 1 1 1 | 0 0 |
| 1 1 1 | 1 1 1 | 1 1 |

**Figure 23.30: The output decoder definition for this example.**

Figure 23.31 shows an example timing diagram for our solution, which could be the most important timing diagram in this text up until now. Here is some helpful verbage.

- The most striking feature is the difference between the **BCNT** and **SBCNT** outputs. The output values are different, as they are two different counts. Note when these counts change values. The **SBCNT** only changes value on a rising clock edges. The **BCNT** can change values both on the rising clock edges and when the **CE** signal changes. Regardless of the rising clock edge, an unasserted **CE** signal causes the **BCNT** output to be "00". In other words, **BCNT** is a function of both the present state and the **CE** signal; changes in **CE** cause an immediate change to the **BCNT** output.

- We use arrows to show how the **CE** input affects the **BCNT** output. The first rising edge of the CE signal causes a value to appear on the **BCNT** output. The **BCNT** value of "01" does in fact correspond to the **SBCNT** value of "001". The FSM is effectively remembering the state; the unasserted **CE** signal is effectively temporarily clearing the **BCNT** output, but then restoring the **BCNT** output once the **CE** signal asserts. We can see this behavior happen between the signal edges labels '2' & '3' as well as '4' & '5' in the timing diagram.

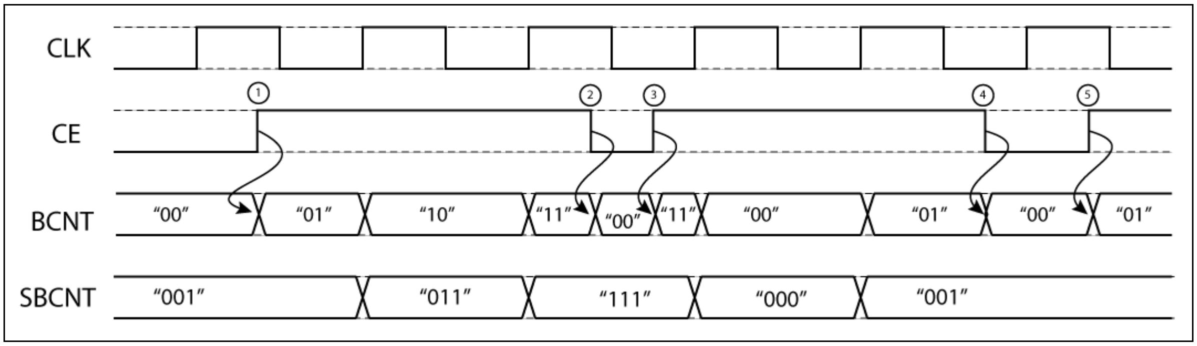


**Figure 23.31: An example timing diagram for our solution.**

## 23.3 FSM Illegal State Recovery

The FSMs we've examined at this point contained a certain quality that is not always present in all FSMs. Note that all FSM designs up to this point used every code available in FSM's state registers. For example, each example we explored contained four states, which was the maximum number of states that we could represent using two single-bit storage elements.

Consider the case where we have a count sequence of five numbers that we implement using a FSM. The FSM for this case requires a minimum of three 1-bit storage elements (or a register with a data width of at least three). The potential problem here is that with three flip-flops, we can represent up to eight states. What happens to the three extra states that are not associated with the desired count sequence?

If you need to create a super solid FSM design, you need to know what all unused states are doing. The problem is that some unforeseen condition may put your FSM in a state that is not part of the desired sequence. Then what happens? The idea is to design your FSM to "fix" itself if it by chance finds itself in a state that it was not intended to be in. What we need to do is design the FSM such that it has "illegal state recovery". The next example sheds light on the problem.

**Example 23.10: Counter Design with Illegal State Recovery**

Design a counter that counts in the following sequence: 0, 5, 7, 3, 6, 0, 5… Use a minimum number of storage elements in your design. Direct all unused states to the state associated with the zero count.

***Solution:*** This problem is similar to the other counter problems except there are more numbers in the count sequence and those numbers are not in a typical counting order. The first step is to generate a top-level BBD, which we show in Figure 23.32.

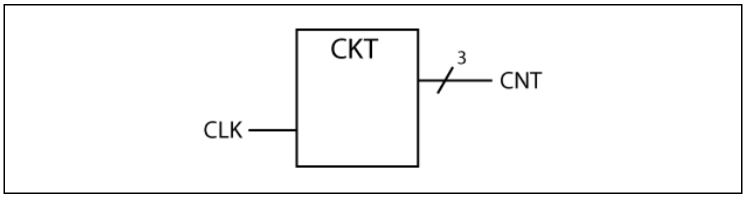


**Figure 23.32: The top-level BBD for this example.**

The next thing we need to do is to generate a state diagram that models a solution the problem. Figure 23.33 shows a first pass at the state diagram. We consider this a first pass because while it satisfies the "counting" part of the problem, it doesn't provide illegal state recovery. We need to add illegal state recovery, so we need to represent the states not listed in the counting sequence.

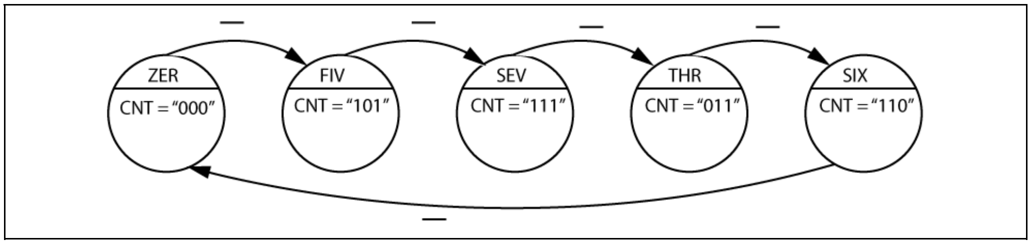


**Figure 23.33: The initial state diagram for Example 23.10.**

What we're trying to avoid in this problem is the generation of *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 inherently contains hang states. Figure 23.34 shows an example of a state diagram with hang states. In Figure 23.34, we do indeed have the desired sequence; but the state diagram lists all possible states associated with the state register (which is three bits wide).

In reality, we implement FSMs with real circuitry, which means they are susceptible to various types of noise[1]. It just may happen that the noise places your FSM in a state that is not part of the desire sequence, which according to Figure 23.34, puts you in a hang state and you'll never make it back to the desire sequence. The FSM is thusly hung because it is stuck in a hang state. Figure 23.34 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[2]. Bummer!

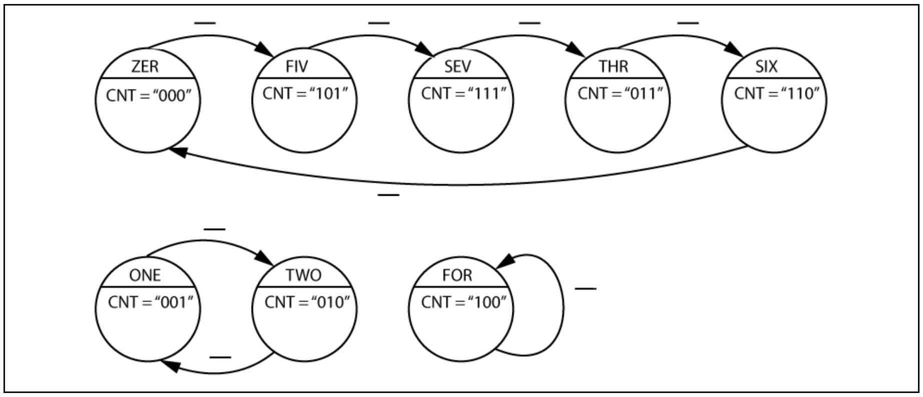


**Figure 23.34: A state diagram containing hang states and other terrible things.**

The approach that saves the day is to direct the unused states back to a state in the desired count sequence. If for some reason your FSM finds itself[3] in a hang state, you'll quickly (in one clock cycle) return to a count value in the desired sequence. Figure 23.35 shows the associated state diagram for this approach. The problem description states that you should direct all of your unused states back to state "000". From this point, it is not a big deal to generate the PS/NS table using techniques from previous examples.

[1] This refers to unwanted electronic effects. A loud stereo will most likely have not effect on your digital circuit designs.

[2] Imagine if your FSM were 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. 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.

[3] Yes Virginia, FSMs are self-aware (or about as self-aware as the average academic administrator).

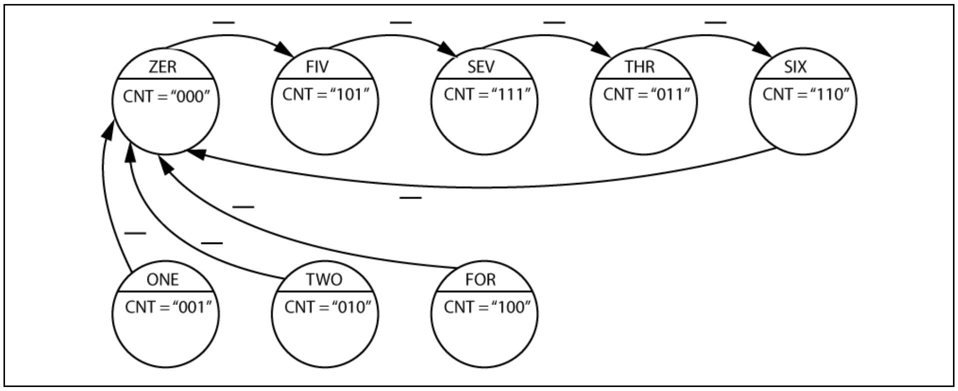
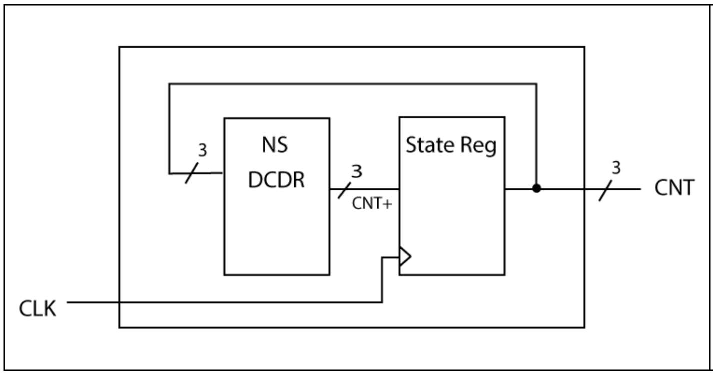


**Figure 23.35: The state diagram with hang-state recovery.**

Figure 23.36(a) shows the underlying hardware for our solution. We can use the output of the state register as the **CNT** output, which means we do not need to include an output decoder in the hardware. Figure 23.36(b) shows the PS/NS table, where the shaded rows in represent states not included in the desired sequence. Each of the shaded rows do in fact direct the FSM back to ZER (**CNT**="000") state called for in the problem description. Now that we include illegal state recovery in our FSM design, we say that the FSM is *self-correcting*. Making your FSM designs self-correcting is important because statistically speaking, you're going to have unused states in your FSM because of the binary nature of the elements that FSMs use to store the state variables.



| **PS**<br>**CNT** | **NS**<br>**CNT$^+$** |
|---|---|
| 0 0 0 | 1 0 1 |
| 0 0 1 | 0 0 0 |
| 0 1 0 | 0 0 0 |
| 0 1 1 | 1 1 0 |
| 1 0 0 | 0 0 0 |
| 1 0 1 | 1 1 1 |
| 1 1 0 | 0 0 0 |
| 1 1 1 | 0 1 1 |

**(a)** **(b)**

**Figure 23.36: A block diagram for our FSM (a), and the PS/NS table (b).**

**Example 23.11: FSM Design #10: Self-Correcting Specialty Counter**

Design a counter that repeatedly counts in the following sequence: {4, 18, 20, 30, 8}. The circuit has a **HOLD** input (positive logic) that stops the counter from counting while asserted. When the **HOLD** input is asserted, the count output is halved. Make the circuit self-correcting by directing all unused states to count = 4. Outputs associated with unused states should be all 1's. Minimize the amount of memory in circuit. Provide the top two levels of circuit diagrams and a state diagram for your solution.

**Solution**: This problem has several interesting attributes that are not patently obvious from reading the problem. This problem does not provide you with all the information, as evident from the lack of a BBD. So let's start thinking it out on our way to drawing a high-level black box diagram. The following bullets provide an example of issues that you the design must deal with on your way to solving this problem.

- The maximum value of the output count is 30, which indicates we need five bits on the output to represent the count. This may lead you to think that the state registers need to be five bits. However, since there are only five unique numbers in the count, the state registers only need to be three bits wide.
- According to the previous bullet, we have three bits for the state registers, but only five states that we need to represent the count. This means the FSM has three unused states that we need to account for in both the next state decoder and the output decoder.
- The FSM has one 5-bit output, which is the count. The exact form of this count output depends on the value of the **HOLD** input, thus this is a Mealy-type output.

At this point, we're ready to draw the top-level BBD. Figure 23.37 shows where the previous bullets have left us.

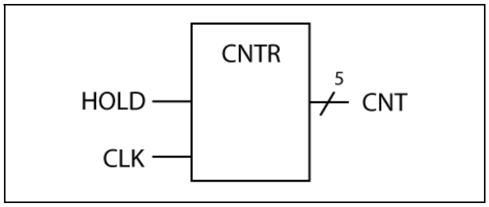


**Figure 23.37: The top-level black box diagram for this problem.**

The lower-level schematic of our circuit contains the three standard sub-modules of an FSM. Figure 23.38 shows the detailed schematic for this solution. Note in Figure 23.38 that the output count is 5-bits and that the output decoder is a function of both the present state (ST) and the external input (**HOLD**).

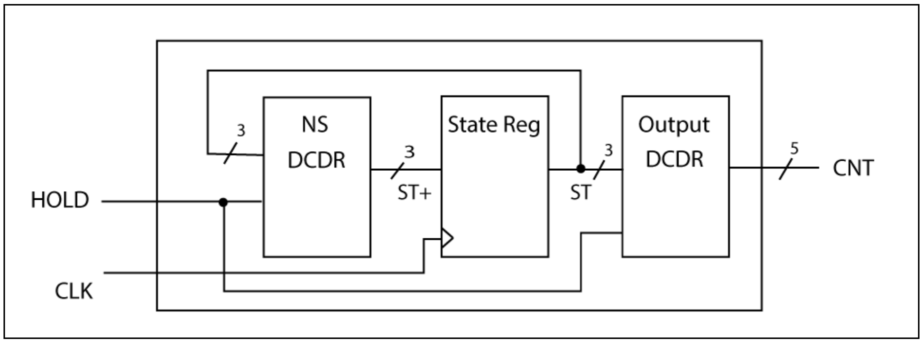


**Figure 23.38: The lower-level BBD to support our solution.**

The next step is to generate either the state diagram or the definitions for the next-state and output decoders. Unlike previous problems we've worked with, we use the same table for both the next state decoder and the output decoder. This approach can be confusing, so we start by reminding ourselves that the next state (ST+ in this case), is only associated with the next-state decoder. Similarly, when we work with defining the output decoder, it is always dependent upon the present state.

We have one more decision to make before we proceed with the generating the table for the decoder. This counter has five output values in its count sequence, yet we already established that we'll represent those five values with a 3-bit state register. This means that we must correlate the state values to the output count values. How exactly we relate these two sets of values is arbitrary, so we decide to relate them in a way that causes us the least confusion.

Our approach is to relate the individual state values in order {0, 1, 2, 3, 4} to the given order of the count sequence {4, 18, 20, 30, 8}. Our only constraint here is that the output count sequence is in the given order. The combined decoder table is flexible in the way it can model the FSM, so we always strive to model the FSM in a way that reduces our chances of making an error. This means that we associate the first value in the output count (4) with the first possible state values ("000").

Table 23.1 shows the completed combined next-state & output decoder table. Here are some of the important things to notice about this table.

- We use the abbreviation **H** for the **HOLD** input
- We list the three variables of the PS on the far left of the next-state decoder inputs. This is not the only way to do this but this approach makes the table more readable.
- The table also includes double lines around the same PS states for increased readability.
- The table includes the outputs in both binary and decimal for increased clarity.
- The row pairs delineated by double lines representing the states show that for any given state, the output can be different because the output is dependent upon the input.
- We arbitrarily assign the state "101"→"111" (5-7) as the unused states. The table uses darkened cross-hatching to indicate this attribute.
- We direct the unused states back to the "000" state to make the FSM self-correcting. The problem stated to make the FSM self-correcting by directing unused states back to the 4 count; the 4 count corresponds to the "000" state.

| PS | NS | output | |
|---|---|---|---|
| ST H | $ST^+$ | CNT | CNT (dec) |
| 0 0 0 0 | 0 0 1 | 0 0 1 0 0 | 4 |
| 0 0 0 1 | 0 0 0 | 0 0 0 1 0 | 2 |
| 0 0 1 0 | 0 1 0 | 1 0 0 1 0 | 18 |
| 0 0 1 1 | 0 0 1 | 0 1 0 0 1 | 9 |
| 0 1 0 0 | 0 1 1 | 1 0 1 0 0 | 20 |
| 0 1 0 1 | 0 1 0 | 0 1 0 1 0 | 10 |
| 0 1 1 0 | 1 0 0 | 1 1 1 1 0 | 30 |
| 0 1 1 1 | 0 1 1 | 0 1 1 1 1 | 15 |
| 1 0 0 0 | 0 0 0 | 0 1 0 0 0 | 8 |
| 1 0 0 1 | 1 0 0 | 0 0 1 0 0 | 4 |
| **1 0 1 0** | **1 0 0** | **1 1 1 1 1** | **n/a** |
| **1 0 1 1** | **1 0 0** | **1 1 1 1 1** | **n/a** |
| **1 1 0 0** | **1 0 0** | **1 1 1 1 1** | **n/a** |
| **1 1 0 1** | **1 0 0** | **1 1 1 1 1** | **n/a** |
| **1 1 1 0** | **1 0 0** | **1 1 1 1 1** | **n/a** |
| **1 1 1 1** | **1 0 0** | **1 1 1 1 1** | **n/a** |

**Table 23.1: The combined next-state & output decoder definition for this example**

Now that the combined next-state & output decoder table are complete, we can move onto the state diagram. While the table description of the FSM in Table 23.1 is complete, it is cumbersome to read; a better option is to transfer the information in the table to a state diagram. Figure 23.39 shows the associated state diagram. Here are a few important things to note regarding the state diagram.

- The state diagram is not 100% complete, as it does not represent the fact that our FSM is self-correcting. We address this later.
- The FSM does not include how we encode the states. We consider this a low-level implementation detail and we are designing at a higher level.
- We use **H** to represent the **HOLD** input in order to save space and clutter in the state diagram.
- The external input is associated with the state transition arrows because it controls the state transitions from a given state.
- The outputs of the FSM are a function of both the state and external input. In this example, the external output is always different in every state as it is a function of the external input. This requires that we associate the external outputs with the external inputs.
- We put in a special annotation to indicate that the binary values are the **CNT** output.

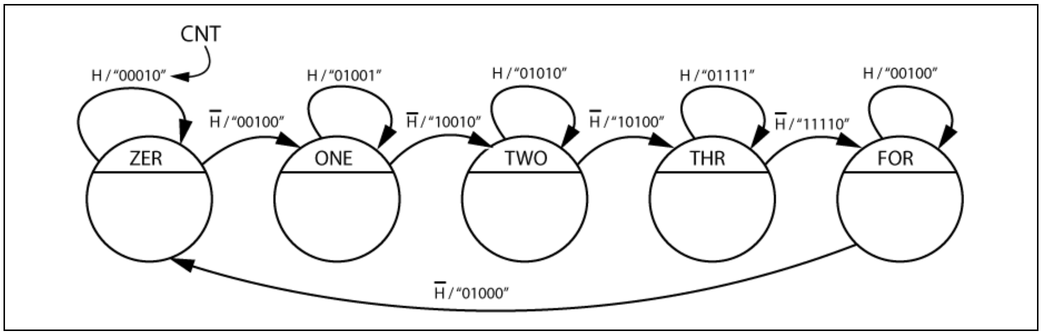


**Figure 23.39: The partial (but adequate) state diagram to support our solution.**

The state diagram in Figure 23.39 does not completely represent the information in Table 23.1. To be complete, we must include the information associated with the unused states, since our work included making the FSM self-correcting. The state diagram in Figure 23.40 represents the complete state diagram for this problem. Here is some extra information to note about Figure 23.40:

- Each of the three unused states unconditionally transition back to state "FOR", which the original problem stated. Directing unused states back to a valid state makes this FSM self-correcting.
- We show the **CNT** output in the top five states as a Mealy-type output, which is because the value of **H** determines the value of the **CNT**. The **CNT** in the unused states is always the same, so we list that **CNT** value in the state bubble. Though this listing makes it appear like a Moore-type output, it is a Mealy-type output based on the five valid states in the FSM.

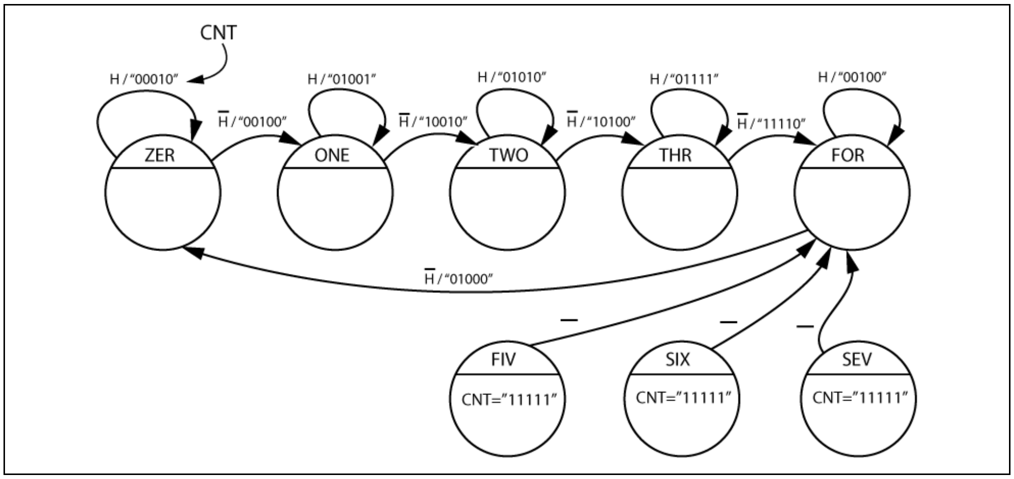


**Figure 23.40: The complete state diagram to support our solution.**

## 23.4 FSM Overview and Summary

The term "Finite State Machine" has many official meanings and definitions in digital-land. As you saw previously, any circuit that has the ability to remember something (namely bits), can be regarded as having a "state". A circuit-oriented definition of a FSM is this: *a circuit whose behavior can be modeled using the concept of "state" and the transitions between the various states in that circuit.* We soon move onto using FSMs primarily as controller circuits, or *a circuit that control other circuits*.

People use FSMs in one form or another in many different technical disciplines and each discipline seems to have its own particular flavor of representing FSMs. Despite these many flavors to modeling FSM, always keep in mind that the best approach is to be clear in a way that expedites the transfer of information. Always remember that the state diagram is a model that visually describes the behavior of the FSM.

## 23.5 High-Level Modeling of Finite State Machines

Digital design typically classifies FSMs as one of different two types: Moore-type or Mealy-type. In this text, we simplify this definition as follows: there is only one type of FSM, but FSMs can have Moore-type and/or Mealy-type outputs. All FSMs share the same properties: the only difference is the two types of FSM outputs.

Figure 23.41 shows a basic model of an FSM. We can abstract the FSM's internal circuitry into three separate blocks: 1) Next State Decoder, 2) the State Registers, and 3) the Output Decoder. The output decoder can have two types of outputs, which we refer to as Moore and Mealy-type outputs; Moore-type outputs are a function of the present state of the FSM while Mealy-type outputs are a function of both the FSM's present state and the external inputs. Table 23.2 provides a detailed description of the FSM's individual modules.

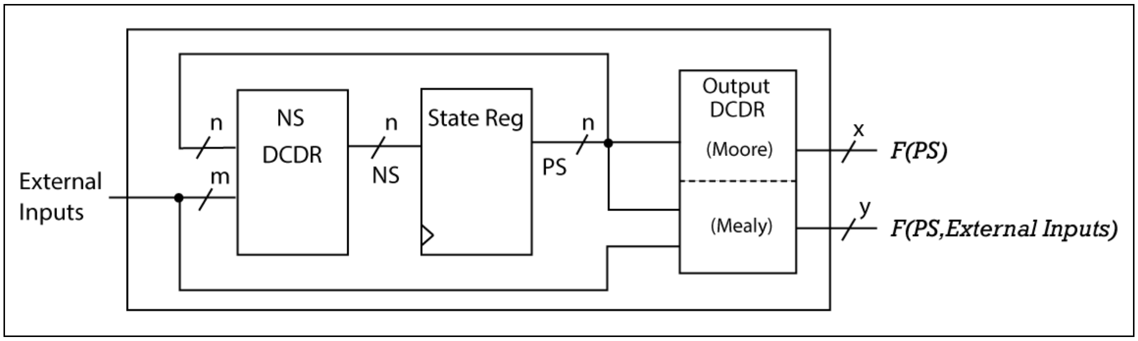


**Figure 23.41: The lower-level BBD for a generic FSM.**

| Module | Description and Comments |
|---|---|
| **State Registers** | The **State Registers** represent the memory elements in the FSM. The term *register* implies the circuit is a synchronous storage element. The state register is the only sequential module in an FSM; the other two modules are both combinatorial circuits. The state registers store the *state variables* of the FSM; the value stored in the state registers is the state of the FSM. |
| **Next State Decoder** | The **Next State Decoder** is a combinatorial circuit that provides excitation input logic to the state register module. The next state logic generally has two types of inputs, which provide the *excitation inputs* to the state registers: 1) the current value of the state variables (the present state, and, 2) the inputs from the external world. Excitation inputs to the state registers determine the *next state* of the state register. On the next active clock edge, the data inputs to the state registers becomes the next state of the FSM, which is why we refer to next state decoder as the *next state logic*. The external inputs to the next-state decoder function as status signals from the world outside of the FSM. |
| **Output Decoder** | The **Output Decoder** is a combinatorial circuit that generates the external outputs of the FSM. The output decoder is responsible for generating the two types of FSM outputs: Moore-type outputs and Mealy-type outputs. Moore-type outputs are a function of the FSM's state only, while Mealy-type outputs are a function of both the FSM's state and the external inputs to the FSM. The outputs from the output decoder generally serve as control signals to the device(s) controlled by the FSM. |

**Table 23.2: A detailed description of the three main FSM functional blocks.**

## 23.6 The FSM: Symbology Overview

Probably the hardest thing about FSMs is understanding the state diagram symbology. The good news is that it's relatively simple once you work with it. Although we developed an intuitive approach to the FSM structure and symbology earlier in this chapter, we present it again from a different angle.

### 23.6.1 The State Bubble

FSMs use the state bubble to represent a particular *state* in an FSM. Figure 23.42(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 visually delineate it from other states, which is why the state bubble contains identifying information. State bubbles provide the state with a symbolic name that identifies the purpose of that state to the human reader.
- Timing diagrams represent the states by the time slots representing the possible states. Figure 23.42(b) shows that the boundaries of these time slots delineated the associated active edges of the FSM's clock input, which is the clock input to the state registers.. Figure 23.42(b) show that the state registers are rising-edge triggered (RET) because the rising clock edge defines the state boundaries.

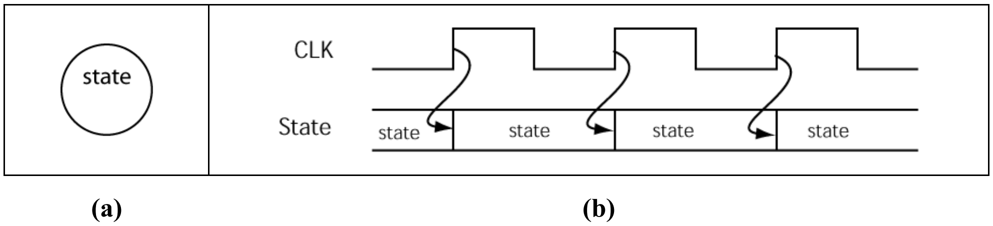
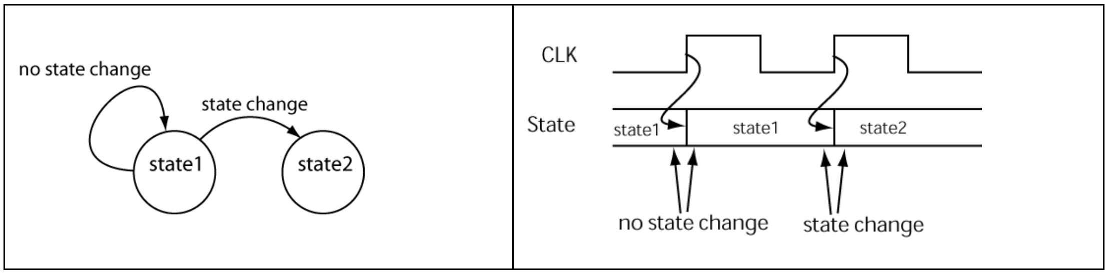


**Figure 23.42: The State Bubble and associated timing diagram.**

### 23.6.2 The State Diagram

The state diagram is one of many methods we use to model FSMs. The main purpose of the state diagram is to convey meaning and understanding to the human viewer. State diagrams provide four main forms of information: 1) the states in the FSM, 2) the state transitions the FSM makes, 3) the input conditions controlling the state transitions, and, 4) the output values associated with the FSM. Figure 23.43(a) shows a fragment of a state diagram. The following verbage describes some of the key features of this state diagram.

- We refer to the terminology describing how a FSM goes from one state to another as a *state transition* or just *transition*. State diagrams use singly directed "arrows", directed from the source state to the destination state to represent state transitions.
- There are only two possible state transitions in a state diagram from a given state. On the active clock edge, a transition can occur from, 1) one state to another state (indicated by the "state change" label in Figure 23.43(a)), or, 2) the FSM can remain in the same state (indicated by the "no state change" label in Figure 23.43(a)). We refer to the "no state change" arrow as a "*self-loop*".
- The state diagram contains no explicit clock signal; the clock signal is implied rather specifically listed. 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 the FSM.
- The two states in Figure 23.43(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.
- The state names in Figure 23.43(a) give no indication how we would represent the states if we were to implement the FSM. In other words, the state diagram provides no commitment to the actual state variable assignment that disambiguates the states on a hardware level.
- The relation between the timing diagram in Figure 23.43(b) and the state diagram in Figure 23.43(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. The state bubble essentially represents all the time between any two active edges of the system clock. The state transition arrow represents what happens on each of the FSM's 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 state transition occurs on every active clock edge, but sometimes it transitions back to the same state.
- The concept of Present State (PS) and Next State (NS) is somewhat hard to define in a timing diagram such as the one in Figure 23.43(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 changes as you traverse the timing diagram. PS/NS tables do a better job of presenting present and next-state information.



(a) (b)

**Figure 23.43: A state diagram (a) and the associated timing diagram (b) with some interesting details.**

### 23.6.3 State Transitions Controlling Conditions

As you would guess from examining the state diagram of Figure 23.43(a), there must be some mechanism that decides which transition will occur from a given state on the next active clock edge. In Figure 23.43(a), state1 has two arrows leaving the state, which mean there are conditions associated with those arrows that decide on which transition occurs.

There are two forms of information that determine the transition a FSM takes: 1) at least one of the external inputs to the FSM, and, 2) the present state of the FSM[4]. The external inputs to a FSM are generally status signal from the circuit the FSM is controlling. Each state has its own set of conditions that govern transitions, so we're concerned on a state-by-state basis what external input conditions determine the state transitions from a given state. Figure 23.44 shows that we indicate the conditions governing transitions by placing the conditions next to the state transition arrows. On this note, there are three important things to keep in mind:

1) The conditions associated with the state transition arrows leaving a given state must be mutually exclusive. This means that there can never be the same input conditions associated with two different transitions arrows leaving the same state.

2) The set of conditions associated with a particular state must be complete, meaning it must provide a transition arrow for every possible meaningful combination of input conditions. If there is a set of conditions in given state not covered by the associated state transition arrows, the FSM won't know what to do[5]. State diagrams should leave no room for guessing, if they do, their behavior will not be deterministic (which is an impressive way of saying your FSM won't always work as you intend).

3) If the transition is unconditional, then the state diagram indicates this by listing a "don't care" symbol by that transition.

[4] Recall that the PS and the external inputs are the inputs to the next-state decoder.

[5] 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 unintentionally forgotten.

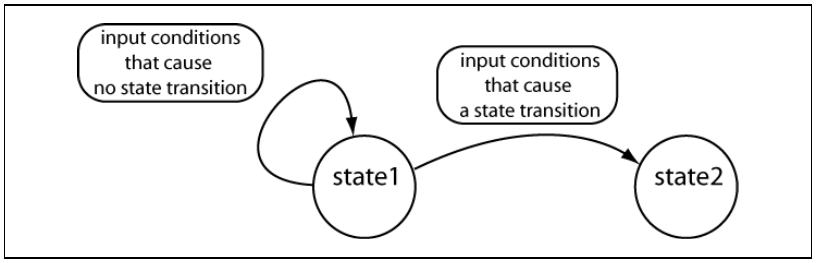

input conditions that cause no state transition
input conditions that cause a state transition
state1
state2

**Figure 23.44: How state diagrams indicate the conditions associated with state transitions.**

### 23.6.4 FSM External Outputs

The external outputs from a FSM are generally “control signals” that are controlling other circuits. The state diagram has different states and thus the control signals output from one state are generally not the same as control signals output from other states, so the FSM is performing different control functions based on the different states.

There are two different types of outputs in a FSM: Mealy-type outputs and Moore-type outputs. Although these outputs are similar 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.

Since Moore-type outputs are a function of the state variables only, we represent them by placing their values inside the state bubble. Figure 23.45 shows a state diagram that uses this approach. There can be any number of outputs represented inside the bubble.

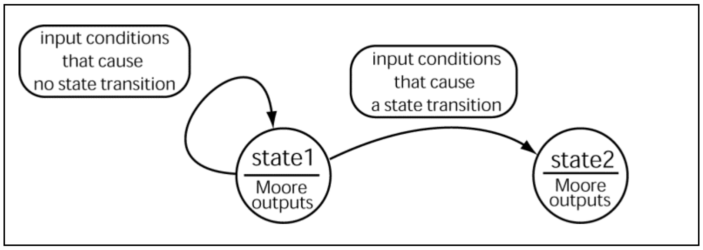

input conditions that cause no state transition
input conditions that cause a state transition
state1
Moore outputs
state2
Moore outputs

**Figure 23.45: The State Bubble with associated Moore outputs.**

We can’t represent Mealy-type outputs inside 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. We separate external inputs and outputs with a forward slash. Figure 23.46 shows an example of this approach; we comma-separate multiple Mealy-type outputs.

Figure 23.46 lists two sets of Mealy-type outputs because there are two transitions from state1. The arrows are associated with the state transitions, which are based upon the current external inputs; the Mealy-type outputs are also a function of those same inputs. Since the Mealy-type outputs are a function of the external inputs, we represent them by placing them next to the external inputs. ***We always associated Mealy-type outputs with the state the arrow is leaving (and not the state the arrow is entering).***

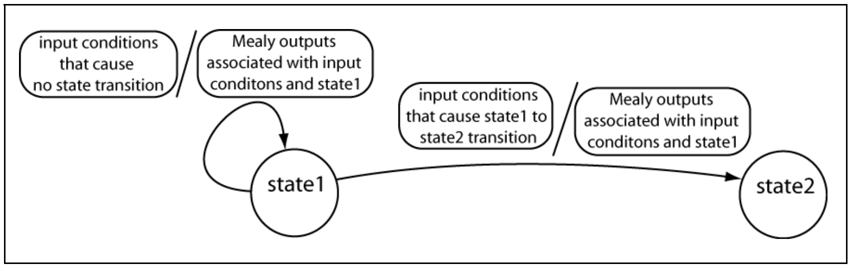

input conditions that cause no state transition
Mealy outputs associated with input conditons and state1
input conditions that cause state1 to state2 transition
Mealy outputs associated with input conditons and state1
state1
state2

**Figure 23.46: Representing Mealy-type outputs in a state diagram.**

In addition, we can represent both Mealy and Moore-type outputs in the same state diagram. Figure 23.47 shows an example of a state diagram that contains both Mealy and Moore-type outputs.

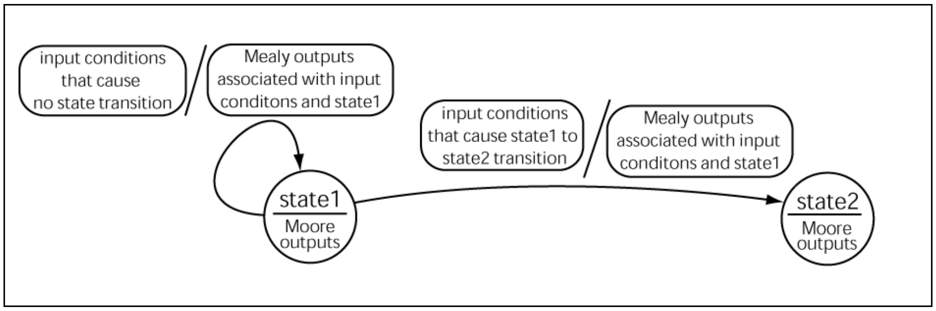

input conditions that cause no state transition
Mealy outputs associated with input conditons and state1
input conditions that cause state1 to state2 transition
Mealy outputs associated with input conditons and state1
state1
Moore outputs
state2
Moore outputs

**Figure 23.47: A state diagram that has both Mealy and Moore-type outputs.**

### 23.6.5 Non-Important FSM Outputs

While there are times when you may need to generate a “complete” state diagram, you must remember that the state diagram is primarily meant for a human viewer. Combine this notion with the fact that even a modest sized FSM can have enough external inputs and outputs to quickly compromise the readability of the state diagram.

There are generally many outputs from a FSM, but the state diagram does not necessarily need to assign a value for every output in every state. If in any state a given output is not assigned, it is assumed to be a “don’t care” in the context of that state, which means that output does not affect the external operations associated with that state. You can thus omit outputs from a given state if those outputs don’t matter for that state. It is not necessarily bad practice to list all external outputs for each state, but your state diagram becomes harder to understand.

### 23.6.6 Non-Important FSM Inputs

The external input conditions control the state transitions of the FSM; these conditions must be mutually exclusive. This seems like we requires a complete set of inputs for each transition and for every state, but this is not the case. In real FSMs, you’ll find that not all external inputs matter in every state. In those cases, we don’t need to include the inputs that don’t matter next to the state transition arrow. If we include the inputs that don’t matter, we make our state diagrams less readable.

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 no limit, though we do need to ensure the conditions governing the transitions are mutually exclusive. Figure 23.48 shows a state diagram fragment with many arrows leaving the “state1” state. The point Figure 23.48 is making is that there is no limit to the number of transition arrows leaving a given state. There are a few key issues to be aware of regarding the transition arrows exiting a given state.

- Your state diagram must account for every possible set of external input conditions for every state. For example, if your FSM has “n” external inputs, every state must necessarily account for $2^n$ possible combinations of those inputs in order to completely specify the FSM. In reality, the $2^n$ is the worst-case scenario; you often find that not all inputs matter for all states.
- You must make sure that all conditions associated with the arrows leaving a given state are mutually exclusive, which means that no two arrows can have the same conditions. If two states had the same set of conditions, the FSM would know the correct transition.
- You can’t assume that an FSM stays in the same state if you don’t explicitly and completely specify all transition arrows leaving the state. This means that if there is a condition where the FSM does not transition to another state, it must indicate this condition with a self-loop, which explicitly states the associated conditions.

For example, what we know from Figure 23.48 is that there must be *at least* three external inputs to this FSM because there are five arrow leaving “state1”. If this FSM only had two external inputs, we could only uniquely represent four transitions. With these there external inputs, we could represent up to eight different arrows leaving “state1”. Since Figure 23.48 only has five transitions but can handle up to eight transitions, some of the arrows in Figure 23.48 must be associated with more than one combination of the three inputs if the state diagram is indeed correct.

FSM are neither magical nor intelligent. FSMs do exactly what you design them to do. This means you must never allow the FSM to “make a decision” on its own. It’s quite easy to not completely specify a FSM and get a good feeling that the FSM is working properly in all of your testing. Inevitably, if you don’t properly specify the FSM, it will fail, and probably fail during a demo of your product to a potential buyer or investor.

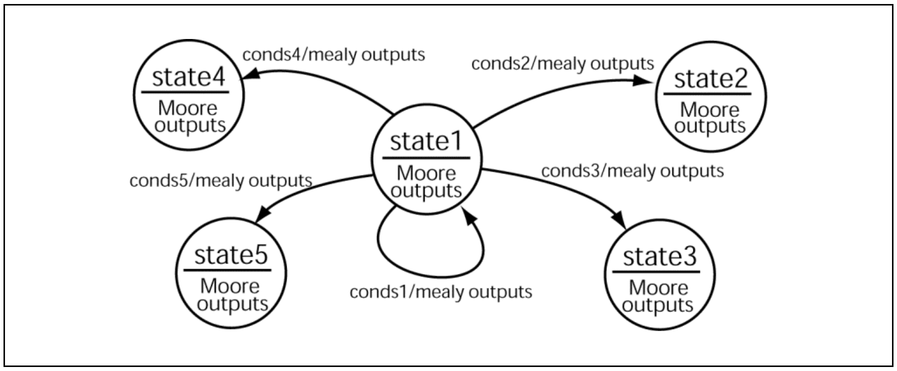


**Figure 23.48: The State Bubble.**

## 23.7 The Final State Diagram Summary

Figure 23.49 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 and use a particular symbology.

- Singly directed arrows represent state transitions
- The FSM has external inputs that govern the state transitions from a given state
- Each transition arrow lists the external inputs that control its transition
- The state bubbles list the Moore outputs since they are only a function of state
- We list Mealy-type outputs with the external inputs (and hence the state transitions) since they are a function of both the present state and the external inputs.

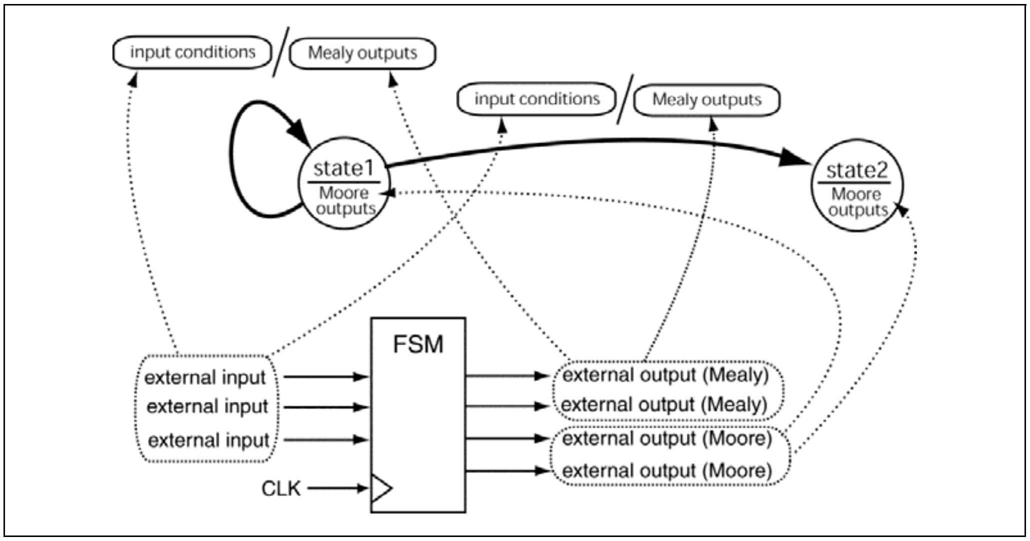


**Figure 23.49: The relation between the state diagram and the high-level FSM.**

The good news is that once you understand FSMs, and traverse the associated learning curve, you'll agree that there is not much to them. Here is everything in a nutshell.

- The heart of the FSM is the state registers; the heartbeat of the FSM is the clock signal that controls the state-to-state transitions of the FSM.
- On each active clock edge, the state of the FSM can transition to the present state (self-loop) or transition to a different state.
- The next state is a function of the present state of the FSM and the external inputs, which form the inputs to the next-state decoder.
- The outputs of the next-state decoder are the inputs to the state registers and thus determine the next state of the FSM.
- The FSM's 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) or a function of both the state variables and the external inputs (Mealy-type).

## 23.8 Chapter Summary

- 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 singly directed arrows 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-to-state transitions are synchronous and thus occur on the active edge of the clock; we show these with an arrow leaving a state and that same arrow entering a state.
- Asynchronous transitions are "somewhere-to-state" and are not synchronized with an active clock edge; we indicate these transitions using an arrow not coming from a state but entering a state.
- 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.
- State diagrams are easier to understand if they omit external inputs and outputs (both Moore & Mealy) from the state diagram under circumstances where they don't matter (when they are don't cares relative to a given state (output) of given transition (external input).
- FSMs can end up in hang states under certain circumstances. Designers can avoid this unwanted condition by designing the FSM to be self-correcting, which is done by directing all unused states back to a valid state in the given FSM.

## 23.9 Chapter Exercises

**1)** Briefly explain the general purpose of a state diagram.

**2)** Briefly explain why do individual states in state diagrams have unique, self-commenting labels.

**3)** Briefly explain why we typically omit lock signals from state diagrams.

**4)** Briefly explain why we label unconditional transfers with some type of "don't care" symbol.

**5)** Briefly explain why PS/NS tables don't include clock signals.

**6)** Briefly explain how we represent asynchronous signals in state diagrams.

**7)** Briefly explain the main function of an FSM's next-state decoder.

**8)** Briefly explain the main function of an FSM's output decoder.

**9)** Briefly explain the main purpose of an FSM's state registers.

**10)** Briefly explain the different between Moore and Mealy-type outputs on FSMs.

**11)** Briefly describe why it is most convenient to not place Mealy-type outputs in the state bubbles.

**12)** Briefly describe with it is most convenient to place Moore-type outputs in the state bubbles.

**13)** Briefly explain what is meant by the term "unused state" in an FSM.

**14)** Briefly explain what is meant by the term "hang state" and how an FSM can end up in a hang state.

**15)** Briefly explain the difference between a hang state and an unused state in an FSM.

**16)** Briefly explain the main strategy behind designing FSM to be self-correcting.

**17)** Briefly explain why some FSM designs inherently do not have hang states.

**18)** Briefly explain why the transition arrows associated with a given FSM state must have conditions that are mutually exclusive.

## 23.10 Design Problems

For each of the following problems:

- Show all the underlying hardware, but minimize your use of hardware, particularly with the state registers
- Completely specify all decoders with an appropriate table
- Assume all inputs and outputs are positive logic unless stated otherwise.
- Provide a state diagram that models the circuit you created to solve the given problem

**1)** Use a FSM to design a synchronous 2-bit binary counter that has an UP and DN (down) input. When the up input is asserted, the counter counts up. When the DN input is asserted, the counter counts down. When both the UP and DN inputs are simultaneously asserted, the counter clears (output "00"). If neither the UP or DN input is asserted, the counter's output values does not change.

**2)** Use an FSM to design an up counter that counts repeatedly in the following sequence: { "0001", "0010", "1000", "1000"…}: . When the counter's UP input is asserted, the counter counts up; otherwise the counter output does not change.

**3)** Use an FSM to design a 3-bit binary up counter that counts. When the counter's UP input is asserted, the counter counts up; otherwise the counter output does not change. The counter also has a RCO (ripple carry out) output that indicates when the counter has reached its maximum count.

**4)** Use an FSM to design a 3-bit binary counter that counts either using 3-bit odd or 3-bit even count values. When the counter's EVN input is asserted, the counter counts up using even number; otherwise the counter counts up using odd number.

**5)** Use an FSM to design a 3-bit binary counter that counts either using 3-bit odd or 3-bit even count values. When the counter's EVN input is asserted, the counter counts up using even number; otherwise, the counter counts up using odd number. The counter has a RCO (ripple carry out) output that indicates when the counter has reached its maximum count. Note that the RCO signal is dependent upon there value of the EVN input, so that the RCO is asserted when the count is 6 and the EVN signal is asserted; otherwise the RCO is asserted when the count is 7 when the EVN signal is not asserted.

**6)** Use an FSM to design an up counter that counts repeatedly in the following sequence: { "100001", "110010", "011000", "001100", "101010"…}: . When the counter's FOR input is asserted, the counter counts up; otherwise, the counter output does not change. Design the FSM to be self-correcting.

**7)** Use an FSM to design a binary up counter that repeatedly counts in the following sequence: {...0, 2, 4, 6, 8, 10…}. When the counter's UP input is asserted, the counter counts up; otherwise, the counter output does not change. The counter also has a RCO (ripple carry out) output that indicates when the counter has reached its maximum count. Design the counter to be self-correcting.

**8)** Use an FSM to design a binary counter that counts in the following sequence: (…2, 17, 23, 11, 30, 2, 17, 23…). This circuit has an active-low asynchronous RST input that forces the count to be '2' when asserted; otherwise allows the circuit to count. Use only simple registers in your design (no LD signal). Make this circuit self-correcting by directing unused states to the state associated with the count value 17.

# 24 FSM Clocking Issues

## 24.1 Chapter Overview

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 at a maximum clock rate.

### Main Chapter Topics

**SEQUENTIAL CIRCUIT ATTRIBUTES:** Many digital circuits contain a system clock. This chapter describes the basic terminology associated with clocking signals.

**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.

### Chapter Acquired Skills

- Be able to describe attributes of clocking signals such as period, frequency, and duty cycle
- Be able to describe physical attributes of digital circuits such as set-up & hold times
- Be able to calculate the maximum clock frequencies of simple sequential circuits

## 24.2 Clocking Waveforms

We consider FSMs to be synchronous circuits because they contain synchronous memory elements. The term synchronous refers to the fact that changes in the state of the FSM's state registers are synchronized to the FSM's active clock edge. This section describes some of the important terms involved in clocking digital circuits.

### 24.2.1 The Period

The most important aspect of clocking waveforms is that the clock signal is most always periodic. We define a periodic clock signal as one that has attributes that remain constant over time; no matter where in time you view the waveform, the clock signal always appears to have the same form. Figure 24.1 shows both a periodic (CLK1) and a non-periodic waveform (CLK2).

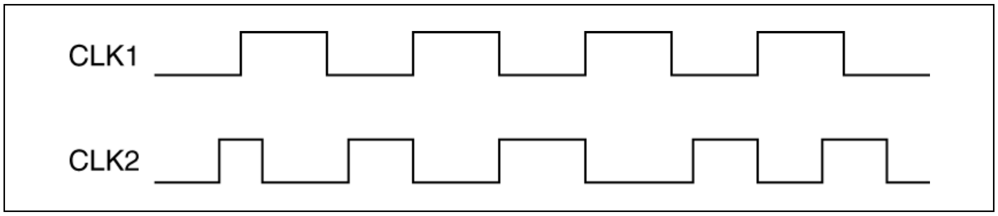


**Figure 24.1: A periodic (CLK1) and non-periodic (CLK2) waveform.**

A periodic waveform is a waveform that repeats itself "every so often", or periodically. The *period* of the waveform indicates the amount of time required for the waveform to repeat itself, which makes "time" the unit of measure associated with the period. Figure 24.2 shows a periodic waveform where we use the "T" to clearly delineated one period. We consider this waveform as periodic because the waveform between (a) and (b) is the same as the waveform between (b) and (c).

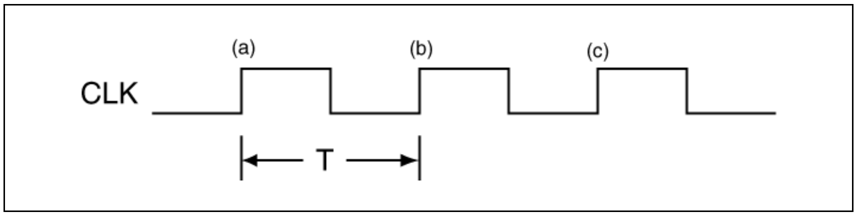


**Figure 24.2: Timing diagram showing a timespan we consider the period.**

### 24.2.2 The Frequency

The *frequency* of the waveform represents the number of times a signal repeats itself over a given amount of time. This definition is to general so we usually refine it somewhat to make it more usable. The "span of time" we're usually interested in is one second (1s). Using this one-second time slot simplifies the translation of period to frequency.

Period and frequency have a reciprocal relationship when we consider the amount of time is one second; Figure 24.3 shows these relationships. The units for frequency are Hertz, or Hz for short. The term Hertz is technically defined as the number of *cycles per second*, which refers to the number of times a given signal repeats itself in one second. The term Hertz has units of **s$^{-1}$**, which underscores its reciprocal relationship to the period.

| | |
|---|---|
| $Period = T = \dfrac{1}{frequency} = (frequency)^{-1}$<br>Units: time (seconds) | $frequency = \dfrac{1}{Period} = \dfrac{1}{T} = (T)^{-1}$<br>Units: Hz (seconds)$^{-1}$ |
| **(a)** | **(b)** |

**Figure 24.3: The calculations and units for Period and Frequency.**

---

**Example 24.1: Waveform Frequency Calculation**

A given waveform has a 40ns period. What is the frequency of this waveform?

**Solution:** Taking the reciprocal of the period provides the frequency the calculation in Figure 24.4 shows.

$$frequency = \frac{1}{40ns} = \frac{1}{40x10^{-9}} = 25x10^{6}\,Hz = 25MHz$$

**Figure 24.4: The solution to Example 24.1.**

**Example 24.2: Waveform Period Calculation**

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.

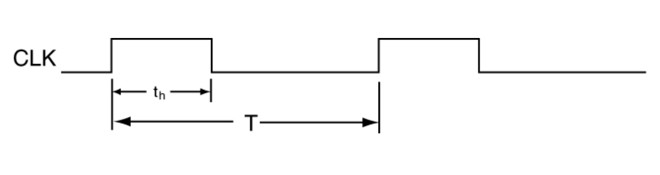
$$Period = T = \frac{1}{50MHz} = \frac{1}{50x10^{6}\,s^{-1}} = 20x10^{-6}\,s = 20\mu s$$

**Figure 24.5: The solution to Example 24.2.**

### 24.2.3 Duty Cycle

All the periodic waveforms we've dealt with up to now were symmetrical, which means that the signal was high and low for the same percentage of time. Sometimes the clock signal high and low times are not equivalent; in these cases, we use the term *duty cycle* to describe the waveform.

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 24.6(a) shows the official equation for duty cycle. Because the duty cycle refers to a ratio, there are no units associated with duty cycle.



$$dutycycle = {}^{t_h}\!/_{T}$$

Units: none

**(a)** **(b)**

**Figure 24.6: Duty cycle calculations and units.**

**Example 24.3: Duty Cycle Calculation**

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.

## 24.3 Practical Synchronous Circuit Clocking

Most of our FSM discussion thus far dealt with the notion of idealized storage elements, which allowed us to focus on the basic functioning of the devices. We now must take into account a few timing considerations in order for our sequential circuits to work properly with increasing clock speeds. Many factors prevent digital circuits from working properly, but our focus is on two major timing considerations. In addition to propagation delays, register have issues associated with the synchronous nature of the circuit that we need to consider.

### 24.3.1 Setup and Hold Times

One of the consequences of properly clocking synchronous circuits is that you need to pay attention to the device's non-clock control inputs near the active edge of the clock. More specifically, control inputs need to remain stable for a given amount of time both before and after the active clock edge. We refer to the amount of time the control input needs to remain stable before the active clock edge as the *setup time* and the amount of time the control input needs to remain stable after the active clock edge as the *hold time*.

Figure 24.7(a) and Figure 24.7(b) show the setup and hold times associated with a rising-edge and falling-edge triggered synchronous device, 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 and the hold time. If the control input were to change during these times, the output, and thus, the state of the flip-flop would be indeterminate. If your circuit *violates* a setup or hold time, your device may become *metastable*[1], which means the output of the device is neither high nor low; it is somewhere in-between and it may stay there for an extended length of time.

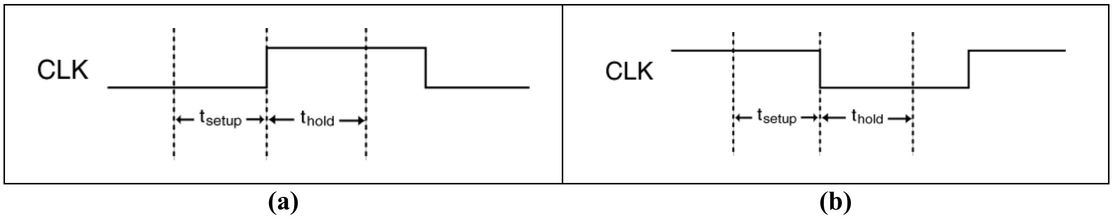


**Figure 24.7: Setup and hold times for rising edge (a) and falling edge (b) triggered flip-flops.**

Setup and hold times are associated with many different types of digital circuits, and the idea is always the same: keep a signal stable for a given amount of time both before and after some critical clock edge. We consider a few practical aspects of a sequential circuit that 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 a while and then it hits you: *you violated a setup and/or hold time*.

## 24.4 Maximum FSM Clock Frequencies

In this modern age, faster is generally associated with better; the same is true for digital circuits. Namely, for a given 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 circuit's sequential elements can operate at without violating things such as setup and hold times?

Figure 24.8 shows a model of an FSM. Recall that there are propagation delays associated all digital circuits; there are also issues of setup and hold times associated with sequential logic. From the diagram of Figure 24.8, you should sense that the amount of circuitry in the various boxes lowers the maximum rate at which the FSM can operate, with the idea that signal requires more time to propagate through larger circuits than smaller circuits. Attributes in each of the submodules in Figure 24.8 affect the maximum clock frequency of the circuit.

For this discussion, we assume that the Output Decoder does not affect the maximum clocking frequency of the circuit. What does matter is the propagation delay though the Next State Decoder, the setup times associated

[1] And yet again, a digital design word makes it out of digital design land. We often use the word metastable to describe people who are unpredictable; the type you'll do best to avoid. Academic administrators, for example.

with the state registers, and some combination of the state register's hold time and/or the propagation delay through the register. These items require time: as the time accumulates, the period for the clock signal becomes greater, and hence, the maximum clock frequency becomes lower.

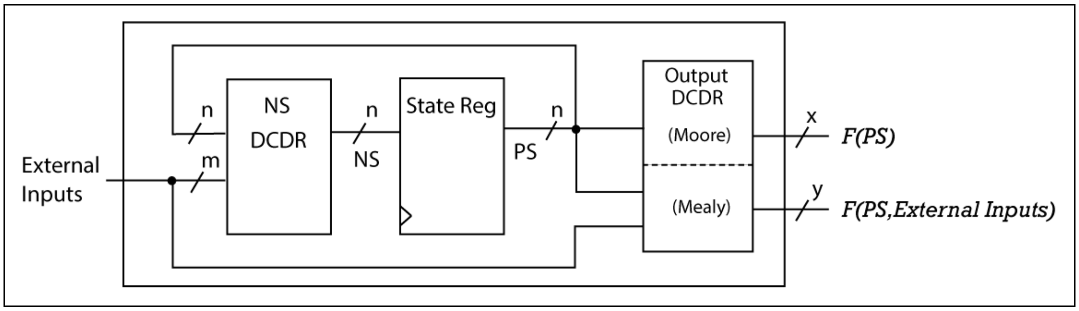


**Figure 24.8: The generic FSM model.**

In order to simplify the analysis of FSM circuits, we also make some other assumptions about this circuit. For a given flip-flop, we have both a hold time and a propagation delay time that we need to deal with. For these problems, we assume that the propagation delay for the state register 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 and setup times associated with the state registers. Figure 24.9 provides a visual representation of these attributes.

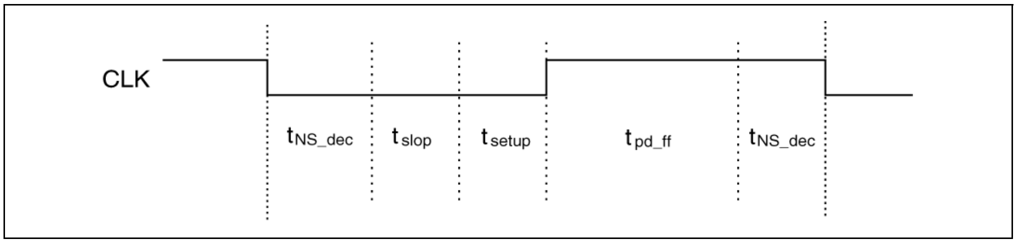


**Figure 24.9: The set-up & hold times for a rising and falling clock edge.**

Figure 24.9 shows four time slices that we need to consider when calculating maximum clock frequencies. Despite the fact that the timing diagram shows it 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 diagram to the same portion of the waveform on the left side of the diagram. Another factor we 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[2]. We use these four values to calculate the minimum period as Figure 24.10 shows. Figure 24.10 shows the minimum period is the reciprocal of the maximum clock frequency.

$$T_{min} = t_{ns\_dec} + t_{slop} + t_{setup} + t_{pd\_ff}$$

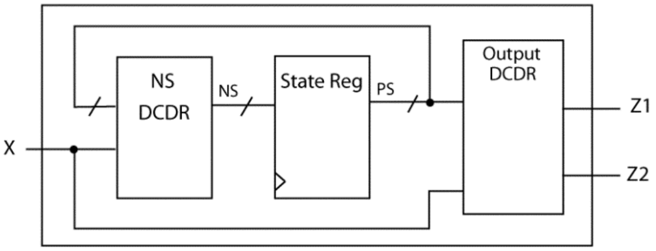
$$\text{Frequency}_{max} = (T_{min})^{-1}$$

**Figure 24.10: Official calculations for minimum period and maximum clock frequency.**

[2] These factors would include ambient temperature variations and physical variations in the device itself.

**Example 24.4: Maximum Clock Frequency Calculation for FSM**

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, and the next state decoder has a worst-case propagation delay of 18ns. 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 outputs drive a circuit that is not sensitive to the maximum clock frequency.



**Solution:** We don't need to worry about the X input because the problem states that the X input value is stable. The problem also stated that the outputs are another item we don't need to worry about. What we need to do for this problem is total up the various delays in order to find the maximum clock frequency we can drive this FSM at and still have it operate properly. The safety margin of 12ns makes of the $t_{slop}$ value. Figure 24.11 shows the final solution for this example.

$$T_{min} = t_{ns\_dec} + t_{slop} + t_{setup} + t_{pd\_ff}$$

$$T_{min} = 18ns + 12ns + 10ns + 13ns$$

$$Frequency_{max} = (T_{min})^{-1} = (53ns)^{-1} = 18.9MHz$$

**Figure 24.11: The calculations: plug and chug.**

## 24.5 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 can also described by the frequency, which is defined as the reciprocal of the period.
- Periodic waveforms are also described by their duty cycles, which are 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.

## 24.6 Chapter Exercises

**1)** Briefly describe the units associated with the following three clock signal attributes:

a) Periodic Signal

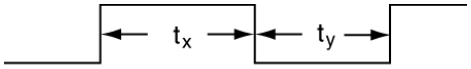
b) Period

c) Frequency

d) Duty Cycle

**2)** Breifly describe whether non-periodic clock signals can have duty cycles.

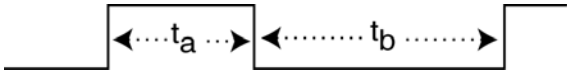
**3)** For the system clock signal displayed below with $t_x$=30ns and $t_y$=25ns, find the period, frequency, and duty cycle of the waveform. (1ns = 1x$10^{-9}$ seconds)



**4)** 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 = 1x$10^{-9}$ seconds).

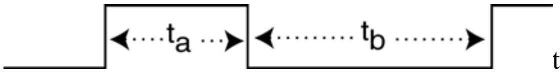
**5)** A system clock if running at 50M Hertz. What amount of time is the signal high if the system clock has a 40% duty cycle? (1 M Hertz = 1x$10^6$ Hertz)

**6)** The following clock waveform is in a low state for 80% of the period. Find the duty cycle, period, and frequency (its OK to only setup the frequency calculation).



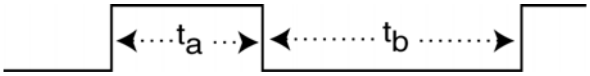
$t_b$ = 20ns

**7)** The following clock waveform is in a *high* state for a 40% of the period. Find the duty cycle, period, and frequency (it's OK to only setup the frequency calculation).



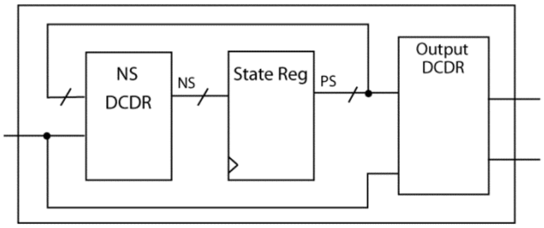
$t_a$ = 20ns

**8)** The following clock waveform is in a *low* state for a 20% of the period. Find the duty cycle, period, and frequency (it's OK to only setup the frequency calculation). The diagram is not drawn to scale.



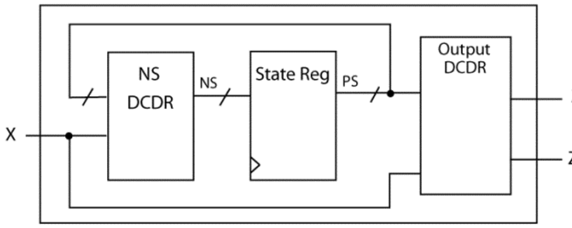
$t_b$ = 60ns

**9)** 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 Z outputs drive a circuit that is not sensitive to the maximum clock frequency. Assume the X input is stable. Use the listed circuit parameters for this problem:

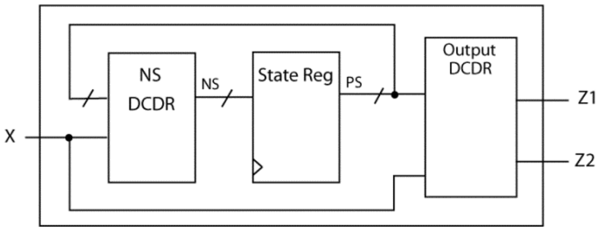
| | |
|---|---|
|   NS<br>DCDR<br>NS<br>State Reg<br>PS<br>Output DCDR<br>X<br>Z1<br>Z2 | *flip-flop propagation delay:* **20ns**<br>*NS DCDR propagation delay:* **10ns**<br>*flip-flop set-up time:* **6ns**<br>*flip-flop hold time:* **7ns** |

**10)** For the previous problem, you now need to add a different 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?

**11)** 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 Z outputs drive 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:

| | |
|---|---|
|   NS<br>DCDR<br>NS<br>State Reg<br>PS<br>Output DCDR<br>X<br>Z1<br>Z2 | *flip-flop propagation delay:* **17ns**<br>*NS DCDR propagation delay:* **9ns**<br>*flip-flop set-up time:* **8ns**<br>*flip-flop hold time:* **7ns** |

**12)** 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 Z outputs drive a circuit that is not sensitive to the maximum clock frequency. Assume the X input is stable. Use the listed circuit parameters for this problem:

| | |
|---|---|
|   NS<br>DCDR<br>NS<br>State Reg<br>PS<br>Output DCDR<br>X<br>Z1<br>Z2 | *flip-flop propagation delay:* **20ns**<br>*NS DCDR propagation delay:* **15ns**<br>*flip-flop set-up time:* **5ns**<br>*flip-flop hold time:* **7ns** |

# 25 Introductory Controller-Based FSM Design

## 25.1 Introduction

Our previous work with FSMs has primarily been involved with implementing FSM using relatively low-level hardware modules. We're moving towards using FSMs as controller circuits, but we're first need to gather more experience generating state diagrams to solve problems. We know the mechanics of FSMs; now we change our focus to state diagram generation.

Here are the important truths regarding modern FSM design: 1) we rarely implement FSMs using low-level hardware, and 2) generating the state diagram represents most of the engineering associated with designing FSMs. Although anyone can implement a FSM from a given state diagram, it requires a complete understanding of all aspects of FSMs, all aspects of digital design, and a complete understanding of 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 associated timing diagram renders you ready to handle any control problem. Our initial approach to designing FSMs was to design both the underlying logic and the associated state diagram. In this chapter, we move away from implementing FSMs with lower-level logic and concentrate more on generating the state diagrams.

**Main Chapter Topics**

**HISTORICAL PERSPECTIVE OF FSMS AS CONTROLLERS**: The chapter provides a brief history of FSMs in the context of controlling digital circuits.

**FSM PROBLEM SOLVING:** This chapter introduces basic state diagram generation in the context of sequence detectors. Sequence detectors provided relatively simple problems to understand which allows you to focus your efforts on generating the associated state diagram.

**Chapter Acquired Skills**

- Be able to solve simple control problems using FSMs as circuit controllers. This set of problems includes basic signal synthesis problems and sequence detectors.
- Describe the history of using digital circuits to control digital circuits.

## 25.2 FSM Historical Overview

The world progressed nicely for bajillions of years without having the concept of finite state machines. 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 path that has led us to where we are today (though a few details are missing).

In relatively recent history, digital stuff (computers and things) started happening[1]. All the new digital stuff required some digital circuitry to control it; FSMs were the logical option. You could have used a computer to

[1] 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.

control a computer, but computers were big and expensive, and had names like "HAL"[2]. The problem with software-based control was that it increased the complexity of the software and required extra program memory[3]. In order to deal with these issues, we typically farmed out the software control requirements of projects to hardware devices, such as FSMs. This approach worked because the required hardware was not prohibitively expensive.

Later, integrated circuits (ICs) started taking over. There were already many ICs out there, but all of a sudden, there were many more ICs out there. These new ICs provided more complex functionality, which meant that some of the control functions handled by FSMs were built into the ICs. There were also ICs dedicated to controlling specific devices (such as memory and interrupt controllers), which became a requirement because control requirements were growing in complexity.

As time went on, microcontrollers (MCUs) started becoming prevalent[4]. The MCUs were more versatile than FSMs in that we could program MCUs to do anything while the FSMs were hardcoded to performing one task. This meant that hardware devices could now essentially be under program control (programs run on the MCUs) rather than under hardware control (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, particularly as the number of control tasks in a given system increase.

As more time went on, Programmable Logic Devices (PLDs) such as FPGAs and CPLDs hit the market. This meant that you could use PLDs to design hardware to handle the control issues. Though PLDs are hardware devices, they are flexible because they are reprogrammable. PLDs were powerful and inexpensive, which meant transferring control from MCUs to FSMs was not overly costly. Thus, the advent of relatively inexpensive but powerful PLDs allows the offloading of control tasks from the system software to external hardware.

One of the downsides of MCUs is their basic limitation is the number of pins they need to interface to the outside world. The pin count generally relates to the cost of the MCU also: the more pins on your MCU, the more you're going to pay for it. 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 architecture becomes more complicated (slower and more error-prone) based on the number of tasks it must control.

The good news is that FSMs are not quite dead; people still use them quite often to avoid some of the hassles created by complicating the software associated with the controlling circuits using MCUs. In addition, not all control problems are well suited for MCUs; some control requirements are too small to warrant farming off to an MCU. 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.

The question arises: How do I use a FSM to control something? The answer is that you must understand the following:

- Understand how the FSM operates in terms of the underlying hardware (such as the state registers, output decoding logic, next state decoding logic)
- 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 outputs, Moore outputs, self-loops, strike, spare, etc.
- Understand the symbology used to describe the FSM; namely, the state diagram symbology

[2] "Sorry Dave, I can't do that".

[3] Keep in mind that back in these days, memory was much more expensive than it was today.

[4] They had actually been around for a while, but they were now less expensive. More importantly, the development environments (primarily PC-based) and associated CAD tools were significantly less expensive.

- Understand how to implement the FSM, either with flip-flops and discrete logic components or high-level modeling with some type of PLD.

Figure 25.1 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 from the FSM that control external components) and the status signals (inputs to the FSM that allow the FSM to know what and how to control the external components). In the FSM model of Figure 25.1, the external inputs act as the status signals from the circuit the FSM is controlling, while the external outputs act to control the components external to the FSM. A clock input keeps things flowing evenly.

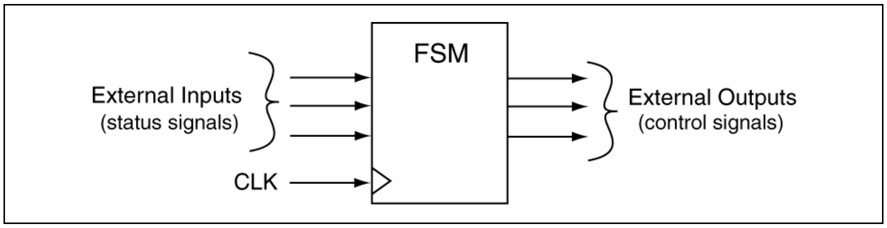


**Figure 25.1: The general view of a FSM used as a controller circuit.**

## 25.3 Digital Design Overview

This section gives an overview of digital design, including Digital Design Foundation Modeling. Much of this information was presented in previous chapters; we include it here for completeness.

Digital design is the process where you create a digital circuit to solve a given problem. There are many approaches you can use to solve given problems, designing a digital logic circuit is only one of them. A given digital design solves problems by having the outputs react to the inputs in a manner such that it solves the given problem. There are two basic types of digital logic circuits:

- **Combinatorial Circuits**: circuit outputs are a function of the circuit's inputs. These circuits can't store information.
- **Sequential Circuits:** circuit outputs are a function of the sequence of the circuit's inputs. These circuits can store information.

The two basic tenets of digital logic are:

- **Digital logic circuits are hierarchical**: We can describe a digital circuit at various levels; the level at which we describe digital logic is generally the one that allows us to transfer as much useful information as quickly as possible. Abstracting digital designs to higher levels aids in understanding and designing circuits.
- **Digital logic circuits are decomposable into a set of standard digital modules**: Although there are many ways to describe digital circuits, we strive to make the descriptions an aggregate compilation of standard digital circuits in able to help us understand the circuits.

### 25.3.1 DDFM Overview

The focus of DDFM is to present digital design in a simple and organized manner, which facilitates and expedites learning the subject matter. These are the main tenets of DDFM:

- The main purpose of digital design is to solve problems using digital circuits
- We can best describe digital circuits in a modular and hierarchical manner
- Digital circuits are a set of digital modules that exchange information under the control of some entity

- We perform digital circuit design in a ***structured***[5] manner, meaning that we can model ***any*** digital circuit using a relatively small subset of digital modules, which we refer to as the ***digital design foundation modules***. Each foundation module performs a relatively small set of simple operations.
- We present the digital design foundation modules at a high-level by modeling the modules in terms of their data, control, and status signals, which allows us to use the modules in designs, while not requiring us to initially understand underlying implementation details.
- We classify the digital design foundation modules as either "controlled" or "controller" circuits
- We consider there to be four approaches to controlling a digital circuit:
    5) **NO CONTROL** (no flexibility in circuit behavior)
    6) **INTERNAL CONTROL** (controlling circuits using internal signals)
    7) **EXTERNAL CONTROL** (controlling circuits with devices such as buttons, switches, etc.)
    8) **CIRCUIT CONTROL** (controlling circuits using FSM or computer).
- We categorize digital design approaches into three categories:
    4) **BRUTE FORCE DESIGN (BFD)**
    5) **ITERATIVE MODULAR DESIGN (IMD)**
    6) **MODULAR DESIGN (MD)**

Figure 1.2 shows a digital circuit containing various modules. We define a digital circuit as a controlled interaction between a set of sequential and combinatorial circuits (the two types of digital circuits). Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it solves the given problem. Figure 1.2 also shows the modularity (the various modules) and the hierarchical (modules within modules, or boxes within boxes) characteristics of digital circuits.

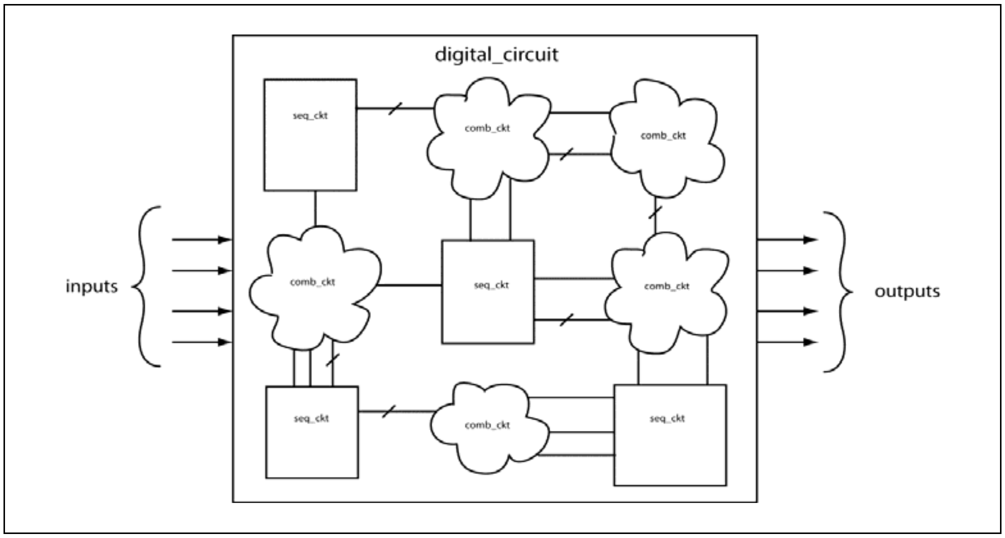


**Figure 25.2: A generic digital circuit containing a set of digital modules.**

Figure 1.3(a) shows the standard approach to modeling digital circuits, where all digital circuit signals were classified as either inputs or outputs. Figure 1.3(b) and Figure 1.3(c) shows how DDFM further classifies inputs

[5] This is an analogy to structured computer program design

and outputs by first separating digital modules into “controlled circuits” and “controller circuits”. Figure 1.3(b) shows that we further classify the inputs to controlled circuits as either “data” or “control” and classify the outputs of controlled circuits as either “data” or “status”. This means the various circuit elements in Figure 1.3(b) are able to 1) pass data from their data inputs to their data outputs under the direction of the “control” inputs, and, 2) describe characteristics of the data transfers using the status outputs. Similarly, the status outputs of the controlled circuit form the status inputs of the controller circuit. The controller circuit of Figure 1.3(c) inputs the status signals of controlled circuits and manages the controlled circuits by outputting the appropriate control signals to control the controlled circuits.

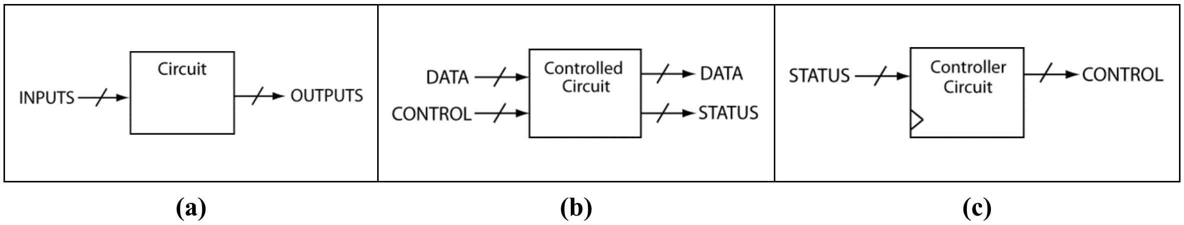


**Figure 25.3: Old digital circuit model (a); models for controlled (b) and controller circuits (c).**

The DDFM paradigm allows us to model all digital circuits as a controller that controls a set of modules. We then consider the solution to any digital design problem as a matter of using a controller to properly control the dataflow through a set of controllable modules. Figure 1.4 shows an example of many circuit modules controlled by a controller circuit; the controller circuit is either a finite state machine (FSM) or some type of computer control, such as a microcontroller. Figure 1.4 includes three different module shapes showing that controllable modules can either be combinatorial or sequential circuits, as well as off-the-shelf computer peripherals.

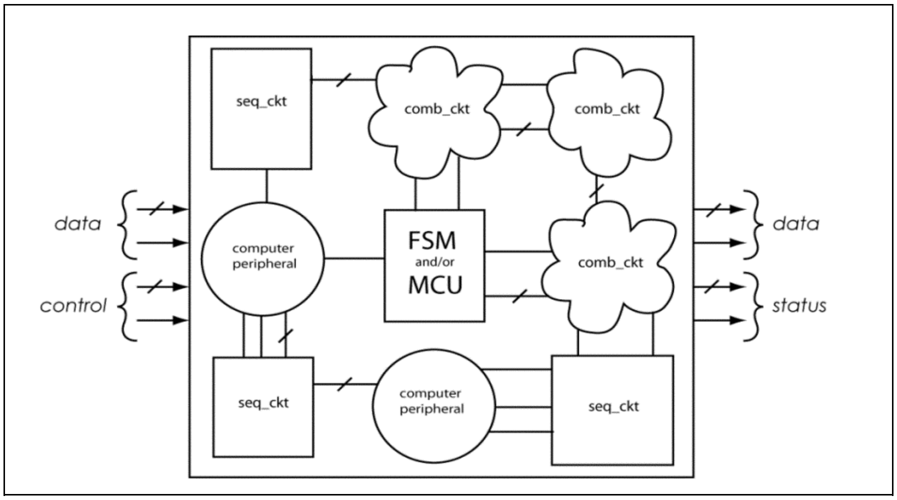


**Figure 25.4: Our unifying digital circuit model.**

### 25.3.2 The Three Approaches to Digital Design

Part of DDFM includes categorizing digital design into three different approaches, which we discuss in more detail later in the text. With some combination of these three approaches, you can create any digital circuit.

**BRUTE FORCE DESIGN (BFD):** Our first approach to digital design. Although simple, its simplicity limits its practicality in non-trivial designs.

**ITERATIVE MODULAR DESIGN (IMD):** Our second approach to digital design. Although IMD removes some of the limitations of BFD, it is only applicable to a few of circuits.

**MODULAR DESIGN (MD):** Our final and most powerful approach to digital design, and is thus where this text expends most of its efforts.

Figure 25.5(a) shows the basic model of a digital logic circuit; we characterize the signals that the outside world sees as either inputs or outputs. Because we need to control the flow of data through the digital circuit, we must more specifically define the inputs and outputs of a basic digital circuit module. Figure 25.5 (b) shows that we further classify the inputs as either “data” or “control” and classify the outputs as either “data” or “status”. This means the various circuit elements in Figure 25.5 (b) are able to 1) pass data from their data inputs to their data outputs under the direction of the “control” inputs and, 2) output characteristics of the data transfers using the status outputs.

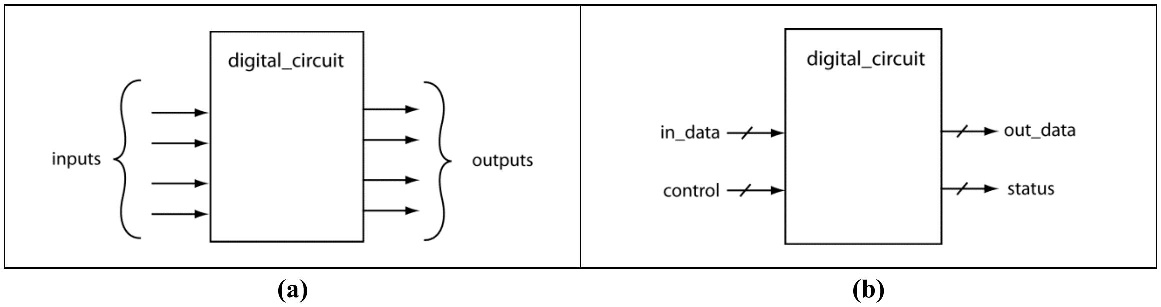


**Figure 25.5: Models for a basic logic circuit (a), and a more refined basic digital logic circuit (b).**

Something must control the flow of data through the generic digital circuit. We therefore must have some other entity that interprets the status signal outputs of the circuit modules and issues control signals to those circuit modules. For this beginning digital design text, we consider the controlling circuit to be an FSM. Figure 25.6 shows a generic model of an FSM. The FSM simply interprets the status signal outputs from various digital modules and then outputs the appropriate control signals that are the various digital modules use as control inputs. Other interesting characteristics to note include:

- FSMs generally do not have data inputs and data outputs. You can design FSMs with data inputs and outputs, but they tend to be klunky and non-generic.
- The FSM is a sequential circuit because it has the ability to store bits. The FSM only stores bits to represent the “state” of the FSM, which it does in its “state variables”.
- The underlying model of the FSM includes three primary elements: 1) the next state decoder, 2) the output decoder, and, 3) the state registers. The next state decoder is a combinatorial circuit that decides the next state based on the given state and status inputs. The output decoder is a combinatorial circuit that generates control outputs based on either state only (Moore-type) or state and status inputs (Mealy-type). Figure 25.7 shows a model for an FSM with both Moore and Mealy-type outputs.

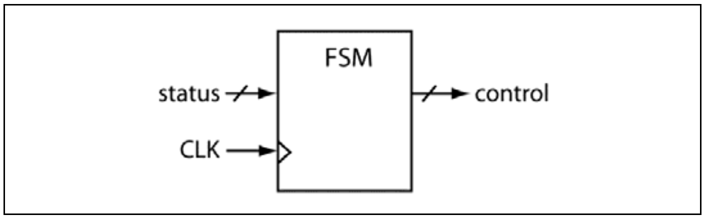


**Figure 25.6: A black box model of a FSM.**

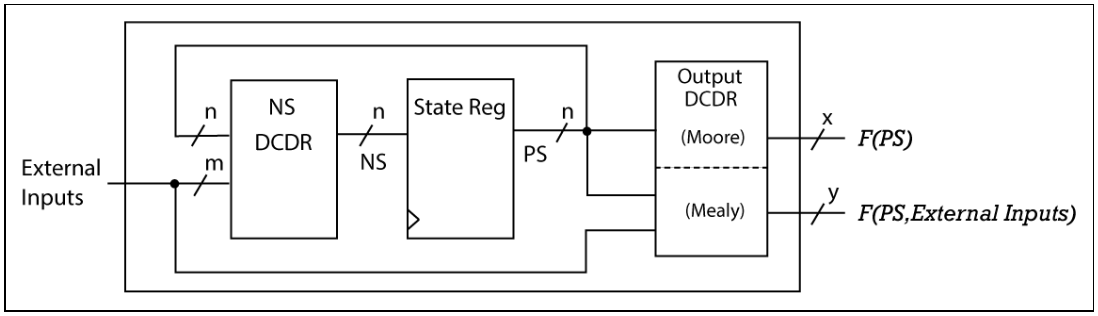


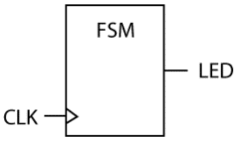
**Figure 25.7: A black box model showing the component parts of an FSM.**

## 25.4 Attack of the Blinking LEDs

Digital designers often use FSMs to aid in the synthesis of signals. This means you can use FSMs to generate output signals with specific properties that would not be easy to obtain using any other digital design techniques. These FSMs are useful handy because they are relatively straightforward to design for novice FSMers, and because they present simple techniques that you often draw upon when using FSMs to solve design problems.

**Example 25.1: FSM Design #1: Blinking LED with 50% Duty Cycle**

Use an FSM to blink a single **LED** with a 50% duty cycle and at half the clock frequency of the FSM's clock. Provide a state diagram for your solution.



**Solution**: There are three main design issues associated with this problem. First, we need to blink an **LED**. Second, we need to blink that **LED** at a 50% duty cycle. Third, the blink frequency needs to be half that of the FSM frequency. The FSM handles each of these requirements quite naturally.

Figure 25.18 shows two versions of a state diagram for our solution. These two solutions are equivalent, and represent to different methods of representing the outputs of state diagram. Here is the fun stuff:

- A Moore-type output nicely implements our solution. This makes sense, particularly since the FSM has only one input: the clock.
- The state diagram has two states: an "on" state and an "off" state. We list the **LED** output as either on or off in the given state. We list the **LED** output value directly in Figure 25.18(a) by specifying the output directly, and somewhat indirectly in Figure 25.18(b) by showing the LED label with either an overbar when the LED is off, and no overbar when the **LED** is on.
- All transitions are unconditional
- The state diagram has two states, both with unconditional transitions. This means that the FSM always transition from one state to the other, which then means that the **LED** spends half the time on and the other half of time off. This provides the 50% duty cycle as requested.

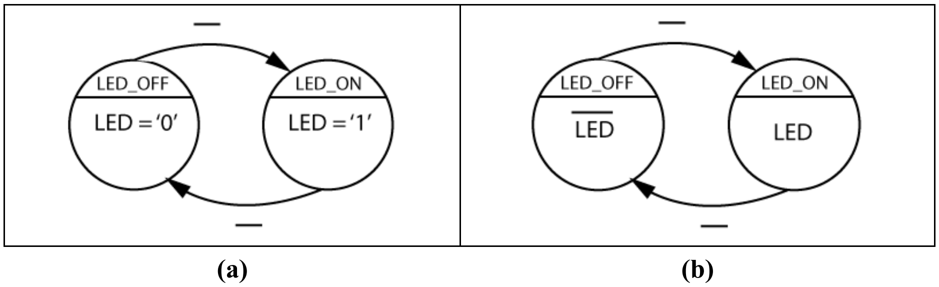


**Figure 25.8: The different but functionally equivalent state diagrams for our solution.**

Figure 25.9 shows an example timing diagram associated with our solution. Here is some cool stuff to realize regarding this timing diagram.

- The timing diagram arbitrarily starts in the LED_OFF state.
- On each active clock edge (the rising clock edge) the FSM changes state, as seen in the STATE line.
- The LED changes value every active clock edge. As a result, the frequency of the **LED** blinking is half the frequency of the FSM's clock signal.

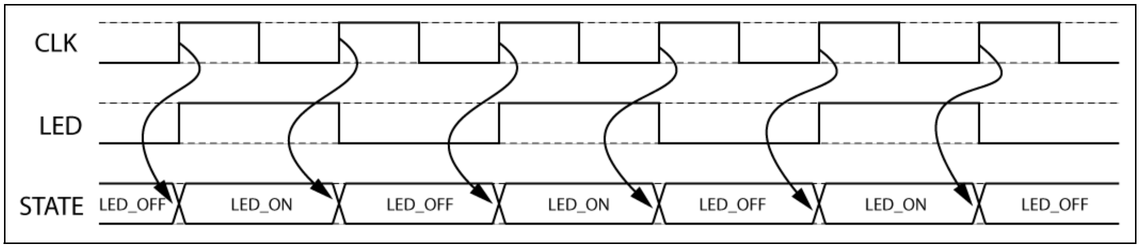
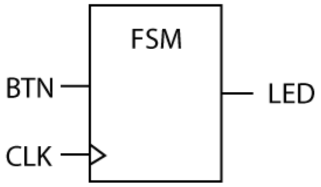


**Figure 25.9: An example timing diagram for our solution.**

**Example 25.2: FSM Design #2: Blinking LED with Control Features**

Use an FSM to blink a single **LED** with a 50% duty cycle at half the clock frequency of the FSM's clock when a button is not pressed. The circuit's button input, when asserted (positive logic), prevents the **LED** from changing status (off vs. on). Provide a state diagram for your solution.



**Solution**: This problem is similar to the previous problem but we added an external input signal that controls the operation of the FSM. When the **BTN** input is asserted, it prevents the FSM from changing states. The way we prevent an FSM from changing states is to not allow it to leave the present state, which means the FSM transitions back to the state it is currently in. Figure 25.10 shows the state diagram for this example.

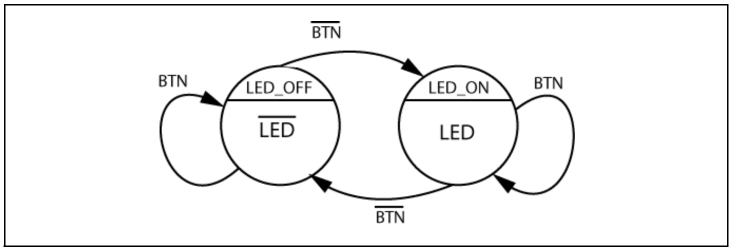


**Figure 25.10: The state diagram for this example.**

Figure 25.11 shows an example timing diagram for this problem. Here are some special things to note regarding this timing diagram.

- The FSM does not change state on the first two active clock edges because the **BTN** input is asserted.
- The FSM toggles on the third rising clock edge as the **BTN** input is not asserted. The FSM also toggles on the fifth and sixth clock edge.
- The LED output follows the **LED** assignment in the state diagram. The **LED** is off (**LED**=’0’) when the FSM is in the LED_OFF state; the **LED** is on in the LED_ON state.
- The arrows indicate that the CLK input (the active edge) and the BTN input combine to cause the state change, and then the state change causes a change in the state of the LED output. The arrows indicate that the clock edge and BTN input caused changes in both the LED and state.

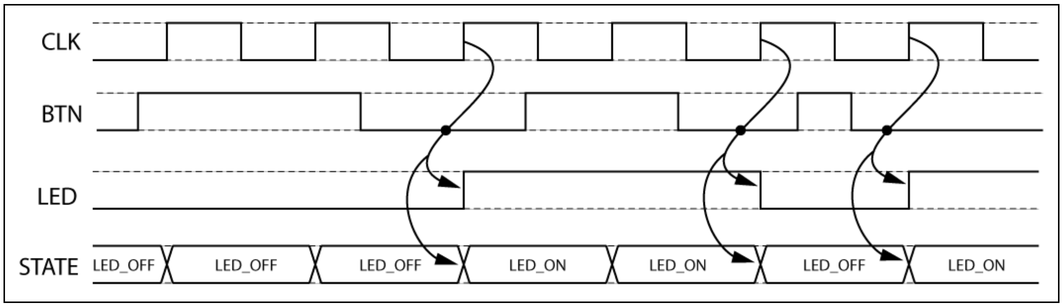
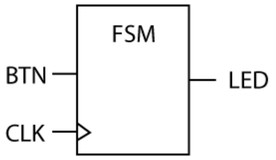


**Figure 25.11: An example timing diagram for our solution.**

### Example 25.3: FSM Design #3: Blinking LED with Control Features

Use an FSM to blink a single **LED** with a 50% duty cycle at half the clock frequency of the FSM’s clock when the circuit’s button is not pressed. This circuit’s **BTN** input, when asserted (positive logic), prevents the **LED** from changing status if the **LED** is off. Provide a state diagram for your solution.



**Solution**: In the previous example, when the button was asserted, the FSM could not change state. When the **BTN** signal is asserted in this problem, it causes the FSM to hold state if the FSM is currently in the LED_OFF state, but does not cause the FSM to hold the state if the FSM is in the LED_ON state. One other thing to note:

- The "-" listed on the transition arrow from the LED_ON state represents an unconditional transition. The conditions associated with the don't care symbol are officially: **!BTN** + **BTN**, which is always true.

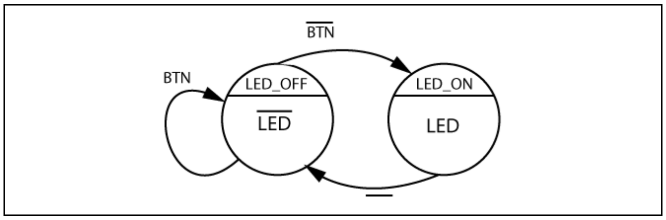


**Figure 25.12: The associated state diagram for our solution.**

Figure 25.13 shows an example timing diagram for the solution. Here are the so-called highlights listed in the order of rising clock edges. .

1) The state does not change because the **BTN** input is asserted.
2) The state changes because the **BTN** input is no longer asserted.
3) The **BTN** input is asserted, but the FSM always transitions from the LED_ON state to the LED_OFF state on the next active clock edge.
4) The **BTN** input is not asserted so the FSM changes state.
5) The FSM always changes state when in the LED_ON state.
6) The FSM changes states because the **BTN** input is not asserted.

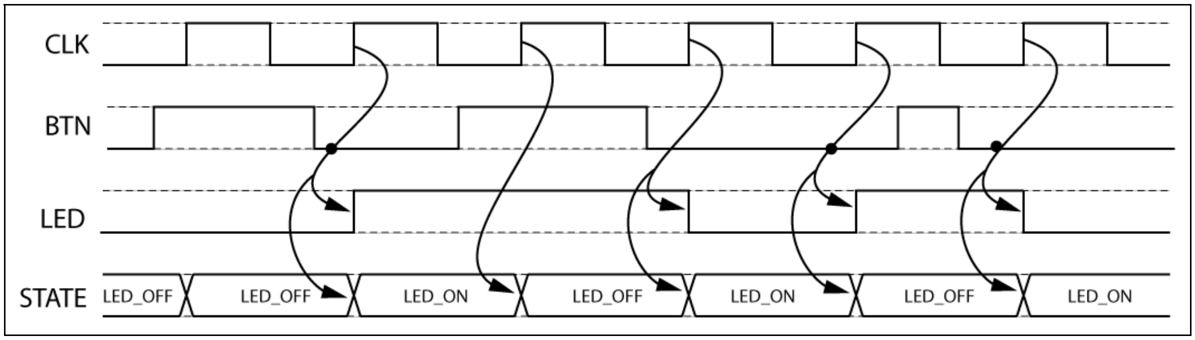
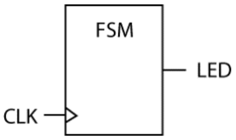


**Figure 25.13: An example timing diagram for our solution.**

**Example 25.4: FSM Design #4: Blinking LED with Special Duty Cycle**

Use an FSM to blink a single **LED** with a 33.3% duty cycle at a frequency of 60MHz. State the frequency of **CLK** signal for this problem. Provide a state diagram for your solution.



**Solution**: This problem is a similar to the first blinking **LED** example, but now we have something other than a 50% duty-cycle. The problem states that we need a 1/3 duty-cycle. Clock signals are generally periodic with 50% duty-cycles. Thus, what we need to do for this problem is to turn on the **LED** for 1/3 of the time, and off for

2/3 of the time. The best way to do with for FSM problems is to add more states to the state diagram to get the **LED** output timing we're looking for. Figure 25.14 shows the state diagram for our solution.

- The state diagram has three states; all state transitions are unconditional
- The **LED** is off in two states and on in one state; this provides the desired 33.3% duty cycle.

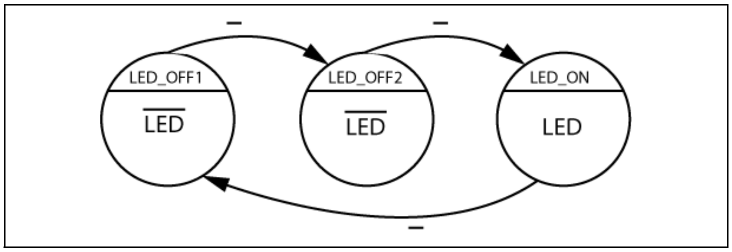


**Figure 25.14: The state diagram associated with our solution.**

The problem asks up to state a CLK frequency in order to blink the **LED** at 60MHz frequency. Mathematically speaking, the period associated with the **LED** blink rate is three times as long as the period of the system clock. Since frequency and period have a reciprocal relationship, the frequency of the **CLK** signal must be three times the frequency of the **LED** blink rate. The desired clock frequency is thus 180MHz.

Figure 25.15 shows a timing diagram associated with our solution. The 33.3% duty-cycle is evident by examining the **LED** output. The FSM transitions from one state to another on every rising clock edge.

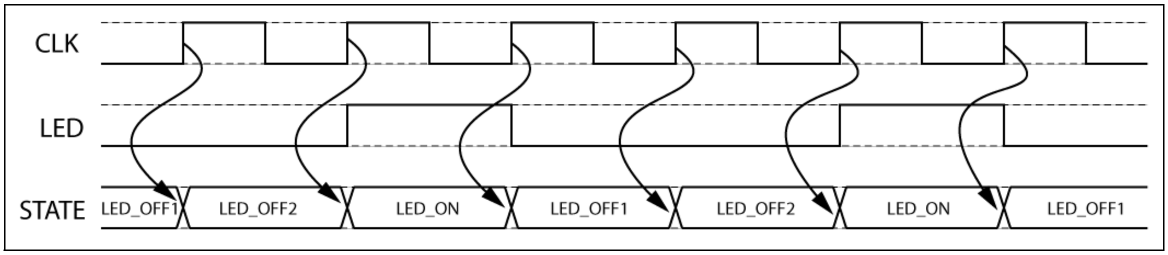


**Figure 25.15: An example timing diagram for our solution.**

**Example 25.5: Maximum Value Displayer**

Design a circuit that finds the largest of four 8-bt unsigned binary inputs after a button is pressed. The maximum value stays on the output until the circuit detects another button press. The circuit also ensures the button is press is released before it is able to find another maximum value. Minimize your use of hardware in your design; don't use more than one comparator in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

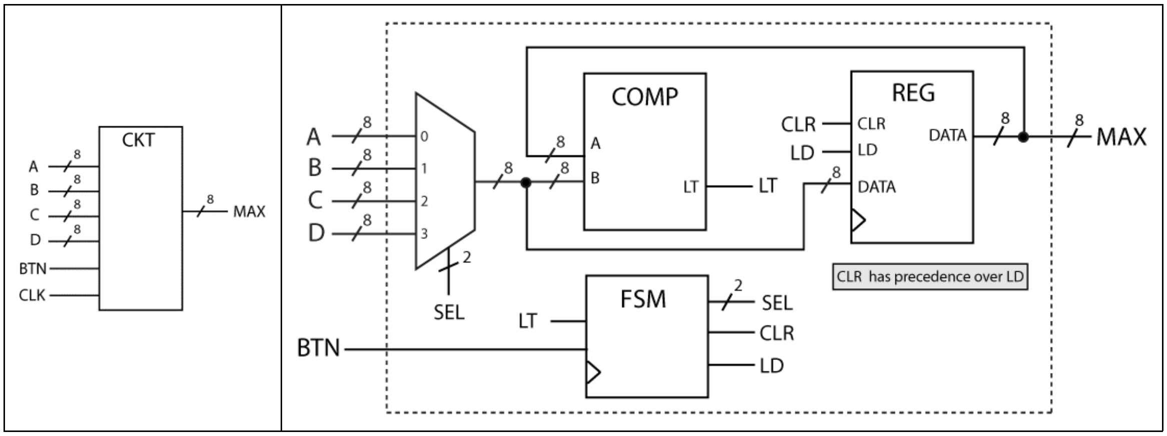
**Solution**: This problem is similar to previous problems we did when before we started working with sequential circuits. The main constraint in this problem is that we use no more than one comparator, which essentially forces us to design a circuit controlled by an FSM. Figure 25.16(a) show the result of the first step in this solution, which is a top-level BBD.

The next step is to create an inventory of the modules this design requires. Here is the thought process:

- Any circuit that establishes maximum and minimum values of a set of data requires a comparator.
- The output of the problem also needs to be persistent, which means we need a register to hold the final value.
- Since we only have one comparator, we need to “select” which input values we are comparing, so the circuit also needs a MUX.
- The circuit requires some type of control, so the circuit needs a FSM.

Figure 25.16(b) shows the lower-level BBD for our solution. Here are some important items to note:

- The diagram only routes data signals in order to make the diagram more readable. We understand that the non-routed signals are status and control signals, which are inputs to and outputs of the FSM, respectively. We clearly  label the FSM’s inputs and outputs on both the “unrouted” signals to clarify their connectivity.
- The FSM has two status inputs: 1) the button, and 2) the **LT** output of the comparator. The FSM uses these signals to determine the state transitions in the associated state diagram.
- The FSM has three outputs: 1) the **SEL**, 2) the **CLR**, and 3) the **LD** signals. The FSM uses these signals to control the other circuit elements. The control outputs of the FSM do in fact connect to the various control inputs of the other modules.
- We must state the **CLR** control input to the register has precedence over the **LD** input.
- The comparator compares a value external to the circuit to the current output of the register. This circuit then continually updates the current maximum value as it examines the other input values.



**(a)** **(b)**

**Figure 25.16: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 25.17 shows the state diagram that models the circuit’s FSM. Here are some of the finer points of the state diagram.

- In the “wait” state, the FSM waits for a button press to start the process. When the circuit receives a button press, the circuit first clears the output register. Clearing the register effectively makes the first comparison with zero, which is the smallest possible value for an 8-bit unsigned binary number. If the circuit determined the minimum value, we would then first initially load this register with all 1’s.

- We modeled the clearing action of the circuit as a Mealy-type output; we could model it as a Moore-type output, but that would have required an extra state in the state diagram.
- The four "C_x" states use the **SEL** input of the MUX to step through the four comparisons. The FSM's **SEL** output differs for each of these four states.
- It is possible to connect the LT output of the comparator directly to the **LD** input of the counter. We did not do this because we want to be able to disable to the register's **LD** control when we find and display the max value. If we made the direct connection, we would have no way of preventing that the maximum value on the output from changing during the two wait states.
- Once the circuit finds the maximum value, it then waits for a button lift in the "wait_btn" state. We do this is because circuits like this one typically operate faster than you can press and lift the button. The self-loop in the "wait_btn" state ensures that the circuit only finds one maximum values per button press[6].

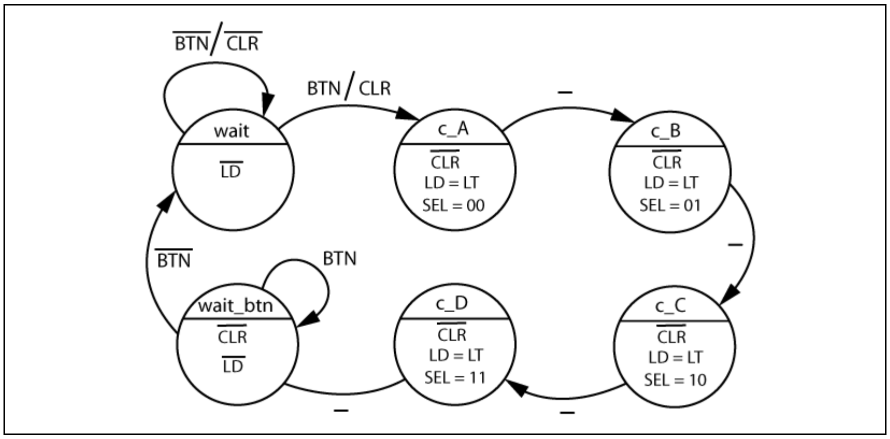


**Figure 25.17: The state diagram associated with this example.**

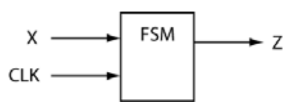
This circuit has both external and circuit control. The BTN input is the external control that serves as a status input to the FSM. The control signals on the MUX and the register are outputs of the FSM, which is circuit control.

## 25.5 FSMs as Sequence Detectors

Designing sequence detectors is one of the earliest topics in FSM design  because they are highly instructive and spiritually enriching while not being overly complicated. Designing sequence detectors gives you practice generating different flavors of state diagrams under limited external input control and with few external outputs.

Figure 25.18(a) shows the general form of a simple sequence detector. There is one external input X and one external output Z; there is also a clock input because this is an FSM. This particular example monitors whether the sequence "101" to appears on the X input. Figure 25.18(b) shows a sample input sequence for the X input and the resulting outputs for two different ways of examining the input sequence. The data in Figure 25.18(b) is the data present when the active clock edge appears on the FSM (each column represents one clock edge).

[6] There are also switch-bounce issues that we're not dealing with here.



```
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 →
```

(a) (b)

**Figure 25.18: A black box diagram (a), and sample inputs/outputs for finding "101" sequence (b).**

Figure 25.18(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 "resets" ("resetting") after finding the correct sequence. In this context, resetting refers to the ability of the output to reuse past inputs regardless of whether they were part of a previously successful[7] sequence or not. In the case where there is no reset, the Z output is a '1' anytime the previous three X inputs[8] are the 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, the FSM can't reuse values from other successfully detected sequences can't in a new sequence.

The resetting and non-resetting flavors allow for two types of problems. Additionally, because we are designing FSM, we can model the Z output as either a Mealy or a Moore-type output. Sequence detector problems can have one of four solutions based on the type of FSM output (Mealy or Moore) and whether the machine resets or not after detecting the correct sequence.

The following diagrams works through the example of Figure 25.18(b) thus producing a result in the four different methods ("Mealy" vs. "Moore" and "resetting" vs. "non-resetting"). We list less detail in some diagrams due to the similarities in the development process. The next four figures show the solutions that represent all the possible conditions for the reset/no reset and Mealy/Moore options.

[7] Meaning that the sequence led to the finding of desired sequence.

[8] Once again, the most correct wording is that '1' was present on the X input when an active edge clock edge arrived. For problems such as these, we generally constrain the X input to only changing no more than once per clock period.

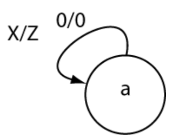
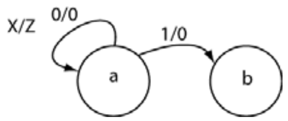
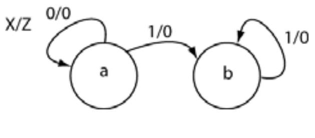
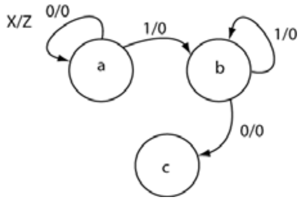
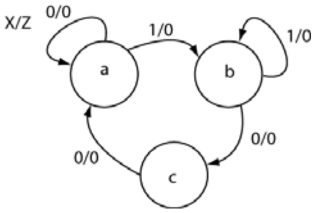
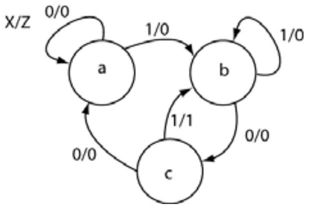
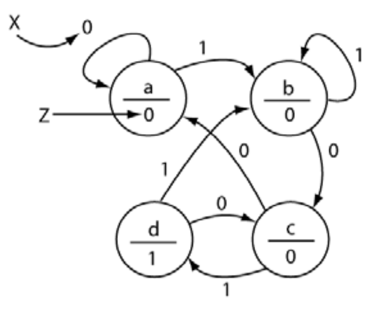
| | |
|---|---|
|  | The best place to start is before the FSM sees any correct values in the sequence. The transition is where the FSM finds the undesired input of ‘0’, so it stays in the state looking for the desired input of ‘1’. Since FSM has not found the correct sequence, the **Z** output is a ‘0’. We arbitrarily assign the state as “a”. |
|  | Each state bubble must account for two arrows leaving the state representing the possible values of the **X** input. A ‘1’ on the **X** input causes a transition to the new state. When in state (b), then you know you’ve seen the first value of the sequence. There are two arrows leaving state (a); conditions associated with those transitions are mutually exclusive. |
|  | As long as the FSM receives a ‘1’ on the **X** input in state (b), we stay in that state as the self-loop arrow indicates. If the FSM receives more ‘1s’ in state (b), there is no reason to exit this state. No matter how many‘1’s you receive in state (b), you won’t leave, the state since ‘1’ is the first value in the desired sequence and ‘0’ is the second value. |
|  | 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. 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). Anytime you receive two contiguous ‘0’s, you must start again because two zeros are not part of the sequence. |
|   X/Z<br>0/0<br>1/0<br>1/0<br>a<br>b<br>1/1<br>0/0<br>0/0<br>c | If the FSM receives a ‘1’ in state (c), two things happen. First, you found the desired “101” sequence and the output **Z** is set to ‘1’. Second, because the FSM 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 could not transition back to (b) if the FSM was to reset after finding the correct sequence. |

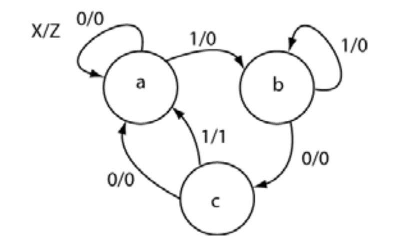
**Figure 25.19: Generation of a state diagram that detects a “101” sequence without resetting.**



The Moore state diagram for 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 have more states than a Mealy-type FSM implementing the same functionality.

**(a)** **(b)**

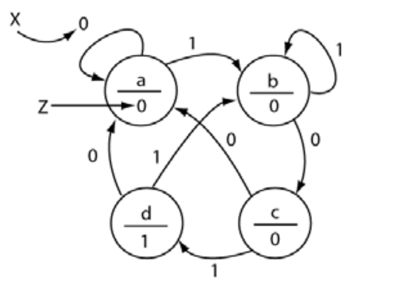
**Figure 25.20: State diagram (a) and explanation (b) for Moore-output (no reset) for "101" sequence.**



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 25.21: 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 25.20. 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 25.21.

**(a)** **(b)**

**Figure 25.22: State diagram (a) and explanation (b) for Moore-output (with reset) for "101" sequence.**

### 25.5.1 Sequence Detector Post-Mortem

Even though you should never simply "follow rules" when you're solving sequence detector problems, here are a few "suggestions" to chew on. As you do more of these designs, you'll develop your own style and collect your own set of tricks that make these problems easier.

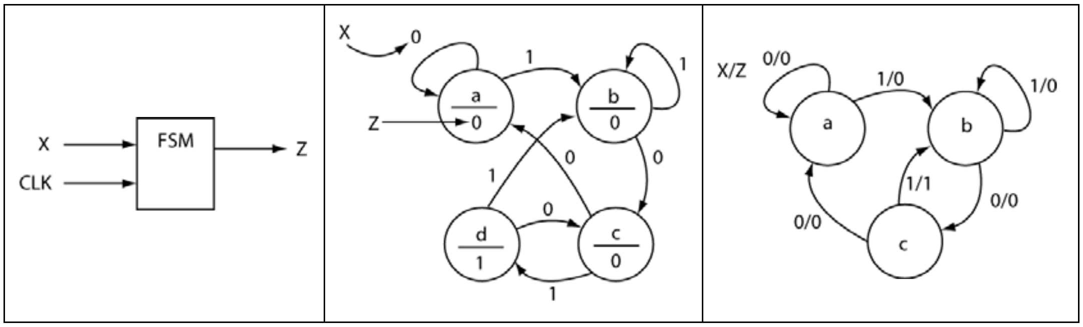
1) Construct a sample input to clarify problem description.
2) Construct a path for the correct sequence first; then go back and fill missing transitions.
3) Try to add new arrows to existing states before adding new states.
4) Verify each state has one exit path for each value of the input variable (two arrow leaving)
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. Keep in mind here is that you can easily generate your own sequence detector practice problems.

## 25.6 Timing Diagrams: The Mealy & Moore-Type Outputs

The final step in developing a true understanding of FSMs is to understand the relationship between the state diagram and the timing diagram. Sequence detector problems provide simple examples for understanding the timing differences between the FSM's Mealy and Moore-type outputs.

The FSM we previously worked with asserted the Z output when the sequence "101" appeared on the X input. Figure 25.23(a) provides a block diagram of this FSM; Figure 25.23(b) and Figure 25.23(c) show the state diagrams for the non-resetting Moore-type and Mealy-type FSMs for this problem, respectively.



(a) (b) (c)

**Figure 25.23: The block diagram of the sequence detector FSM (a), the associated Moore-type output approach (b), and the associated Mealy-type output approach (c).**

The difference between the Mealy and Moore-type state diagram is evident in that the Mealy-type has one less state than the Moore-type. This highlights the functional difference between FSMs with Mealy-type or Moore-type outputs. Here are the two main ramifications.

1) A FSM implemented with Mealy-type outputs generally have fewer states than a functionally equivalent FSM with a Moore-type output. This is because Mealy-type outputs can change in the middle of a state (because they are a function of the FSM's external input) while Moore-type outputs can only change when the state changes. The FSM with Moore-type outputs must have extra states to generate the correct outputs in states that have true Mealy-type outputs.
2) Mealy-type outputs can change with an external input changes, which means that Mealy-type outputs can potentially "react" faster (change output values) because Moore-type outputs need to wait until the next clock edge to change the output.

The main difference between these two diagrams is in 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. We had to do it this way because in the Moore-type state diagram, we required a separate state to indicate when the FSM detects the final bit in the sequence.

For the case of the FSM with a Moore-type output, the output Z is asserted for the duration of the state (state **d** Figure 25.23(b)). 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** & **d**) as in the Moore-type state diagram. 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**). 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 waits for the next clock edge to transition to state **d** where the Moore-type output is ‘1’.

Figure 25.24 shows two example timing diagrams associated with the state diagrams of Figure 25.23(b) and Figure 25.23(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 always synchronized with state changes. The arrows in the top timing diagram of Figure 25.24 show this synchronization.

The lower timing diagram of Figure 25.24 shows that output of the FSM with Mealy-type outputs. In the timing diagram for the Mealy-type outputs, the output **Z** changes at times other than at the same time as the rising edge of the clock. In state **c** of the low timing diagram of Figure 25.24, the **Z** input follows the change in the **X** input. Figure 25.23(c) show this characteristic by the two state transitions from state **c** in the Mealy-type state diagram. The transition associated with the **X**=0 input has an associated output of ‘0’ while the transition associated the condition that **X**=1 has an output of ‘1’. The state diagram thus indicates that the **Z** output has two possible values in state **c**, as the output is a function of the **X** input as well as the state.

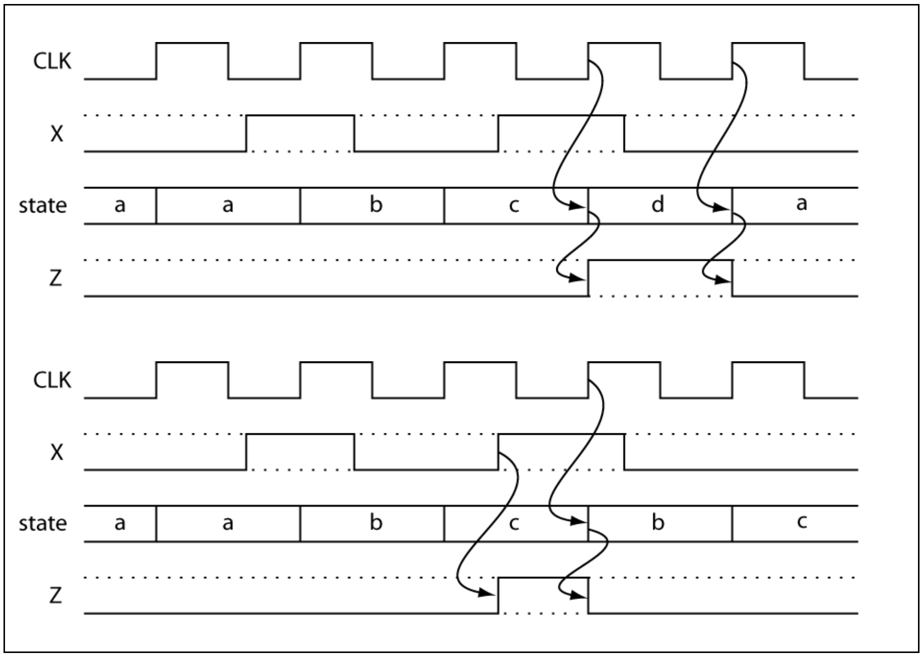
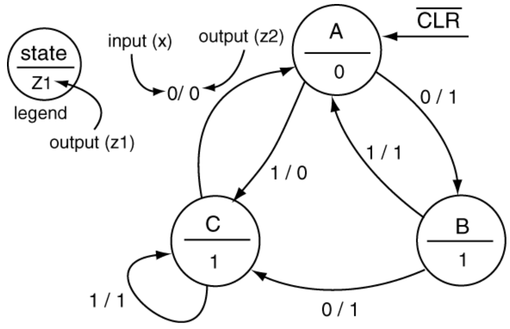
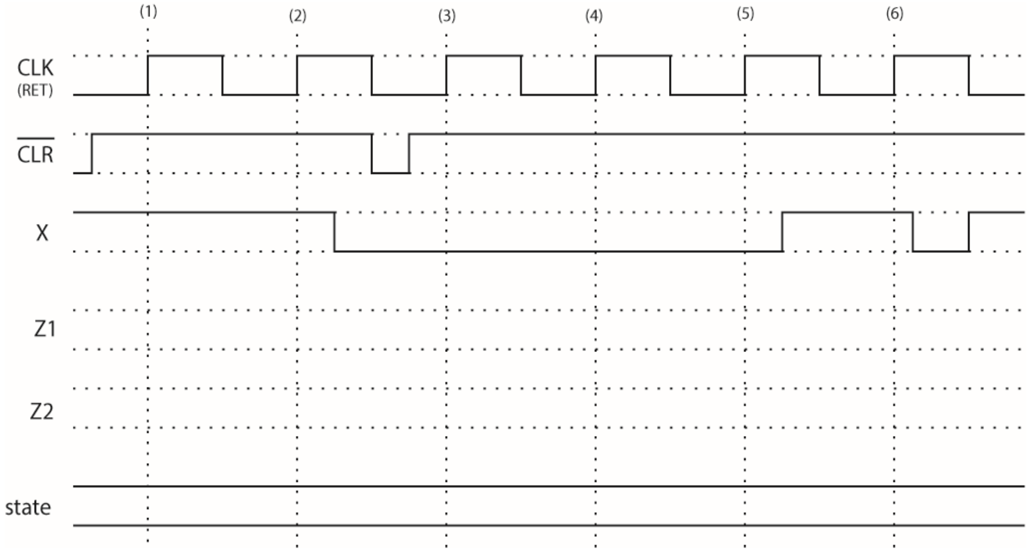


**Figure 25.24: The timing diagram associated with the FSM with Moore-type outputs (top) and the Mealy-type outputs (bottom). Figure 25.23(b) shows the state diagram for the Moore-type FSM while Figure 25.23(c) shows the state diagram for the Mealy-type FSM.**

**Example 25.6: FSM Timing**

Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and outputs **Z** by filling in the "state" and "**Z**" lines in the timing diagram. Assume all setup and hold times are met and 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.





**Solution**: Figure 25.25 shows the solution to this problem. Here is the list of important things to note:

- The **CLR** input is initially asserted, which places the FSM in a state A according to the asynchronous input in the state diagram.
- On the first rising clock edge, the FSM transitions to state C because of the asserted **X** input, which we indicate with an arrow emanating from the rising clock edge combined with the dot on the **X** input. In state C, **Z1** is always a '1' (as it's a Moore-type output). **Z2** is a Mealy-type output, but it is always a '1 in state C because the **X** input remains asserted while in state C for this time period.
- On the second rising clock edge, the FSM does not change state because the **X** input is a '1'.
- In the time interval labeled (3), the **CLR** input asserts, which forces a transition back to the state A. The state does not change when **CLR** unasserted in that same cycle. This change in state causes changes in the **Z1** & **Z2** outputs accordingly.

- For the other states, the **Z1** output is always follows the state as it is a Moore-type output. We can characterize the **Z2** output by examining interval (6) & (7), and by referring to the state diagram. The **Z2** output is an inversion of the **X** input in state A (interval (6)); the **Z2** output is the same as the **X** input in state C (interval (7)).

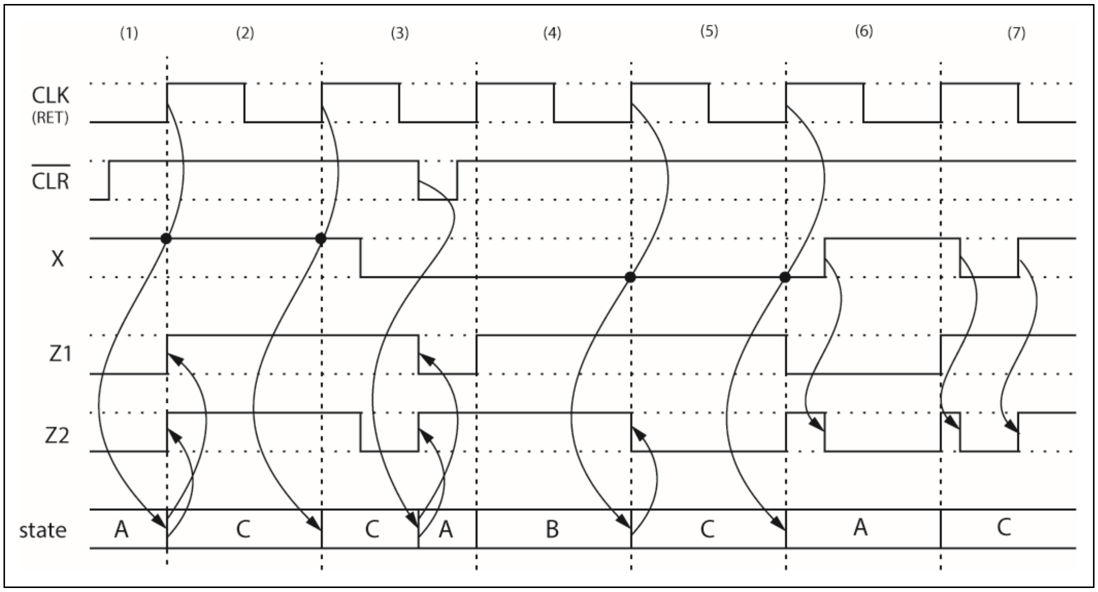


**Figure 25.25: The timing diagram solution.**

## 25.7 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.
- FSMs can be used to synthesize specific output signals, particularly to provide signals with duty cycles other than 50%.
- 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.
- Sequence detector problems can be one of four different types based on the notions of Mealy vs. Moore machines and "resetting" vs. "non-resetting". The notion of resetting implies that the FSM can't "reuse" values of previously found sequence in the search for the next sequence while "non-resetting" can reuse bits from a previously found sequence.

## 25.8 Chapter Exercises

**1)** What is the minimum number of states in a state diagram you would need to obtain a 7/17 duty cycle on an external blinking LED? Briefly explain the reasoning behind your answer.

**2)** Briefly describe an application where a sequence detector would be useful.

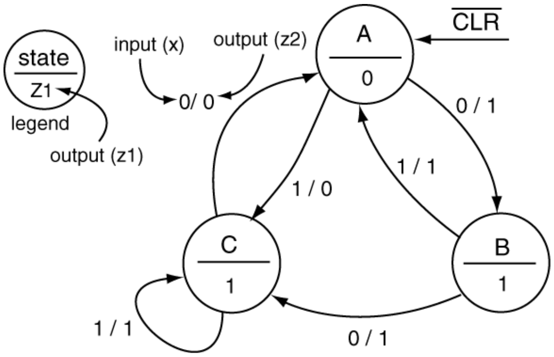
**3)** Briefly describe the operational difference between a FSM with a Moore-type output and a functionally equivalent FSM with a Mealy-type output. Consider both FSMs to have equivalent clock frequencies.

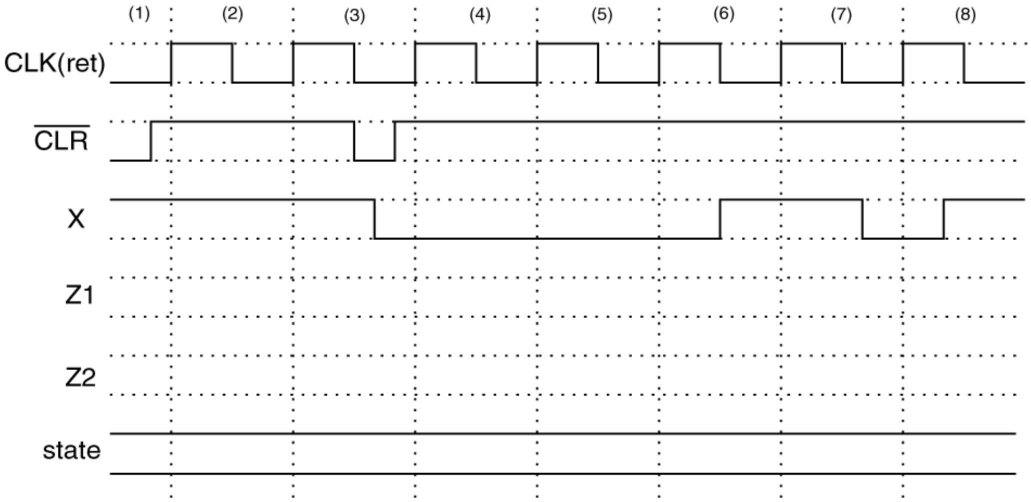
**4)** Briefly describe two advantages to using a FSM exclusively Mealy-type outputs over an functionally equivalent FSM with exclusively Moore-type outputs.

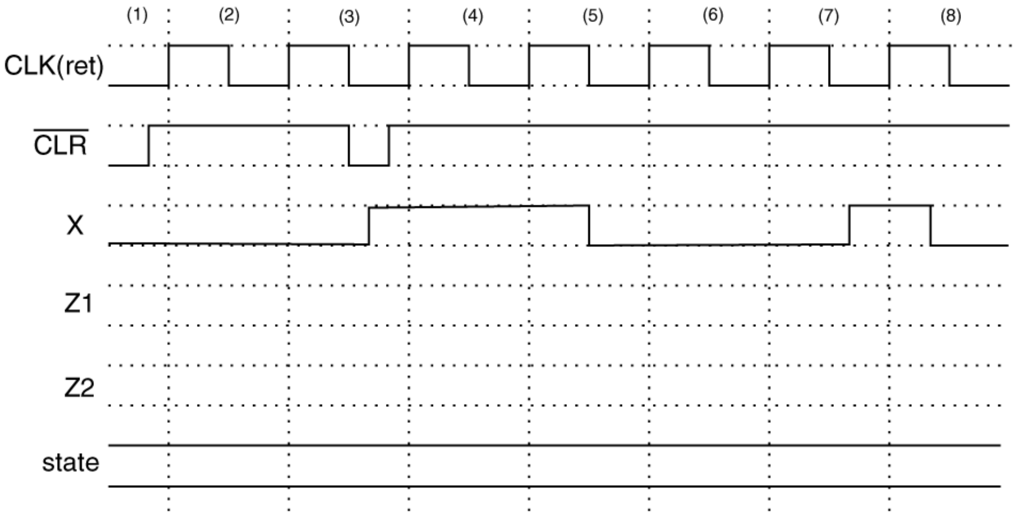
**5)** We often consider FSMs as "reacting". In the context of controlling a digital circuit, briefly describe what we mean by "reacting". Be sure to describe what the FSM is reacting to and what the ramifications of these reactions do in a holistic view of the FSM.

**6)** Briefly explain why it is that FSMs with Mealy-type outputs can react faster than an equivalent FSM with Moore-type outputs.

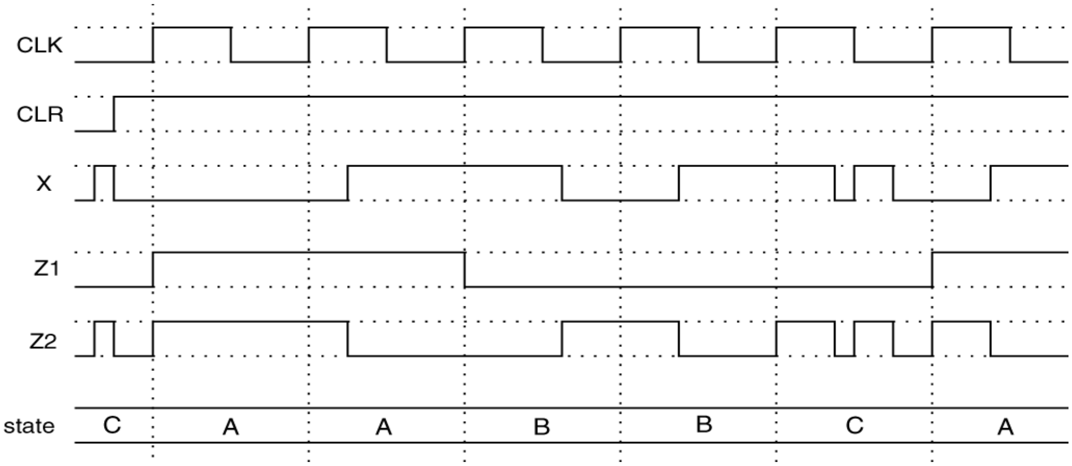
**7)** Use the following state diagram to complete the two timing diagram provided below. Show how the inputs affect the state transitions and outputs Z by filling in the "state" and "Z" lines in the timing diagram. Assume all setup and hold times are met and that propagation delay times are negligible. Assume state transitions occur on the rising edge of the clock signal. Assume CLR is an asynchronous, active low input.



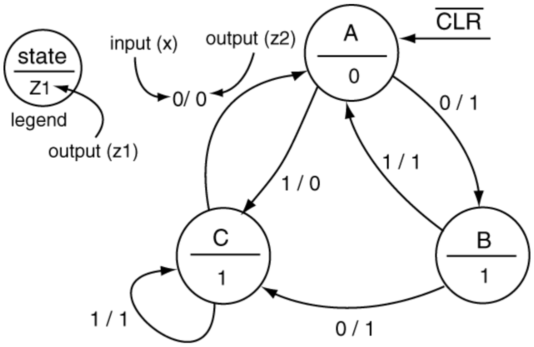


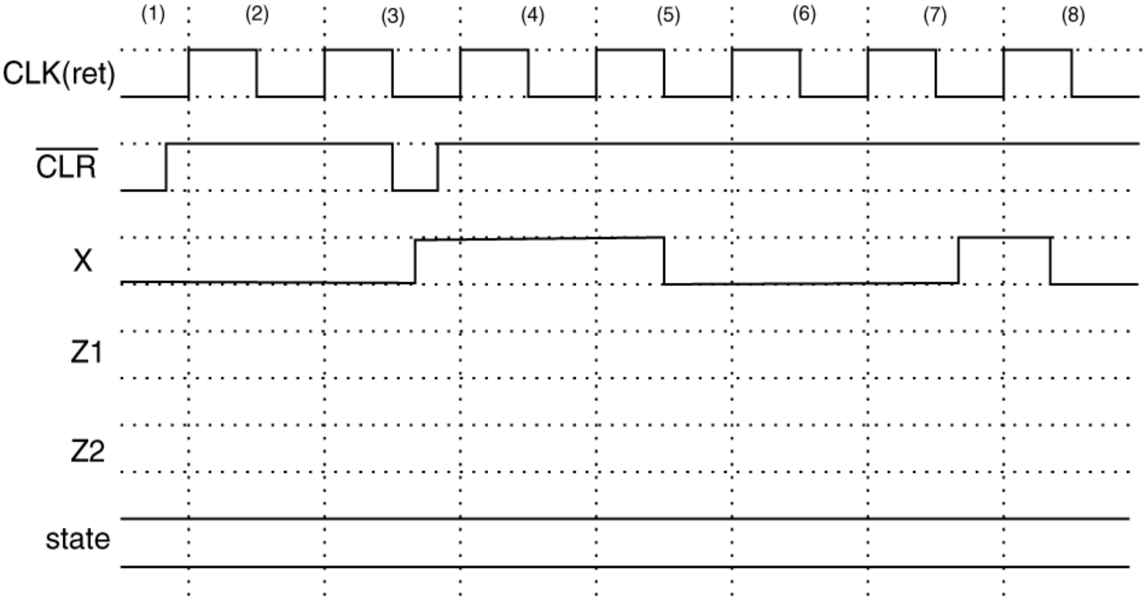


**8)** The following timing diagram completely specifies an FSM. Use the following timing diagram generate the state diagram that would generate the listed timing diagram. For this problem, assume the CLR input to be an asynchronous active low input that places the FSM into the appropriate state. Assume all setup and hold times have been met and that propagation delay times are negligible. Assume state transitions occur on the rising edge of the clock signal.



**9)** Use the following state diagram to complete the timing diagram provided below. Show how the inputs affect the state transitions and outputs Z by filling in the "state" and "Z" lines in the timing diagram. Assume all setup and hold times have been met and 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.


state
Z1
legend
output (z1)
input (x)
output (z2)
0/ 0
A
0
CLR
0 / 1
1 / 1
1 / 0
C
1
B
1
1 / 1
0 / 1


(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
CLK(ret)
CLR
X
Z1
Z2
state

## 25.10 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the number of states in the associated state diagrams
- Minimize the use of hardware when problem require extra hardware
- Assume all inputs and outputs are positive logic unless stated otherwise
- Disregard all setup and hold-time issues
- For sequence detector problems assume the X input is stable when each clock edge arrives and that X can change no more than once per clock period.
- State all forms of control for your solution.

**1)** Provide a state diagram and black box diagram (BBD) that blinks a single LED at half the FSM clock frequency.

**2)** Provide a state diagram and black box diagram (BBD) that blinks a single LED with a 50% duty cycle. If the button is pressed, the LED stops blinking and either stays ON or stays OFF depending on when the button was pressed.

**3)** Provide a state diagram and black box diagram (BBD) that blinks a single LED with a 50% duty cycle. If the button is pressed, the LED always turns off.

**4)** Provide a state diagram and black box diagram (BBD) that implements a 2-bit binary counter. The output of the FSM is two LEDs that represent the 2-bit binary count.

**5)** Provide a state diagram and black box diagram (BBD) that blinks a single LED at 40kHz with a 25% duty cycle. Also state the required system clock frequency.

**6)** Provide a state diagram and black box diagram (BBD) that blinks a single LED at 100Hz with a 20% duty cycle. Also state the required system clock frequency.

**7)** Design a circuit that outputs a single blinking LED. If the button input to the circuit is ON, then the LED blinks at ½ the system clock frequency with a 50% duty cycle; otherwise the LED blinks at ¼ the system clock frequency with a 25% duty cycle. Show a BBD and state diagram for this problem.

**8)** Design a circuit that outputs a single blinking LED. If the button input to the circuit is ON, then the LED blinks at ½ the system clock frequency with a 50% duty cycle; otherwise the LED blinks at ¼ the system clock frequency with a 75% duty cycle. Show a BBD and state diagram for this problem.

**9)** Design a circuit that has eight 8-bit inputs (A,B,C,D,E,F,G,H). Each input is output for one clock cycle and the circuit cycles through the eight inputs continuously so long as the button is pressed. If the button is not pressed (button = ‘0’), the circuit outputs then starts over at outputting the eight values starting with the first value. Use an FSM in this design. Provide a BBD describing your circuit.

**10)** Design a circuit that has eight 8-bit inputs (A,B,C,D,E,F,G,H). Each input is output for one clock cycle and the circuit cycles through the eight inputs continuously. If the button is pressed (button = ‘1’), the circuit outputs does not increment and displays the same count for as long on the button is pressed. The circuit consecutively displays the values in the sequence so long as the button is not pressed. Use an FSM in this design. Provide a BBD and state diagram describing your circuit.

**11)** Design a circuit that has eight 8-bit inputs (A,B,C,D,E,F,G,H). Each input is output for one clock cycle and the circuit cycles through the eight inputs continuously. If the button is pressed (button = '1'), the circuit outputs the next values going forward in the sequence (…A,B,C,D…); otherwise the circuit outputs the values going backwards in the sequence (…D,C,B,A…). Use an FSM in this design. Provide a BBD and state diagram describing your circuit.

**12)** 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.

**13)** Repeat the previous problem but make the FSM reset when the it finds the indicated sequence.

**14)** 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.

**15)** Repeat the previous problem but make the FSM reset when the it finds the indicated sequence.

**16)** 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 output (Z). The Z output is '1' only when the desired sequence is detected. Provide two different state diagrams by considering the Z output to be a Mealy-type output and then a Moore-type output.

**17)** 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.

**18)** 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). This FSM also includes an asychrounous reset input RST that transitions the FSM to the "0100" state.

**19)** 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.

**20)** 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 both a Mealy and then Moore-type machine. Design this FSM to reset once the sequence is found.

**21)** Repeat the previous problem but make the FSM non-resetting when the it finds the indicated sequence.

**22)** 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 resets once the correct sequence is detected.

**23)** Repeat the previous problem but make the FSM non-resetting when a '1' occurs on the output.

**24)** 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 both a Mealy and Moore-type machine. Design your state diagram so that the FSM resets once the correct sequence is detected.

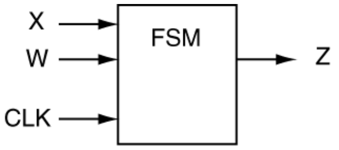
**25)** Repeat the previous problem but make the FSM non-resetting when the it finds the indicated sequence.

**26)** 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:

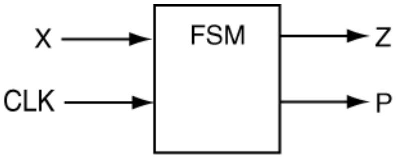
- the X input is stable when each clock edge arrives
- the W input can change when only when the proper sequence is found
- full encoding with three flip-flops will be used to encode the FSM (limits state diagram to eight states!)

| W | Sequence searched for on X Input |
|---|---|
| 0 | 1 0 1 1 0 |
| 1 | 1 0 1 1 1 |



**27)** Design a FSM that detects when the sequence "0101" appears on the FSM input (X). The output (Z) is '1' only when this sequence is detected. Implement this design as a ***Moore*** machine and resets when the correct sequence is found. This FSM has an output **P** that is a '1' when Z is '1' and when the bits previously processed by the FSM have even parity; otherwise the **P** output is '0' when Z is a '1' and the processed bits have odd parity. In other words, this FSM needs to always indicate the proper parity of all the bits previously seen when the correct sequence is found.

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. You only need to provide a state diagram for this design.



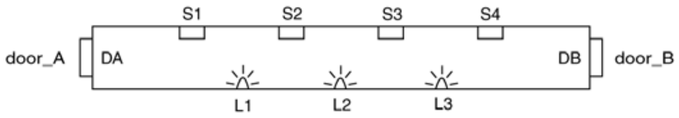
**28)** Provide a state diagram that describes a FSM that indicates when the sequence "10110" appears on the FSM input (X). The output (Z) is '1' only when this condition is detected. mplement this design as a Moore machine. Design you state diagram so that the FSM is non-resetting once the correct sequence is detected. This FSM also has a P output (positive logic) that is asserted when the parity of all the previous bits the FSM has seen is either odd (P=0) or even (P=1). 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.

**29)** Design an FSM that can be used control a car safety device. Unfortunately, you live in an area with lot of large hungry birds that like using your car windshield as a target. Your car windshield has three sensors attached to it and uses them to sense if bird poop is on your windshield. If a sensor has bird poop over it, it outputs a ‘1’; otherwise it outputs a ‘0’.

For this problem, if one and only one sensor senses poop, the FSM actuates a warning light (LT=’1’) until no sensor senses poop. If two sensors sense poop, the FSM turns on the windshield wipers (WW=’1’) until less than two sensors sense poop. If all three sensors sense poop, the car engine is automatically shut off (KILL=’1’) until no sensors detect poop.

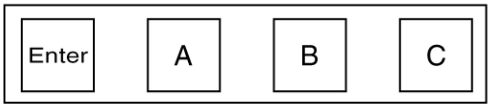
**30)** Design a FSM that creates a power-saving control of a set of hallway lights. The hallway has four sensor inputs (S1,S2,S3,S4), three light outputs (L1,L2,L3), and two door lock outputs (DA, DB) as indicated by the diagram below. The sensors indicate when a person is near and causes the nearest 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. When a person enters the hallway, the FSM locks the two doors from the outside so people can only exit (and not enter); both doors are unlocked when no one is inside. For this problem, make the following assumptions:

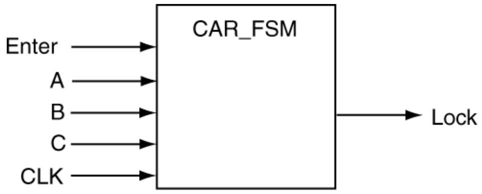
- The sensors completely sense the hallway with no overlap in the coverage area
- Only one person at a time can enter the hallway
- When a person enters one side of the hallway, the person will eventually exit on the other side



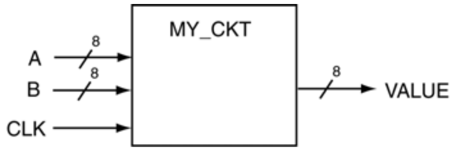
**31)** Design an FSM that can be used to control the lock mechanism of a car. The lock mechanism has four different button inputs in addition to a clock input (as shown below). In order to unlock the car door, you need to input the following code sequence: “enter”, “C”, “BC”, “AC”. Note that you sometimes must simultaneously press two different buttons. This FSM uses the clock to automatically reset the sequence (go back to the “waiting for an ***enter*** state”) if the proper code is not pressed before the next clock edge.

Provide a state diagram for this design. Be sure to state any assumptions you make for this problem. *Minimize the number of states in your design.*

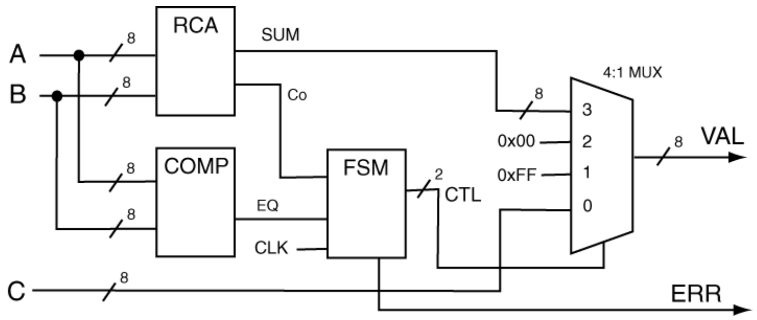


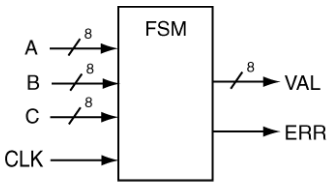


**32)** Design a circuit that does the following. The circuit checks the values of the A & B inputs on each rising clock edge. The circuit always outputs the value of the A input except under the following condition. If the circuit detects that the A & B inputs have been equivalent for three simultaneous clock cycles, the circuit ignores the values of A & B and outputs B for three clock cycles. After the circuit outputs B for three clock cycles, the circuit resumes checking for the A & B equivalency for the previously stated three clock cycles.

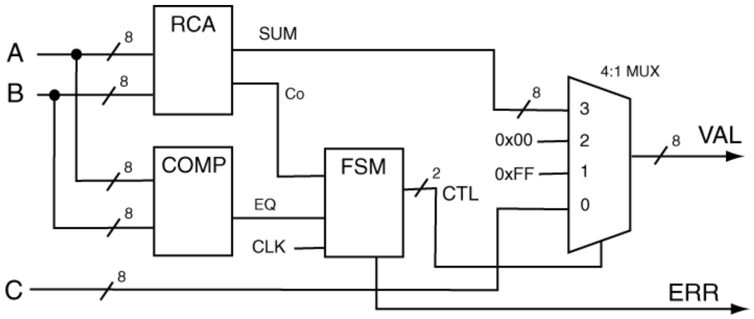


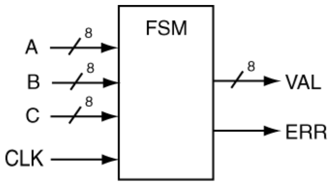
**33)** Using the listed circuit, design a FSM that outputs the sum of (A + B) as long as no carry is generated. If a carry is present on an active clock edge, the circuit outputs a 0x00 then 0xFF for one clock cycle each, then outputs the C value for at least two clock cycles but for as long as A does 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.





**34)** Using the listed circuit, design a FSM that outputs the value of the C signal while the values of A and B are equal. When A and B are found to be not equal on an active clock edge, the circuit outputs 0xFF then 0x00 for one clock cycle each, then outputs the sum of (A + B) as long as no carry is generated. If and when a carry is generated, the circuit goes back to outputting the value of C with the conditions previously described. The circuit also asserts ERR output whenever the circuit is not the value of C. Assume the FSM clock is much faster than then changes in A, B, and C. Disregard all setup and hold-time issues.





**35)** Design a circuit that detects when A=B. When it detects that condition, the circuit outputs 0xFF for at least three clock cycles, or until a button is pressed, whichever is shorter. When the circuit is not outputting 0xFF, it should output the value of A. The circuit does this operation continuously. Consider A & B to be 8-bit unsigned binary values.

**36)** Design a circuit that compares two 8-bit unsigned binary values. It compares A & B and continues to do so until the circuit detects that A>B on three on three consecutive clock cycles. The circuit continues to compare A & B and does so continues to do so until the circuit detects A<B on three clock cycles. While the circuit is looking for A>B, it turns on an LED; the circuit turns off the LED when it is looking for A<B on three clock consecutive cycles.

**37)** Design a circuit that continuously outputs the following sequence of operations. A+B, A-B, -A+B, -A-B, on consecutive clock cycles. If any of the operations are not valid, the circuit outputs zero and turns on an extra single bit signal. Assume the data inputs are 10-bit signed binary (RC format) operations.

**38)** Design a circuit that upon the pressing of BTN1, adds A+B+1 if BTN2 is pressed, or adds A+B if the BTN2 is not pressed. The results of the addition are displayed on the circuit's outputs. If three consecutive addition results generate a carry, then the circuit outputs zero for the sum until BTN1 is pressed, at which time it starts the addition operations again. The circuit outputs zero while it is waiting for a BTN1 press. Consider A, B and the result to be 8-bit unsigned binary numbers. The circuit keeps track of the number of times one and only one of the MSBs of the two operands are set (equal to '1'). This circuit also starts over when this count value reaches 31. This count is persistent while waiting for a BTN1 press, but the BTN1 pressed clears the counter before proceeding with the additions.

**39)** Design a circuit that upon the pressing of BTN1, adds A+B+1 if BTN2 is pressed, or adds A+B if the BTN2 is not pressed. The results of the addition are displayed on the circuit's outputs. If two consecutive addition results generate a carry, then the circuit outputs zero for the sum until BTN1 is pressed, at which time it starts the addition operations again. The circuit outputs zero while it is waiting for a BTN1 press. Consider A, B and the result to be 8-bit unsigned binary numbers.

# 26 Counters

## 26.1 Introduction

Counters are essentially a register with "features". A counter is another type of controlled circuit with many uses in digital design. We worked with low-level implementations of counters in a previous chapter; we now abstract our discussion upwards and discuss counters at the module level.

**Main Chapter Topics**

**COUNTERS**: Counters are simple registers with "features": This chapter defines and describes counters and their associated functionality and vernacular.

**Chapter Acquired Skills**

- Understand the various vernacular associated with counters
- Understand the various control inputs and status outputs of counters
- Use various flavors of counters in solutions to digital design problems

## 26.2 Counters: A Register with Features

A counter is a type of register, which means it inherits all the attributes of a register. The main new "feature" of a counter is that it outputs a given sequence of codewords, which is the "count" sequence. Counters typically synchronize their stepping through the count sequence to an active clock edge input to the counter. Counters can have one or more typical operational features, which we control with the counter's "control" inputs. Counters can also have status outputs that provide external circuits information about the counter.

Our approach is to define and describe every word and/or term you typically hear in the context of counters, and then do a few example problems. 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, HDLs make the modeling of counters trivial.

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.

- Because counters are registers, they are a sequential circuit.
- An active clock edge synchronizes a counter's traversing of the count sequence; there is one count value, or code-word, from the count sequence at each clock cycle.
- A counter's output represents a repeatable sequence of a given number of bits. The sequence the counter "counts" in does not change; the bit-width of the counter won't 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 (and is thus "circular").

There are many different types of counters out there, but introductory digital design courses typically only deal with a few types, which we handily list below. The less common counters that we do not list include Johnson counters, twisted-ring counters, and a few others that I can't recall now.

- **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 counts from 0-9 (up direction).

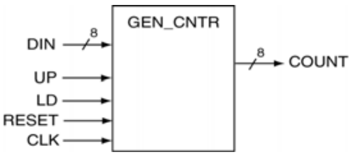
There is a set of vernacular associated with counters. Digital designer must be fluent with all the new terms associated with counters so they can converse with their peers and understand important things such as datasheets. Here are the common terms associated with counters:

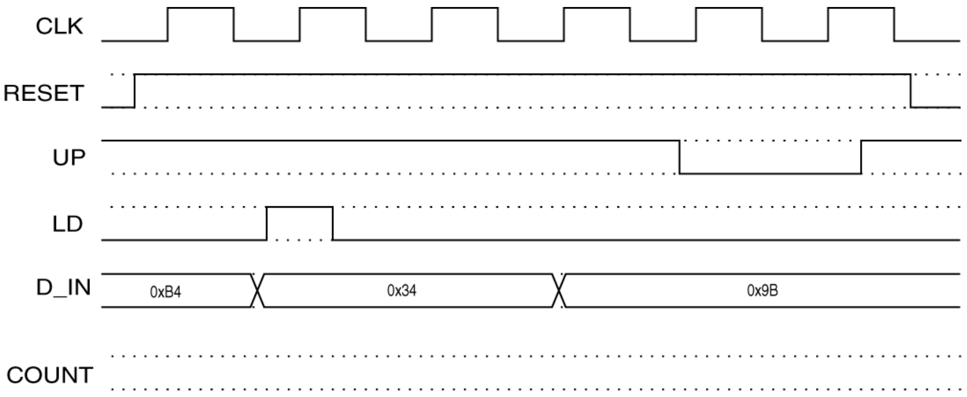
- **n-bit Counter**: A counter that uses n-bits to represent each of the values (or codewords) in its count sequence.
- **Up Counter**: A counter that counts up (increasing count values in count sequence).
- **Down Counter**: A counter that counts only down (decreasing count values in count sequence).
- **Up/Down Counter**: A counter that can counter either up or down according to a control 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 smallest representable value to its largest 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) or minimum count (for a down counter). Counters often use the term RCO to indicate when the counter has reached its terminal count value.
- **Parallel Load**: A characteristic of a counter or shift register indicating that all the storage elements in the device can simultaneously latch external values.
- **Circular**: When counters overflow their maximum or minimum counts, we consider them to "overflow". Counters are typically circular meaning that when the counter reaches the

maximum value, it automatically continues counting in the same direction starting at the minimum value[1].

**Example 26.1: Up/Down Counter Timing Diagram**

The block diagram on the right shows a model of an 8-bit counter. Use the block diagram to complete the following timing diagram. Assume propagation delays are negligent.





**Solution**: This problem shows you everything interesting and useful with counters. Figure 26.1 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 we modeled the **LD** signal, 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.

[1] This characteristic is for an up counter; the same idea is true for a down counter.

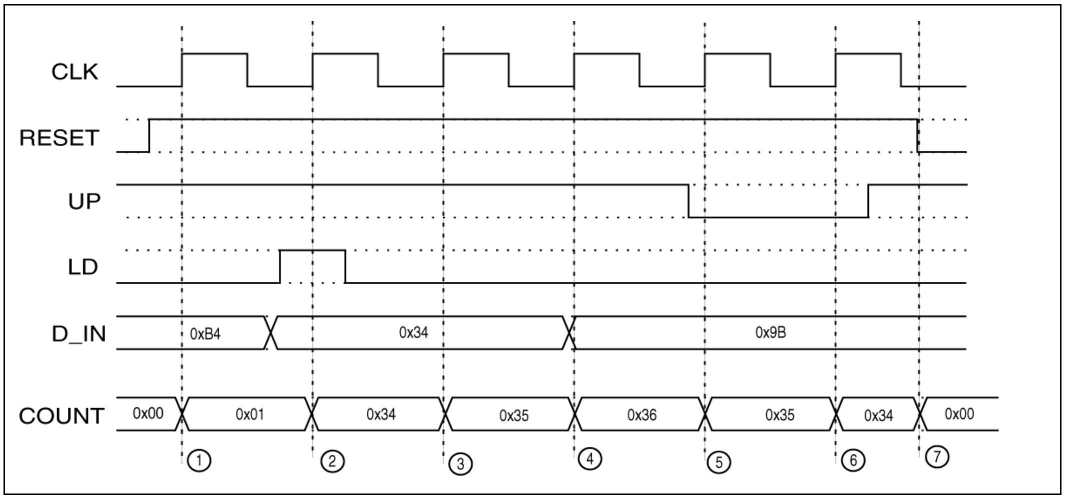
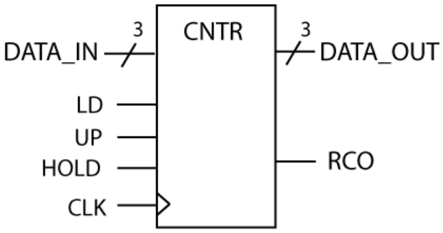
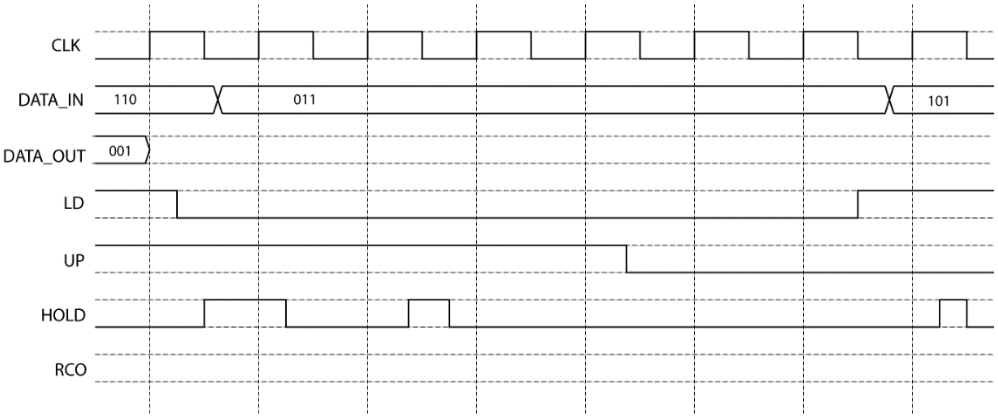


**Figure 26.1: The solution (with annotations) to this example.**

**Example 26.2: Counter Timing Diagram**

Use the BBD on the right to complete the timing diagram below. For this disarm, the **UP** input allows the counter to count up when enabled, and count down when not asserted. The **RCO** output indicates when the counter is outputting its terminal count. The **LD** input takes precedence over the **HOLD** and **UP** input. The **HOLD** input takes precedence over the **UP** input.





**Solution**: Figure 26.2 shows the solution to this example. Here are a few things to note:

- The **LD** signal is asserted on the first rising clock edge, which causes the counter to load the **DATA_IN** value into the register.

- The **HOLD** input is asserted on the second clock edge, which prevents the count output from changing.
- The first count operation occurs on the third rising clock edge; because the **UP** control input is asserted, the counter output increments.
- On the fourth rising clock edge, the counter increments again, but because the counter is at its terminal counter (in the up direction), the count output rolls over to "000".
- On the seventh rising clock edge, the counter decrements because the **UP** input is unasserted. The counter thus rolls under, and the resultant count is "111".
- The **RCO** status output asserts at two different times for distinctly different reasons. The first **RCO** assertion is because the **UP** input is asserted and the counter reaches its terminal count in the **UP** direction. The second **RCO** assertion is because the **UP** signal unasserted and the counter reaches its terminal count in the down direction.
- The **HOLD** input is synchronous, which means the FSM ignores the second and third pulses on the **HOLD** signal, as they do not overlap a rising clock edge.

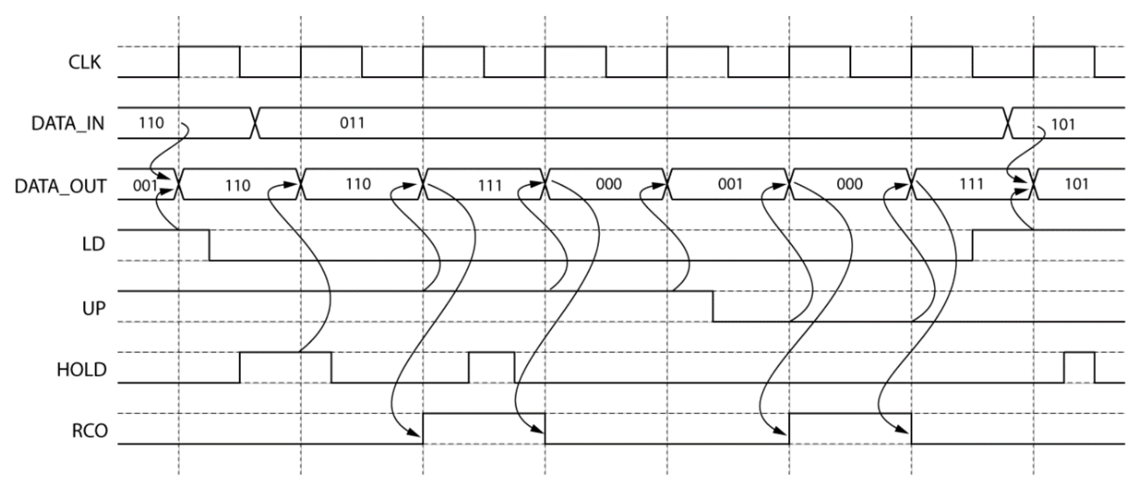


**Figure 26.2: The timing diagram solution for this problem.**

## 26.3 Typical Counter Feature Set Issues

When we use counters in our design, we must specify which features we are using and how the counter inputs and outputs represent those features. We can do this because counters are straightforward to design in HDL. Our mission is to use what counter features we need in our designs, but them must explicitly state how our counter inputs and outputs represent those features. The problem is that a seemingly simple counter input such as "**UP**" is not completely specified if we use it in a schematic. From the way we wrote "**UP**", we can assume that when asserted, it allows the counter to count. What is not clear is what happens when **UP** is not asserted. The only option is to include disambiguation information somewhere obvious in the schematic.

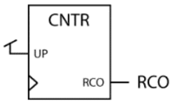
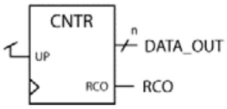
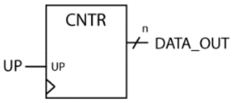
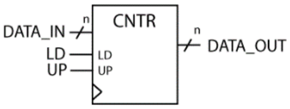
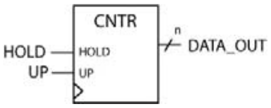
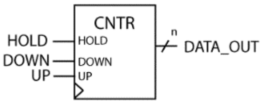
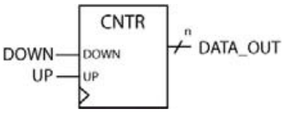
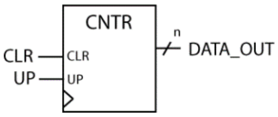
| Circuit | Comments |
|---|---|
| <br> | This counter contains no data inputs or outputs. It is an up counter because the **UP** signal is permanently connected to the asserted state. There is also **RCO**. If you use this module in your circuit, you must state the width of the counter as it is not included in the module. |
| <br> | This counter has a data output but does not contain data inputs. It is an up counter, but we don't need to state the data width as it is given by the width of the **DATA_OUT** signal. |
| <br> | This is an up counter, with a data output width of "n". We can assume that the counter counts up when the **UP** control input is asserted, but we must state what the counter does when the **UP** signal is not asserted. The **!UP** signal typically causes the counter to hold or count down. |
| <br> CNTR<br>DATA_IN<br>n<br>LD<br>UP<br>LD<br>UP<br>n<br>DATA_OUT | This counter has a data input, which means it needs a control signal that allows the counter to load the data. This signal is typically a **LD** signal ("load"). Anytime you have a data input signal, you need a control input for that loading. The **UP** signal means that it counts up when asserted, but you must specify what the counter does when **UP** is unasserted, as well as the precedence of the **UP** and **LD** inputs. |
| <br> | This counter has a **HOLD** signal, which means the counter does not change the output on the active clock edge. We must specify two items: 1) we must specify how the **!UP** affects the counter, and 2) we must specify which input (**HOLD** or **UP**) has precedence if they are both asserted. |
| <br> CNTR<br>HOLD<br>DOWN<br>UP<br>HOLD<br>DOWN<br>UP<br>n<br>DATA_OUT | This counter has three control inputs. In general, we must specify all possibilities for the set of three inputs. **DOWN** means the counter counts down and **UP** means it counts up, but we need to specify what happens if both **UP** & **DOWN** assert simultaneously. You can assume the **HOLD** input has precedence over the **UP** & **DOWN** inputs. |
| <br> | This is a counter that can count both up and down. We must specify what happens when both inputs are simultaneously asserted. |
| <br> | This counter has a **UP** and **CLR** control input. We generally assume that all **UP**/**DOWN**/**HOLD** inputs are synchronous, but it's common for clear-type inputs to be asynchronous. Whatever it may be, if we use this module in our design, we must specify whether the input is synchronous or not. You must specify the precedence of the **CLR** and **UP** inputs. |

**Table 26.1 Counter features that must be further explained when appearing in circuit.**

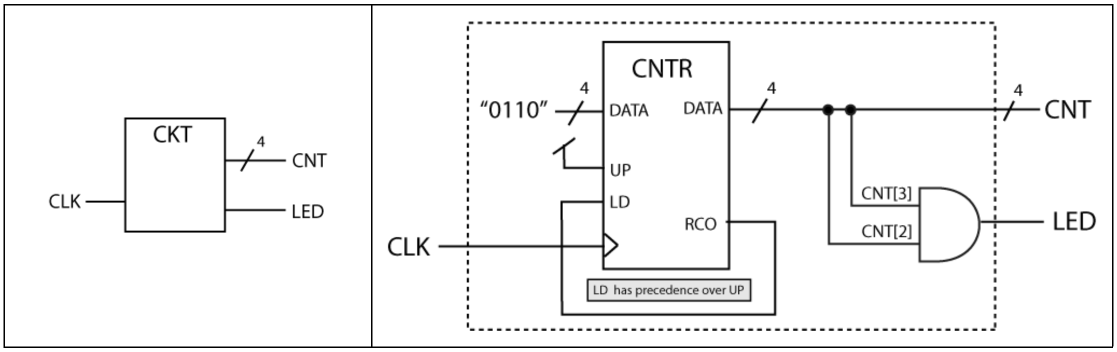
**Example 26.3: Design #1: Counter-Based Design**

Design a circuit with the following count sequence is: (6,7,8,9,10,11,12, 13, 14, 15, 6,7,8,9,10...) This circuit also has an extra **LED** output that is on when the counter output is greater than 11. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: This counter counts up except when the counter reaches its terminal count. There are two main design issues with this problem: 1) we need the counter to count in the given sequence, and 2) we need the **LED** output to indicate when the count is greater than eight. Figure 26.3(a) shows the top-level BBD for our solution.

The next step is to create an inventory of modules out final circuit requires. This counter does not start counting from zero; once the counter output hits 15, the next number in the sequence is 6. In terms of what we consider a typical counter, this means we need to load the counter with a new starting value once the counter hits its terminal count of 15. For this issue, what we need to is connect the counter's **RCO** status output to the **LD** control input of the counter. If we also connect the value of "0110" to the counter's data input, the counter loads the value of 6 after displaying the value of 15.

We also need an **LED** that indicates when the counter output is greater than 11. We can do this by including a comparator in our circuit, but we can do it with less hardware including some extra logic. Note that 12 in binary is "1100"; this means that when the two MSBs of the counter's output are set, the count must be greater than 11. The **LED** output is thus an ANDing of the two MSBs of the counter's output. Figure 26.3(b) show the lower-level BBD for our solution.



**(a)** **(b)**

**Figure 26.3: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.4 shows an example timing diagram for our solution; be sure to note these items:

- We include the **RCO** in this timing diagram to provide a deeper understanding of the problem. When the **RCO** output is asserted on a rising clock edge, the counter loads the value of 6 into the counter.
- The **LED** signal asserts when the count is greater than 11 and remains asserted until the **RCO** signal causes the counter to load the new starting value. We show the causality of these operations with the dot in the **CNT** signal output, synchronized to the rising edge of the clock.
- The **RCO** is a signal internal to the circuit; we include it in the timing diagram for clarity. The **RCO** signal asserts when the counter is outputting its terminal count.

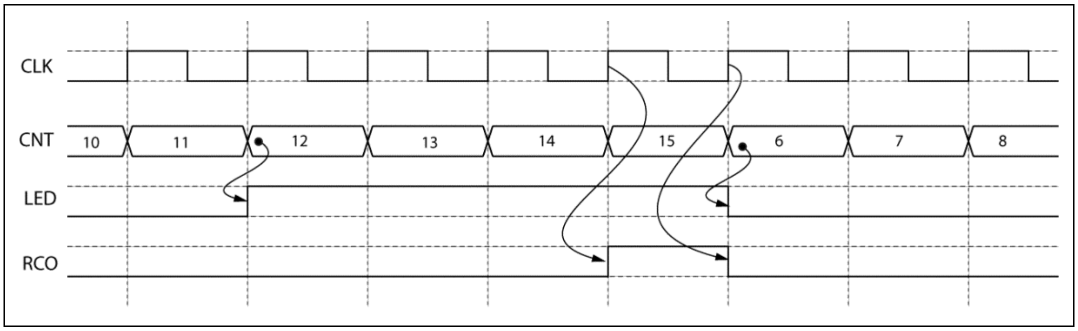


**Figure 26.4: The state diagram associated with this example.**

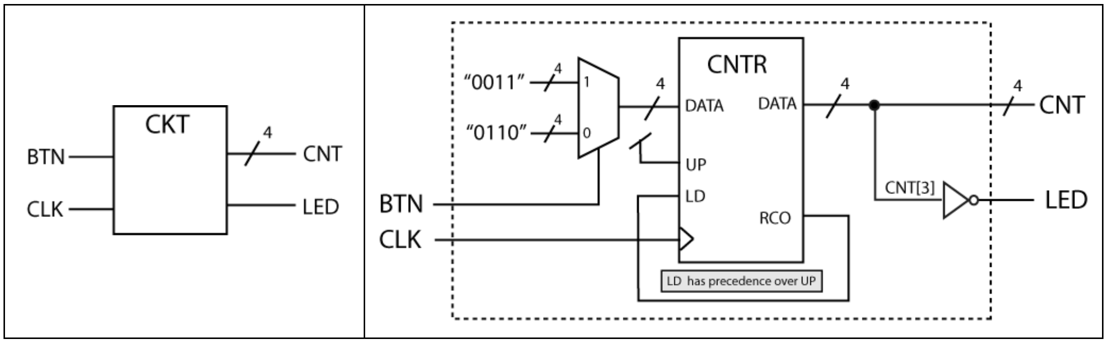
The **RCO** controls the **LD** input of the counter, which is a control input. We hardwire the UP input to always be asserted. These are both forms of internal control.

**Example 26.4: Design #2: Counter-Based Design**

Design a circuit with the following binary count sequences: the count sequence is either (3→15) or (6→15) based on whether a button is pressed or not, respectively. This counter has an extra **LED** output that indicates when the count is less than 8. Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: Both of these counts indicate we're dealing with an up counter. We need to load either 3 or 6 into the counter based on the press of a button. We also need some extra circuitry to indicate when the count is less than 8. Figure 26.5(a) shows the top-level BBD for our solution.

The next step is to make an inventory of the modules our solution requires. This problem requires that the circuit make a decision based on the button; this means the circuit needs a MUX to decide whether to load one of the two starting count values into the circuit. The circuit also requires a **LED** output that indicates when the count is less than 8. We can do this using a comparator, but there is an easier way. For an unsigned 4-bit binary count, if the MSB is set, the count is at least 8. This means we can provide the **LED** output as a complement of the MSB of the **CNT** signal. Figure 26.5(b) shows the lower-level BBD for our solution.



**(a)** **(b)**

**Figure 26.5: A block diagram for circuit (a), and underlying circuitry (b).**

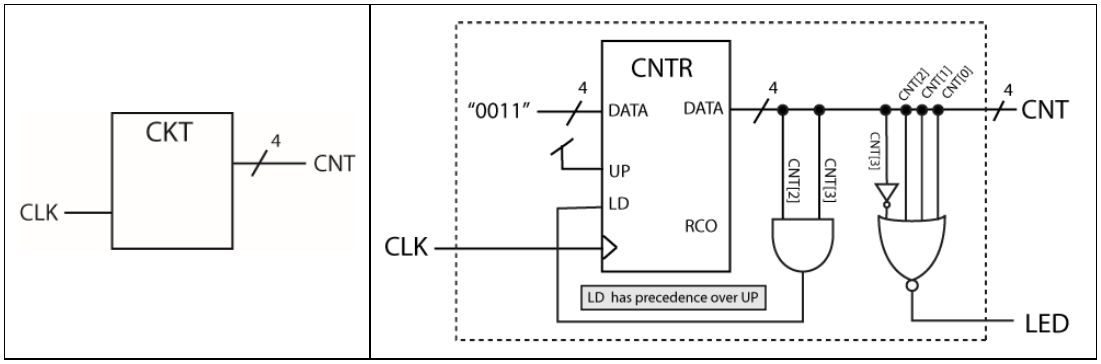
Both the MUX and the counter both have control inputs. An external button controls the MUX's control input while the **RCO** status output of the counter controls the **LD** control input of the counter. Thus, this circuit utilizes both external and internal control.

**Example 26.5: Design #3: Counter-Based Design**

Design a circuit that has the following count sequence: (3→12). This counter has an extra LED output that indicates when the count is 8. Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: Yet again, this counter is some form of an up counter. Previous problems dealt with starting the count at 3; but a problem we need to deal with making 12 the terminal count for the counter. Figure 26.6(a) shows the top-level BBD for our solution.

The next step is to create an inventory of the modules our solution requires. This counter needs the terminal count at 12, which is different from the terminal count of 15 (in the up direction) for typical 4-bit counters. We're tempted to use a comparator for is function, but it is easier to simply provide the logic that determines when the counter reaches the desired terminal count of 12. Additionally, we need to indicate when the count is 8 using the output **LED**. We once again choose to use logic instead of using a comparator. The circuit uses a NOR gate for this function; using an AND gate would require using three inverters rather than the one inverter when we use the NOR gate. Figure 26.6(b) shows the final solution for this example.



(a) (b)

**Figure 26.6: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.7 shows an example timing diagram for our solution. Here are a few things to note:

- The **LED** signal asserts when the counter output is eight; all other count values cause the **LED** signal to de-assert.
- The **LD** signal is an internal signal, but include it in the timing diagram for clarity. The circuit synchronously loads a new starting value into the counter when the counter's count value is 12. The **LD** asserts when after initially outputting 12 (after a nominal propagation delay of the state registers and next-state decoder); the actual loading of the starting count values happens on the next clock edge.

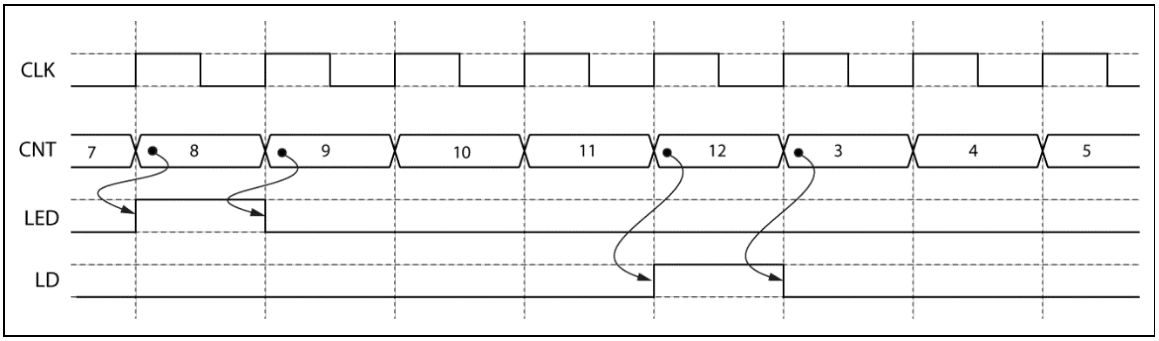


**Figure 26.7: The state diagram associated with this example.**

Finally, the circuit uses internal control; the two MSBs of the count control when LD signal of the counter, which dictates when the counter loads the new starting value. Additionally, we hardwire the UP signal to always be asserted.

**Example 26.6: Design #4: FSM-Based Specialty Counter**

Design a circuit that drives four LEDs with a binary count. The circuit only counts up. The circuit also has an extra LED output that turns on for one [0,15] count, then off for the next [0-15] count, etc. Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution. Use a FSM to control your circuit and provide a state diagram if necessary. Also, describe what controls your final solution. Finally, state the frequency of the blinking LED in terms of the system clock.

**Solution**: The first step in this solution is to generate the to-level BBD, which we show in Figure 26.8(a). The next step in this solution is to discern which underlying modules the solution requires.

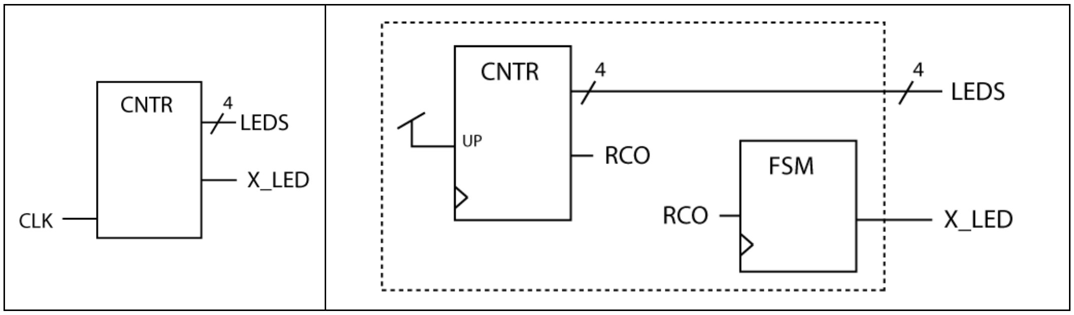
The next step in this problem is to make an inventory of the modules the solution requires. This problem requires a counter; the problem states that the circuit is counting using a standard 4-bit unsigned binary counting sequence: [0,15]. The circuit's other requirement is to turn on an LED for one traversal of the sequence, then turn it off for the next traversal of the counter sequence. This means that the circuit needs to "remember" whether the LED is on (so it can turn it off) or if the LED is off (so it can turn it on).

There are several approaches to doing generating the notion of being able to remember if the LED was on or off. The most straightforward approach is to simply use a 5-bit counter; the MSB of the 5-bit counter could then server as the **X_LED** output. However, the problem requests that we use an FSM, so this solution is not valid (though interesting and instructive). The FSM for this solution needs to monitor the **RCO** output of the 4-bit counter. When the FSM detects the asserting of the **RCO**, the FSM transitions from the X_LED on state to the **X_LED** off state (or vice versa). The status input to the FSM is the **RCO** status output of the counter. We can then control the LED output with the control output of the FSM. Figure 26.9 shows the state diagram we use to describe the FSM in our solution.

Figure 26.8(b) shows the lower-level BBD for our solution. Here are a few things to note in the circuit solution:

- The counter is always counting up, so we connect the counter's **UP** control input to '1'.
- The dotted line in Figure 26.8(b) represents the CKT box in Figure 26.8(a).

- The circuit diagram in Figure 26.8(b) does not connect the control and status signals. Anyone reading the diagram understands that the **RCO** output from the counter module connects to the RCO input to the FSM.
- We omit the clock signal from the lower-level BBD for clarity. Routing clock signals and control/status signals quickly makes circuit diagrams unreadable, so we generally do not do it unless there is some compelling reason to do so. Note that we do leave in the “triangles” to remind ourselves that the associated devices are indeed synchronous and do in fact require connection to a clock signal.



**(a)** **(b)**

**Figure 26.8: A block diagram for circuit (a), and underlying circuitry (b).**

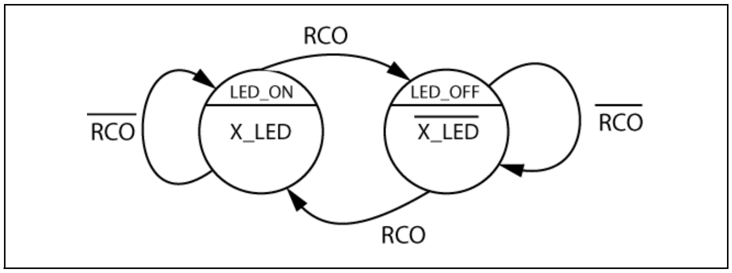


**Figure 26.9: The state diagram associated with this example.**

The circuit uses a 4-bit counter. The FSM uses the **RCO** status output from the counter to control transitions between the two states in the FSM. This means that the FSM changes states every $2^4$, or 16 clock cycles. We calculate the blink frequency from the fact that the LED needs to be on for 16 counts and then off for 16 counts, as directed by the FSM. This means the system clock frequency is 32 times greater than the blink frequency.

We tie the counter’s **UP** control input in this circuit to ‘1’, so that is internal control. The circuit also contains an FSM. This circuit has both internal and circuit control.

**Example 26.7: Design #5: FSM-Based Specialty Counter**

Design a 4-bit binary counter that counts up through two full count sequences, then down for two full count sequences, the up again, etc. This circuit has two extra **LED** outputs. One **LED** toggles each system clock cycle; the other **LED** toggles each completion of the count sequence, no matter if it is the up or down sequence. Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

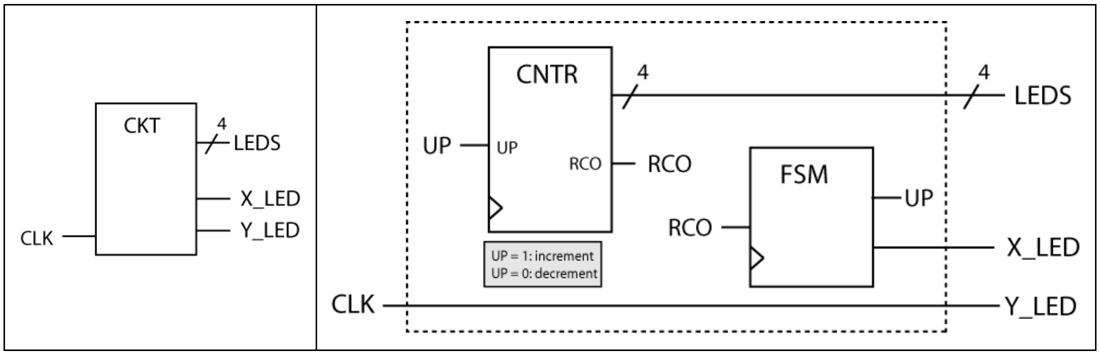
**Solution**: This example uses an up/down counter with a plain 4-bit binary up counter : [0,15]. In addition, there are two LED outputs. We can generate the desired output for one LED by simply connecting it to the system clock. The other LED toggles with each traversal of the count sequence in either the up or down direction.

The control we're looking for in this circuit is for the LED blinking at the lower frequency and the count directions. We can control the blinking LED using states in a state machine: one state for both the on and off LED. The other form of control this circuit requires is the count sequence traverses in the up direction for two sequences, followed by the down direction for two sequences, which means the circuit requires memory to know which state it is in. The most straightforward way to do this is with a FSM.

The next step is to make an inventory of the modules this circuit requires. From the description in the previous paragraph, we seem to only require a standard 4-bit up/down counter and a FSM; if we need other modules, we can add them later.

Figure 26.10 shows the final circuit for this problem. Here are a few comments to solidify your understanding

- One LED connects directly to the clock input without any other logic. The BBD shows this connection, but we once again do not connect the clock to the synchronous to the other synchronous modules in the circuit to make the diagram more readable.
- The other LED is an output of the FSM, thus the FSM controls the blink frequency of that LED.



(a) (b)

**Figure 26.10: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.11 shows the state diagram modeling the FSM that controls this circuit.

- The **RCO** output of the counter controls all state transitions.
- We use Moore-type outputs to control the slower blinking **LED** and the count direction; every state transition toggles the value of the FSM's Moore output.
- The FSM controls the counter direction; the top two states make the counter up; the bottom two states makes the counter count down.

Lastly, we control the circuit using another circuit, namely the FSM. The FSM reads the status output of the counter (**RCO**) and controls the count direction by assigning values to the counter's **UP** control input.

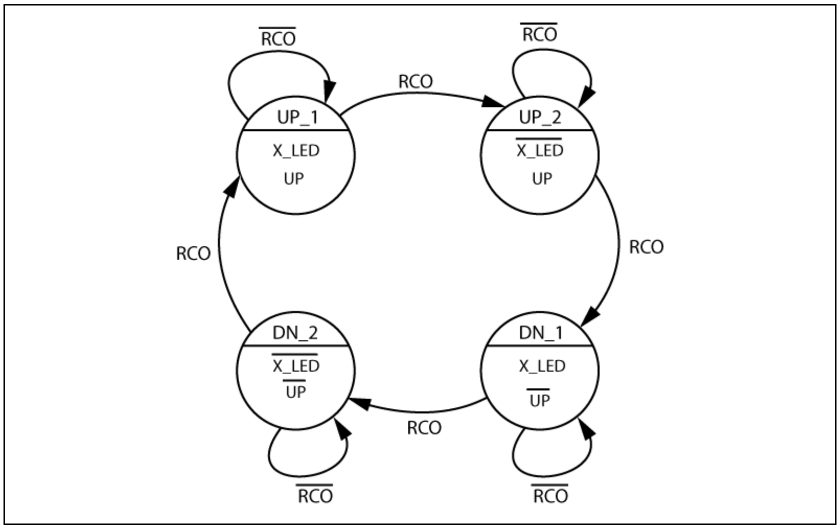


**Figure 26.11: The state diagram associated with this example.**

The FSM in this circuit controls the counter's **UP** input, as well as other operations in the circuit. Because the FSM provides all control in this circuit, we consider the solution as having circuit control.

**Example 26.8: Design #6: Counter-Based Design**

Design a circuit that outputs the following sequence:

(…14,15,0 0 0 1,2,3,4,5,6,7,8,9,19,11,12,13,14,15,0,0,0,1,2…)

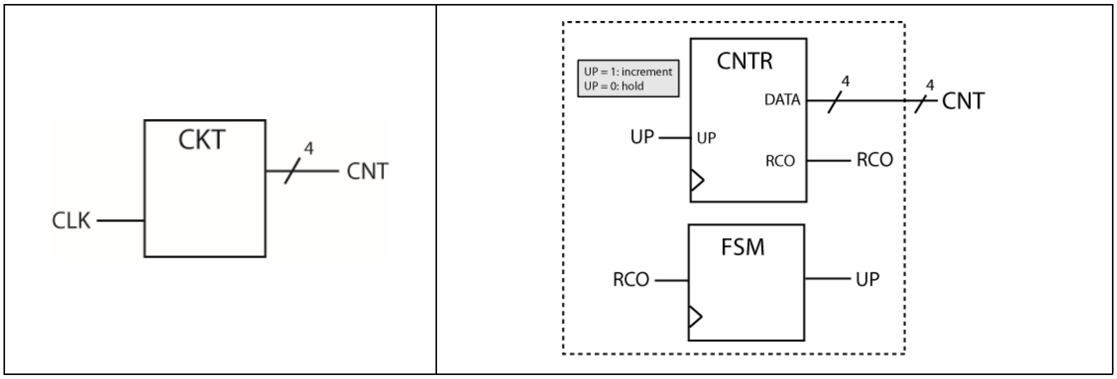
Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: This counter is some form of an up counter, but has a special feature: the counter outputs the value of zero for three counts (clock cycles). Figure 26.12(a) show the top-level BBD for our solution.

The next step is to make an inventory of the modules our solution requires. Here is an example thought process:

- We need something to control the count operation; an FSM is a straightforward choice.
- The heart of this design includes a counter, which the FSM controls. Specifically, we need a counter with an **UP** input such that the counter counts up when the **UP** input is asserted and holds when the **UP** input is not asserted. The FSM must control the **UP** control input of a counter.
- We only require a standard 4-bit up counter and a FSM; if we need other modules, we toss them in later without losing and credit

Figure 26.12(b) shows the lower-level BBD for our solution. We include a note in the lower-level BBD that indicates how the counter's **UP** control input operates; the circuit would not be 100% if we did not do this.



**(a)** **(b)**

**Figure 26.12: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.13 shows state diagram for modeling the FSM that controls our circuit. Here are some interesting things to note:

- The state diagram indicates the circuit waits for an **RCO**, and then does not increment the count for effectively three clock cycles. Although there are only two states where the **UP** signal is not asserted, it is not until the third clock cycle after **RCO** asserts that the counter once again begins increment.
- The transition in the “count” state is conditional as the FSM is waiting for an asserted **RCO** signal to transition to the “hold_1” state. Transitions from the other states are unconditional as those two states only serve to provide a delay on the count of zero.

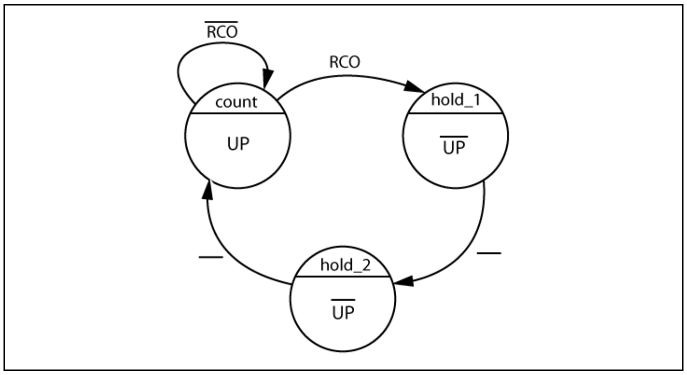


**Figure 26.13: The state diagram associated with this example.**

The timing diagram in Figure 26.14 shows the critical timing of the circuit. Here are some things to note:

- When the counter reaches its terminal count, the **RCO** asserts. When **RCO** is asserted, the FSM transitions to the “hold_1” state.
- When the FSM enters the “hold_1” state, it de-asserts the **UP** signal, which is a control signal for the counter. The timing diagram shows a small offset to highlight the fact that the **UP** signal de-asserts after the active clock edge, which means the counter does not increment on the next clock edge.
- The FSM de-asserts the **UP** signal for two states; the **UP** signal asserts upon entering the “count” state, but does not cause the counter to increment until the next active clock edge. It seems that there should be three hold states in this FSM, but the timing diagram shows that this is not the case.

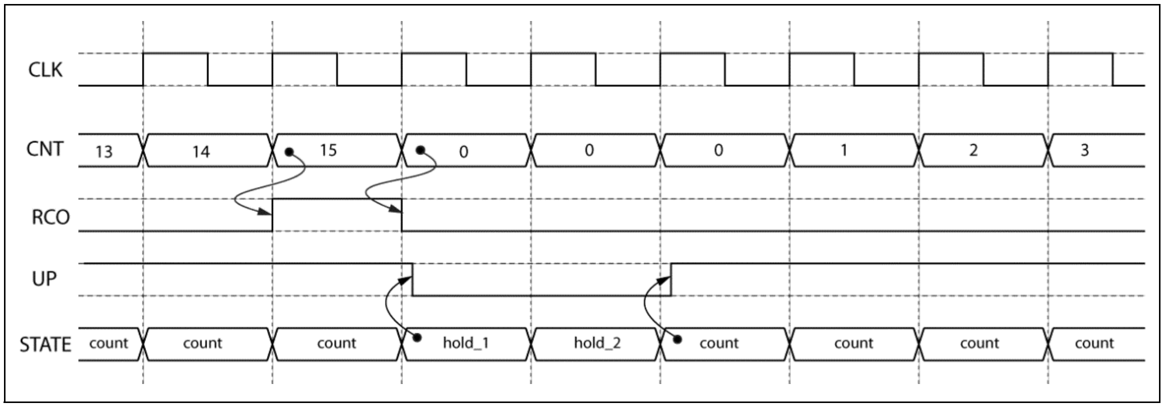


**Figure 26.14: The state diagram associated with this example.**

The counter in this circuit requires control; this problem uses an FSM to control the counter. Thus, this circuit uses circuit control.

**Example 26.9: Design #7: Counter-Based Design**

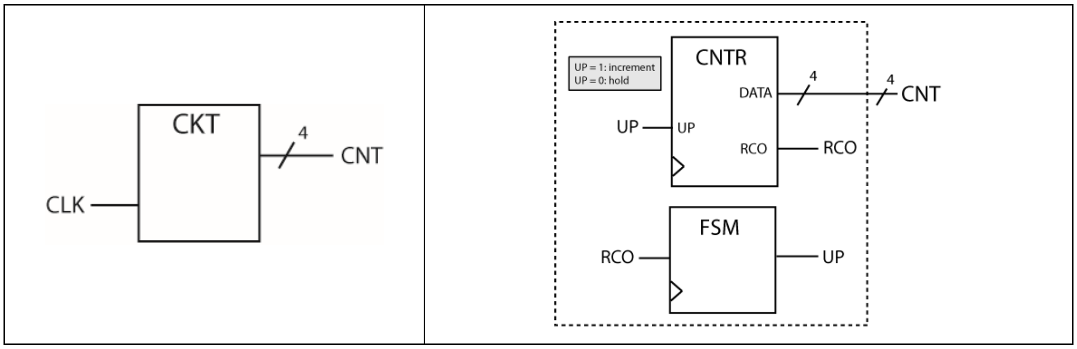
Design a circuit that outputs the following sequence:

(…14,15,15,15,0,1,2,3,4,5,6,7,8,9,19,11,12,13,14,15,15,15,0,1,2…)

Minimize your use of hardware in your design. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: This need requires a special up counter, which is similar to the previous problem, but now the counter pauses for three clock cycles on the count of 15. Figure 26.15(a) shows the top-level BBD for this problem.

The next step is to make an inventory of the modules our solution requires. The heart of this problem is a counter; because the counter pauses on one particular count, we know we need to control the counter in a special way. The best way to do this is to include a FSM that controls the UP input of a counter. Because the count spans from [0,15], we know we need a 4-bit counter. Figure 26.15(b) shows the lower-level BBD for the solution.



(a) (b)

**Figure 26.15: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.16 shows the state diagram for the solution. Here are some important items to note.

- We model the UP signal as a Mealy-type output in the "count" state and as a Moore-type output in the other two states. The UP output of the FSM is officially a Mealy-type output, but modeling it as both types of outputs makes the state diagram more readable.
- Once the counter reaches 15, the RCO asserts, which causes the FSM to transition to the "hold_1" state. The state diagram de-asserts the UP input when the RCO asserts, which holds the count at 15. The count holds at 15 until it eventually returns to the "count" state, where the counter starts at zero due to the asserted UP signal in the "hold_2" state.
- The two hold states provide the three count delay when the count reaches 15.

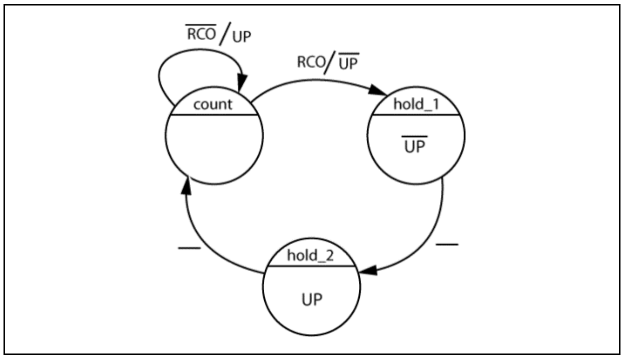


**Figure 26.16: The state diagram associated with this example.**

Figure 26.17 shows an example timing diagram for our solution. We include the RCO in this timing diagram to provide a deeper understanding of the problem. Here are some important points to note in the timing diagram.

- The RCO asserts when the CNT output reaches its terminal value of 15.
- The asserted RCO causes the FSM to transition from the "count" state to the "hold_1" state.
- The UP signal re-asserts upon entering the "hold_2" state. We place tiny delays in the timing diagram to indicate that it's not until after enter the "hold_2" state that the UP signal asserts. This means that the counter does not increment from 15 to zero, until the clock cycle that transitions the FSM from the "hold_2" state to the "count" state. This is a particularly important mechanism to understand.
- When the FSM enters the "count" state from the "hold_2" state, the count is now at zero, and the RCO de-asserts, which allows the FSM to count starting from zero in the "count" state.

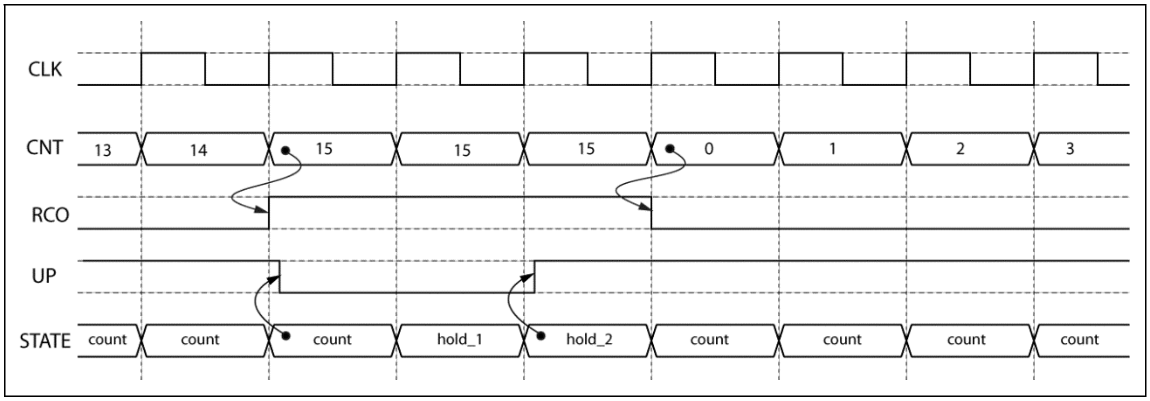


**Figure 26.17: An example timing diagram for our solution.**

The counter in this circuit requires control; this problem uses an FSM to control the counter. Thus, this circuit uses circuit control.

## 26.4 Special Counter Circuits: Event Counters

When you think of counters, you may have the idea that they are simply circuits that step through a given output sequence of values in an automatic manner. However, we can configure counters to act as *event counters*, where the main task is to determine the number of times a certain "event" occurs. The event in question could be things like how many times an RCA generates a **CO**, how many times a circuit sees a certain value, etc. Additionally, this form of counting is a special form of accumulation, but instead of accumulating any value, the circuit is always accumulating '1'.

In the special case where you require a circuit that always accumulates '1', it's always best to use a counter rather than an accumulator[2]. Counters are registers, so they naturally cover the register part of the typical accumulator. Counters typically increment the count, which is an operation that always adds '1 to the current count; this operation handles the addition operation, which the RCA associate with a standard accumulator handles.

**Example 26.10: Design #7: Counter-Based Design**

Design a circuit that counts the number of times the value of 0x47 appears on the circuit's input. Upon the press of a button, the circuit evaluates the circuit's 8-bit unsigned binary data input on 1024 consecutive rising clock edges. The circuit has two outputs: one output shows the final count; the other output is an **LED** that turns on when the circuit completes the inspection of the 1024 input data values. Provide a top-level and lower-level BBD for your solution, and a state diagram if necessary. Also, describe what controls your final solution.

**Solution**: This example "counts" something (a value on the input), which means it's an ideal situation to use a counter as an *event counter*. The counter then counters the number of times the value of 0x47 appears on the circuit's inputs. Our first mission is to generate a top-level BBD. The problem clearly states that the circuit's inputs are an 8-bit data signal, a button, and a clock signal, but the problem does not clearly state the output.

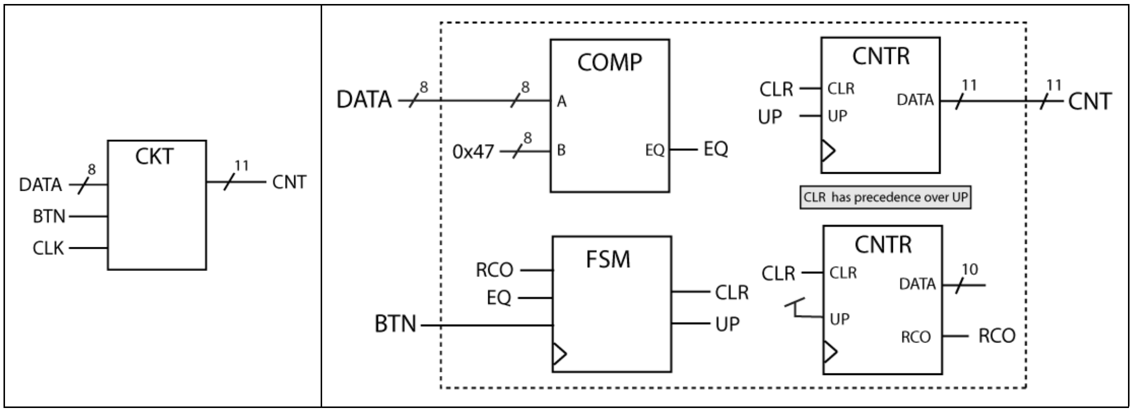
The circuit examines 1024 data inputs on consecutive clock edges; the maximum possible count determines the data width of the count output. The extreme case is where all data inputs are 0x47, which would result in a count of 1024. It would be tempting to think the width of the count output is ten bits due to the fact that $2^{10}$=1024, but this is not correct. The maximum value for ten bits is when all bits are set, which is the number 1023; this indicates that using ten bits for the output width is not sufficient. Thus, the bit-width of the output is eleven bits. Figure 26.18(a) shows the top-level BBD for our solution.

The next step is to make an inventory of the modules our solution requires. We know we require one counter to count the occurrences of 0x47; but we also need another counter to counts 1024 times, which is the number of data inputs we need to examine. We need to make comparisons, so we need a comparator as well. Finally, we need a FSM to control the operation of the circuit. Figure 26.18(b) shows the final circuit for our solution. Here are some important things to note about our solution.

- Both counters use the same **CLR** signal, which is an output of the FSM.
- We hardwire the comparator's **B** input to the value we're checking for.

[2] Recall that an accumulator comprises of an RCA and a register.

- The circuit has no need for the **DATA** output of the 10-bit counter, so we leave it unconnected. The circuit completes examining the input data when the **RCO** associated with the 10-bit counter asserts.
- The diagram indicates that the counter's **CLR** control input has precedence of the counter's **UP** control signal. We officially need to list this to ensure the circuit and corresponding state diagram make sense.
- The 10-bit counter always counts up, we connect the **UP** control of the 10-bit counter to '1'.



**(a)** **(b)**

**Figure 26.18: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.19 shows the state diagram describing the FSM in Figure 26.18(b). Here are some fun things to note about the state diagram.

- The "wait" state waits for a button press. The FSM uses the **CLR** as a Mealy-type output, so when the FSM detects a button press, it asserts the **CLR** signal. Modeling **CLR** as a Mealy-type output is arbitrary, but since it saves a state, we typically do this when we can.
- The "c_A" state is where the circuit actuates the event counter. In this case, we assign the **UP** control output of the to the comparator's **EQ** status output. While it seems tempting to connect the **EQ** directly to the event counter's **UP** control input, this would potentially cause the event counter to increment while in the "wait" state. There are a few ways to handle this issue; having the FSM deal with the issues is always the best solution.
- The FSM continues in the "count" state until the FSM receives an asserted **RCO** output from the circuit's event counter.

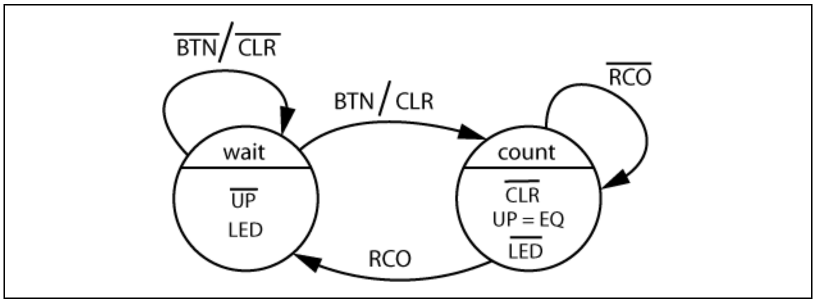


**Figure 26.19: The state diagram associated with this example.**

This circuit has two forms of control. First, the BTN input is an external control as it starts the search process in the circuit. The two counters in the circuit have control inputs that the circuit's FSM provides; thus, the circuit has both external control and circuit control.

## 26.5 Digital Design Foundation Notation: Counters

We consider the counter to be a Digital Design Foundation modules. The counter is a controlled circuit. Figure 26.20 shows the appropriate digital design foundation notation for the counter. This foundation module is more flexible and thus harder to define than other foundation modules. For example, the only required signal for a counter is a clock, as we consider the counter a synchronous device; the only required information we need to know about counters is the bit-width of their internal storage elements. Because counters are straightforward to design and/or model in with an HDL, we typically only include (or connect) counter inputs and outputs as we need them.

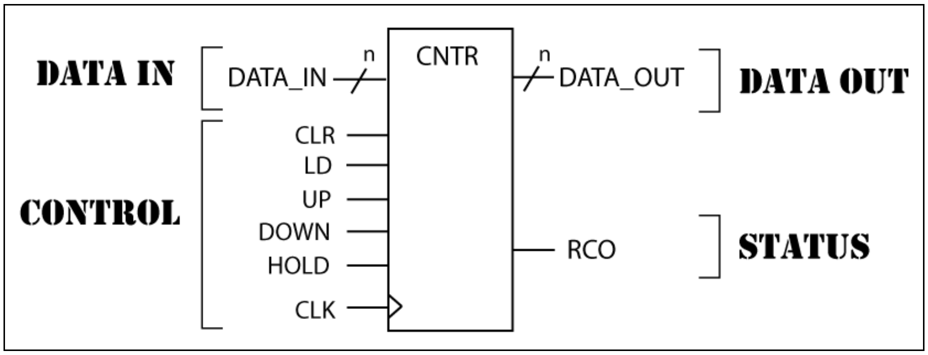


**Figure 26.20: Typical data, control and status signals for a counter.**

Table 26.2 shows all the inputs and outputs that we can typically associate with a counter. Essentially Table 26.2 lists a set of features that we can apply to a counter. The two things to note about this list is 1) that not every counter has every listed feature, and 2) actual counter implementations typically combine many of the control features as required into less signals than listed.

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | DATA_IN | A counter is a register, so it can typically load data in to the counter’s storage elements. The DATA_IN input is the data that is loaded to the counter. |
| OUTPUT DATA | DATA_OUT | A counter is a register, so the DATA_OUT signal is the data currently being stored in the counter’s storage elements. The DATA_OUT signal is necessarily a given value in the counter’s count sequence. |
| CONTROL | CLK | Counters are typically synchronous circuits, in that many counter operations are synchronized with the active edge of the clock signal. |
| | LD | As with registers, this signal controls the latching (loading) of the DATA_IN signal to the counters storage elements. This signal is always synchronous. |
| | CLR | Latches 0’s into each of the counter’s storage elements. Can be synchronous or asynchronous. |
| | HOLD, EN | Prevents the output from changing (HOLD) or enables the output to change (EN) based on other control signals (sort of the same idea) |
| | UP | Directs counter to count “forward” in the sequence; the an asserted up signal counts forward while an non-asserted count signal counts backwards |
| | DOWN | Directs the counter to count “backward” in the sequence. |
| STATUS | RCO | This signal indicates when the counter has reached the terminal value in the associated count sequence. For counters counting up, the terminal value is the max count value (all internal storage elements set); for counters counting down, the terminal value is the min counter value (all internal storage elements cleared). |

**Table 26.2: The foundation description for a full-featured counter.**

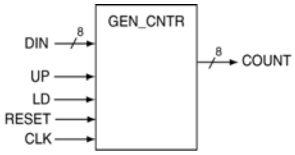
## 26.6 Chapter Summary

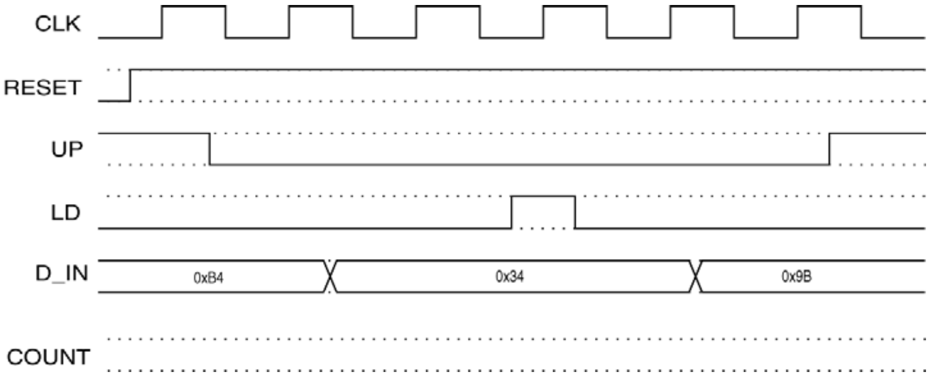
- A counter is a special type of register; we consider a counter to be a register with added features beyond that of a simple register. Counters typically have load and clear inputs as do simple registers, but also have extra inputs to control the operation of the counter.
- The counting and most other operations on a counter are synchronous. Often times the clear control input is asychrounous. BBDs must state such information in order to be correct.
- Extra inputs to counters are hold inputs (prevents counter output from changing state) and up & down inputs (allows the counter to increment or decrement, respectively).
- The exact function of counter outputs are not always evident from schematic diagrams, which means BBDs must include appropriate information to disambiguate the such issues in order to achieve correctness.
- Counters often include status outputs such as RCO (ripple carry out) that indicate when the counter has reached its terminal count.
- Counters output a repeatable sequence of values, which we refer to as the count. Counters come in many flavors; this text only uses binary counters. Other counter types include decade counters, Johnson counters, twisted ring counters, etc.
- Counters can act as accumulators when the item being accumulated is a ‘1’. We often refer to counters configured in this manner as event counters.

## 26.7 Chapter Exercises

**1)** 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 DIN loading into 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.





**2)** Show a schematic that uses two standard 8-bit up counters to implement a 16-bit up counter.

**3)** In your own words, describe how it is that a counter can replace an accumulator in certain circumstances.

**4)** Briefly describe the difference between the RCO when the counter is counting up verse counting down.

## 26.8 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the number of states in the associated state diagrams
- Minimize the use of hardware when problem require extra hardware
- Assume all inputs and outputs are positive logic unless stated otherwise
- Disregard all setup and hold-time issues
- For sequence detector problems assume the X input is stable when each clock edge arrives and that X can change no more than once per clock period.
- State all forms of control for your solution.

**1)** Design a circuit that displays the sum of two 8-bit unsigned binary inputs. The circuit also outputs the number of times the two input values generate a carry. The count of the number of carrys is based on what values are being summed on the rising clock edge of the circuit; the maximum value of this count is 255 (the counter rolls over to 0 automatically). Don't use an FSM in this design.

**2)** Design a circuit that displays the sum of two 8-bit unsigned binary inputs. The circuit also outputs the number of times the two input values generate a carry. The count of the number of carrys is based on what values are being summed on the rising clock edge of the circuit; the maximum value of this count is 255 (the counter rolls over to 0 automatically). To make this circuit start operating, it must detect a button press, which also causes a clearing of the counter. Also, once the count value reaches the maximum count, it rolls over, but the circuit must wait for another button press before the cycle is repeated.

**3)** Design a circuit that outputs either the output of a 4-bit binary counter or zero. If the button input to the circuit is pressed (button='1'), then circuit outputs the binary count; otherwise the circuit outputs zero. The binary counter is an up counter and thus always counts up whether the button is pressed or not. Don't use an FSM in your design.

**4)** Design a circuit that outputs either the output of a 4-bit binary counter or zero. If the button input to the circuit is pressed (button='1'), then circuit outputs the binary count; otherwise the circuit outputs zero and disables the counter as long as the button is not pressed. The binary counter is an up counter and thus always counts up. Don't use an FSM in your design.

**5)** Design a circuit that outputs the following sequence: (…1, 0, 3, 0, 5, 0, 7, 0, 1, 0…) on an active (rising) clock edge. Minimize the bit-width of the sequence outputs. Don't use an FSM in your design.

**6)** Design a circuit that has eight 8-bit inputs (A,B,C,D,E,F,G,H). Each input is output for one clock cycle and the circuit cycles through the eight inputs continuously. If the button is pressed (button = '1'), the circuit starts over at outputting the eight values starting with the first value (A). The circuit outputs the first value (A) as long as the button remains pressed. Don't use a FSM in this design.

**7)** Design a circuit that outputs the following sequence: (…2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 0, 0, 2…) an active clock edge (rising). Minimize the bit-width of the sequence outputs. Don't use an FSM in your design.

**8)** Design a circuit that does the following. If the circuit detects a button press on an active clock edge, the circuit increments the 8-bit output, but then holds that count value for three clock cycles before waiting for the next button press. While the circuit is holding the count value, the circuit outputs the value 0xF3. Consider this counter to be an 8-bit counter and that it only counts up.

**9)** Design a circuit that has four 8-bit inputs (A,B,C,D). Each input is output for one clock cycle and the circuit cycles through the four inputs continuously. If the button is pressed (button = ‘1’) for two clock cycles, the circuit starts the count sequence over starting with the A input. The circuit keeps traversing the sequence while the number of button presses is less than two.

**10)** Design a circuit that implements a 4-bit binary count. The counter count up normally counts up, but if a button is pressed for two clock cycles, the counter waits three clock cycles before restarting the count at the value of 2. The circuit also has three LED outputs; when the circuit is not counting, the LEDs show “110”; otherwise the LEDs show “001”.

**11)** Design a circuit that has four 8-bit unsigned binary inputs. When a button is pressed, the circuit finds the largest of the four input values and continually outputs that value. Use no more than one comparator in your design. Use an FSM in this design.

**12)** Design a circuit that drives four LEDs with a binary count. The circuit only counts up, and only counts up if a button is ON and has been on for at least two clock cycles. Use a standard counter for this problem. The circuit also has two extra LED outputs; one LED indicates when the circuit is in the counting mode; the other LED output indicates when the button is ON but the count is not counting.

**13)** Design a circuit that drives four LEDs with a binary count. This circuit only increments every third clock cycle. This circuit also has an extra LED output that blinks with a 66.7% duty cycle.

**14)** Design a circuit that drives four LEDs with a binary count. This circuit only increments every third clock cycle. This circuit also has an extra LED output that blinks with a 33.3% duty cycle. This circuit also has a button input that synchronously clears the counter when pressed; otherwise, the button has no effect.

**15)** Design a circuit that drives five LEDs with a special binary count. The count sequence is: (3, 4…29, 30, 31, 3, 4…). This circuit also has an extra LED output that is ON when the counter output is greater than 15. Do this problem both with and then without an FSM.

**16)** Design a circuit that drives four LEDs with a binary count. This circuit has a button that controls the resetting of the circuit, where the button must be pressed for two clock cycles in order for the a synchronous reset to occur but the counter keeps counting while the button is pressed before a reset. The count output from this circuit only counts in the up direction. Assume the button will not be pressed for more than once per clock cycle.

**17)** Design a circuit that drives four LEDs with a binary count. This circuit has a button that controls the resetting of the circuit, where the button must be pressed for three clock cycles in order for the reset to occur. The count output from this circuit only counts in the up direction. This circuit also has an LED output that turns on for one clock cycle to indicate a reset is occurring.

**18)** Design a circuit that drives four LEDs with a binary count. This circuit has a button that disables the count sequence as follows: if button is pressed for at least three clock cycles, the counter retains the same count or two clock cycles or for as long as the button is pressed before it begins counting up again. The count sequence only counts up, and keeps counting while the button is pressed for less than three clock cycles. .

**19)** Design a circuit that drives four LEDs with a binary up count. This circuit has a button that disables the count sequence for eight clock cycles if the button is pressed. The circuit then continues the counting sequence for at least one clock cycle before entertaining the notion of a delaying for eight clock cycles if the button is once again pressed. Use no more than four states in your design. For the state diagram, show both a Mealy and Moore-type FSM. HINT: use two different counters in this design.

**20)** Show how you can connect two 8-bit up/down counters to create a 16-bit up/down counter. Don’t use a FSM in this problem.

**21)** Design a 4-bit binary counter circuit. The counter counts up continuously when either of the circuits’ two buttons are pressed, otherwise the counter counts down. Don’t use a FSM in this problem

**22)** Design a 4-bit binary counter circuit. The counter counts up continuously when one and only one of the circuits two buttons are pressed, otherwise the counter counts down. Don’t use a FSM in this problem

**23)** Design a 3-bit binary counter that counts in the following sequence if a button is pressed: (…3,4,5,6,7,3,4…). If the button is not pressed, the circuit counts in the following sequence: (…1,2,3,4,5,6,7,1,2,3,4…).Don't use a FSM in this problem

**24)** Design a 4-bit binary counter circuit that counts in one of the two sequences: (0,6,7…,12,13,14,15,0,6…) or (0,3,4,5,6,7…,12,13,14,15,0,3,4,5…). The circuit switches back and forth between the two sequences. The circuit also has a button that when pressed, makes the circuit's count output zero but internally keeps track of the internal counting . The state diagram does not need to represent the 16 states of the count.

**25)** Design a circuit that outputs the following sequence: (…0,1,2,3,4,5,6,0,1…).

**26)** Design a circuit that outputs the following sequence: (…6,7,8,9,10,11,12,6,7….).

**27)** Design a circuit that outputs one of the two following sequences: (…4,5,6,7,8,9,10,4,5…) or (2,3,4,5,6,7,8,9,10,2,3…). The circuit outputs the first sequence if the button is pressed or the second sequence otherwise.

**28)** Design a circuit that outputs one of the two following sequences: (…4,5,6,7,8,9,4,5…) or (5,6,7,8,9,10,11,12,5,6…). The circuit outputs the first sequence if the button is pressed or the second sequence otherwise.

**29)** Design a circuit that outputs the following two sequences: (…8,9,10,11,12,13,14,15,0,8…) or (…10,11,12,13,14,15,0,10…). The circuit outputs one sequence and then pauses on the count of zero for at least two clock cycles or until a button is pressed (whichever is shorter) before it starts counting the next sequence. The output continuously cycles through the two sequences.

**30)** Design a circuit that outputs the following three sequences: (…8,9,10,11,12,13,14,15,0,8…), (…10,11,12,13,14,15,0,10…), and (…7,8,9,10,11,12,13,14,15,0,7,8…). The circuit outputs one sequence and stops at a zero count until a button is pressed. The output continuously cycles through the three sequences. Design a circuit that continually outputs the following two sequences one after the other. This circuit has four input switches that form a binary number. The sequences are (0,7,8,9,10,11,12,13,14,15,0,7…) and (0,"switch value",…15,0,"switch value",…).

**31)** Design a circuit that continually outputs the following sequence: (…7,1,2,3,4,5,6,7,6,5,4,3,2,1,2,3 …).

**32)** Design a circuit that displays the sum of two 8-bit unsigned binary inputs. The circuit also outputs the number of times the two input values generate a carry. The count of the number of carrys is based on what values are being summed on the rising clock edge of the circuit; the maximum value of this count is 255 (the counter rolls over to 0 automatically). If the summation of the two inputs generates a carry, the sum output shows 0xFF; otherwise it displays the sum.

**33)** Design a circuit that displays the result of one of the following addition operations: A+B, B+C, A+C, B+B. The choice of which result it displays is based on the values of two switches; SW1 & SW0, where "00" chooses A+B, "01" chooses B+C, etc. Any operation that generates a carry outputs the value of 0xFF instead of the result of the addition and also turns on an LED. The LED remains off when there is no carry generated. Consider A, B, C, and the result of the additions to be 8-bit unsigned binary values.

**34)** Design a circuit that displays the result of one of the following addition operations on consecutive clock cycles: A+B, B+C, A+C, B+B. Any operation that generates a carry outputs the value of 0x00 instead of the result of the addition and also turns on an LED. The LED remains off when there is no carry generated. The circuit initially displays 0xFF until the a button press that starts this display sequence; the state of the LED does not matter in this state. Once the circuit completes the display sequence, the circuit waits for another button press before starting the sequence again. Consider A, B, C, and the result of the additions to be 8-bit unsigned binary values.

**35)** Design a circuit that displays the result of one of the following addition operations on consecutive clock cycles: A+B, B+C, A+C, B+B. If any of the operations generate a carry, the circuit does not go onto the next operation until the data inputs are such that they do not generate a carry. Any operation that generates a carry outputs the value of 0x00 instead of the result of the addition and also turns on an LED. The LED remains off when there is no carry generated. The circuit initially displays 0xFF until a button press that starts this display sequence; the state of the LED does not matter in this state. Once the circuit completes the display sequence, the circuit waits for another button press before starting the sequence again. Consider A, B, C, and the result of the additions to be 8-bit unsigned binary values.

**36)** Design a circuit that, at the press of a button, outputs A+B (addition) for eight clock cycles, then outputs A+C for eight clock cycles. This circuit also has two other outputs in addition to the sums: one output indicates which operation is being output, the other signal indicates when the operation generates a carry. If a carry is generated, no need to do anything to the sum output. The circuit outputs zero as it waits for a button press, otherwise it always outputs the sum. Assume inputs values are in 10-bit unsigned binary format.

**37)** Design a circuit that, upon the press of a button, continuously outputs A+B (addition) for as many clock cycles are required to generate 16 operations without a carry, then does the same thing for a set of A+C calculations. This circuit also has two other outputs in addition to the sums: one output indicates which operation is being output, the other signal indicates when the operation generates a carry. If a carry is generated, the circuit outputs zero. The circuit should output zero while waiting for a button press. For this problem, consider the inputs and sum output to be 16-bit unsigned binary numbers.

**38)** Design a circuit that counts the number of time the input A is two greater than the B input on the clock edge, where both A & B are 8-bit unsigned binary number. The count automatically rolls over after logging the 127 occurrence. The output of this circuit is the count total.

**39)** Design a circuit that counts how many times B>A on the circuit's active clock cycle edge. The count goes up to 50. When the count reaches 50, the circuit clears the counter and counts the number of times A>B. A & B are 10-bit unsigned binary numbers. The circuit outputs the count. The circuit does this continuously.

**40)** Design a circuit that counts the number of times A=B, then A<B, then A>B, on an active clock edge. This circuit switches modes when the counter reaches 50 and also clears the counter. A & B are 10-bit unsigned binary numbers. The circuit's output are the count value, and a single-bit output for each of the three tests (A=B, A<B, A>B).

**41)** Design a circuit that counts the number of times the sum of A+B is both valid and negative. This count resets when it reaches 50 or 40, depending on whether a button is pressed (pressed==50). Once the circuit reaches it the maximum count, it holds that count until a different button is pressed, at which time it clears the counter. The circuit does not start counting again until the circuit's other button is pressed. Consider A & B to be 8-bit signed binary values in RC format. The only output of the circuit is the count value.

**42)** Design a circuit that shows a count of how many times a given number on the input has repeated itself based on the value of the input on the active clock edge. The input number is an 8-bit unsigned number. The circuit starts at the press of a button and runs forever without getting tired. The output shows the number of repeats from 0-127; this count rolls over automatically after the 127th consecutive value is detected on the input on the active clock edge.

**43)** Design a circuit that counts how many times it detects the sequences "1001" and "11011" on a single serial input. When it detects the former sequence, it increments an 8-bit counter; when it detects the latter sequence, it decrements the same 8-bit counter. The most straight-forward approach is to use two FSMs in your; be sure to provided state diagrams for any FSM you use in your design. Assume the serial input does not change more than once per clock cycle. Allow the counter to underflow or overflow as needed. If both sequences are found simultaneously, the counter does not change its output. Both sequence detectors are the resetting type.

# 27 Shift Registers

## 27.1 Introduction

Registers come in many forms: this chapter deals with one a common form: the shift register. As the name implies, shift registers are registers with special functionality. This chapter introduces shift registers along with their basic applications.

**Main Chapter Topics**

**SHIFT REGISTERS:** This chapter describes various flavors of shift registers and their basic implementations. This chapter also describes several special aspects of shift register including rotates, barrel shifts, and arithmetic shifts.

**Chapter Acquired Skills**

- Be able to describe the shift register's special mathematical operations
- Be able to use the various vernacular associated with shift registers
- Be able to describe shift types including barrel shifts, arithmetic shifts, and rotates
- Be able to describe the various control inputs of shift registers
- Be able to use shift registers in solutions to digital design problems

## 27.2 Shift Registers: Another Specialty Register

A shift register is another type of register. Shift registers, and their various flavors, are useful devices because of their ability to quickly perform a small but useful subset of mathematical operations.

We can decompose a shift register down to its most basic component, which we refer to as a shift register cell. This cell is a storage element, which we model as a D flip-flop. Figure 27.1 shows a schematic diagram of a generic shift register. Upon further inspection, you should discern the following:

- We can model the n-bit shift register as a set of "n" specially connected D flip-flops. The D flip-flops in the shift register share the same clock signal.
- The difference between simple registers and shift registers is in the way that the individual storage elements connect to each other. While simple registers have D flip-flops that receive data from the inputs, the shift register's storage elements receive data from interconnections between individual storage elements. Figure 27.1 shows that the output of one flip-flop becomes the input to the adjacent flip-flop in the shift register, which allows the device to "shift".
- The number of bit storage elements in a shift register defines shift registers. The shift register in Figure 27.1 represents a generic model of a shift register including the magic ellipsis in strategic locations. Common descriptions of shift registers include "a 4-bit shift register" or "an 8-bit shift register", etc. Figure 27.1 shows a generic "n-bit shift register".

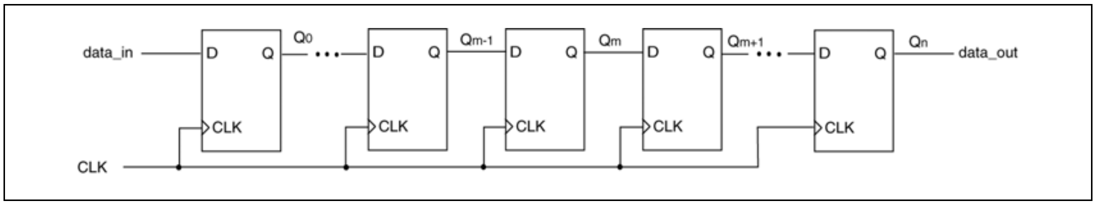
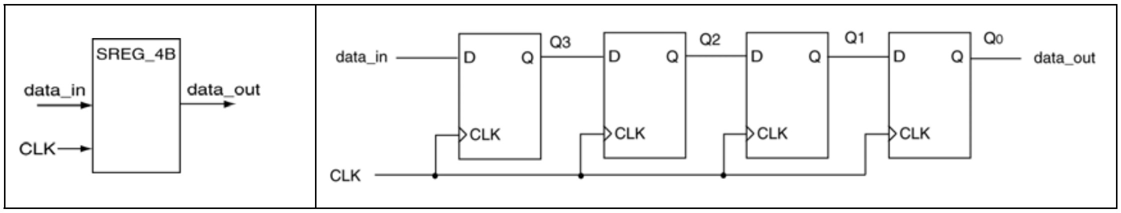


**Figure 27.1: A typical n element shift register.**

Figure 27.2(a) shows a schematic diagram of a 4-bit shift register while Figure 27.2 (b) shows a model of the underlying circuitry. Figure 27.3 shows an example timing diagram for a 4-bit shift register in Figure 27.2(b). Figure 27.3 contains annotations to help with the following description.



**(a)** **(b)**

**Figure 27.2: A block diagram for a 4-bit simple register (a) and a model of the underlying circuitry of a 4-bit shift register (b).**

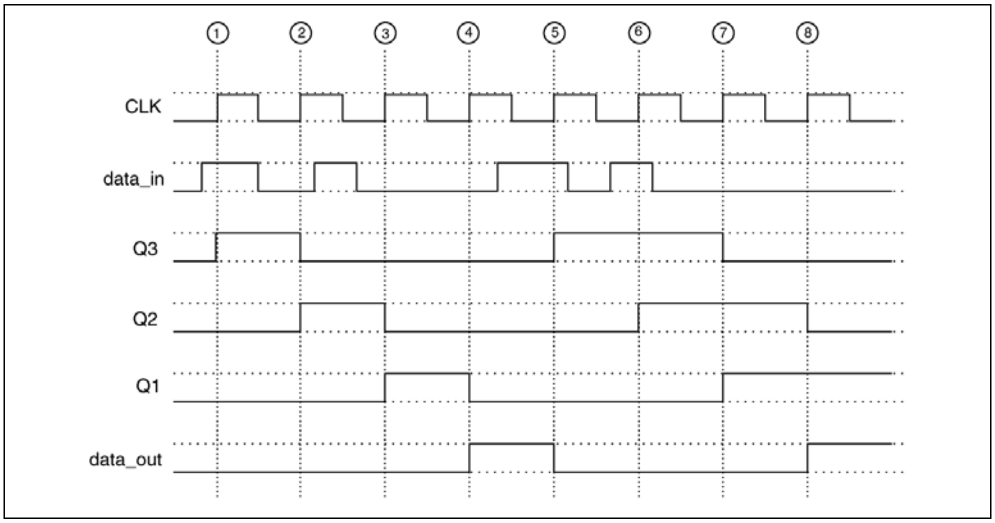


**Figure 27.3: An arbitrary timing diagram associated with the shift register of Figure 27.2(b).**

- This is a 4-bit shift register, meaning that the shift register circuitry contains four storage elements. Figure 27.2(a) shows a BBD for a 4-bit shift register.
- The schematic in Figure 27.2(b) labels each of the internal shift register signals to help describe the operation of the basic shift register in Figure 27.3.

- The "**Qx**" notation indicates the bit positions of the storage elements in the shift register. We consider **Q3** the higher order bit while **Q0** (or data_out) is the lowest order bit[1]. Note that **data_out** and **Q0** are the same signal.
- We consider shift registers to "shift" in either direction; that is, they shift to the left ("shift left") or shift to the right ("shift right"). Figure 27.2(b) shows a right-shifting shift register.
- The notion of this circuit shifting is primarily a term of convenience and not altogether accurate. The "thing" being shifted in Figure 27.2(b) is the "data". Another way to view this is that the circuit inputs 1's and 0's from the left side of the circuit and passing them through to the right side.
- Since this is a sequential circuit, the storage elements have a state associated with them. For the timing diagram of Figure 27.3, the initial state of each storage element is '0', which is arbitrary.
- Since the storage elements are D flip-flops, they 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. 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.
- The "data_in" input can change at various times; it only has an effect on the active clock edge.

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 view 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. The right-shift operation (one shift in the right direction) is the same thing as a divide-by-two operation with truncation[2].

Another issue that usually surrounds shift registers is the notion of cascadeabilitly. If you're unfortunate enough to use shift registers on discrete ICs, you may need to use a bunch of them to obtain the data width that you need. For example, if you need a 64-bit shift register, and all you have to work with are ICs containing 8-bit shift registers, you'll need to cascade[3] eight 8-bit shift registers in order to create a 64-bit shift register.

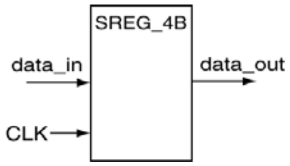
[1] Keep in mind that we often use shift registers for mathematical operations; numbers generally have weights associated with the bit positions (unless you're a cave-person).

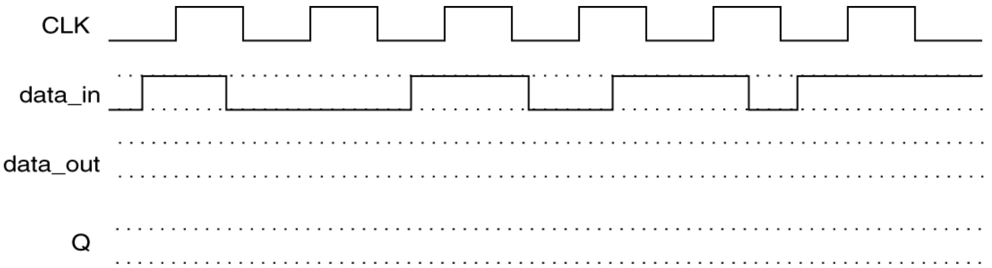
[2] Truncation means the lowest order bit is lost; a similar operation is "round-up" where the value of the lowest order bit is "taken into account" and your weeds are killed at the same time.

[3] In this context "cascade" is a fancy way of saying "connect up the part properly".

**Example 27.1: A Simple Shift Register Timing Diagram**

Using the block diagram on the right to complete the timing diagram provided below. Consider the circuit to be a 4-bit right shifting shift register 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 problem asks for what is 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). Shift registers typically provide all stored data bits as outputs.

Figure 27.4 shows the solution to this example. Here are a few items to note:

- 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.
- The fact that the shift register is dividing the current shift register contents by two (with truncation) when the **data_in** signal is a '0'.

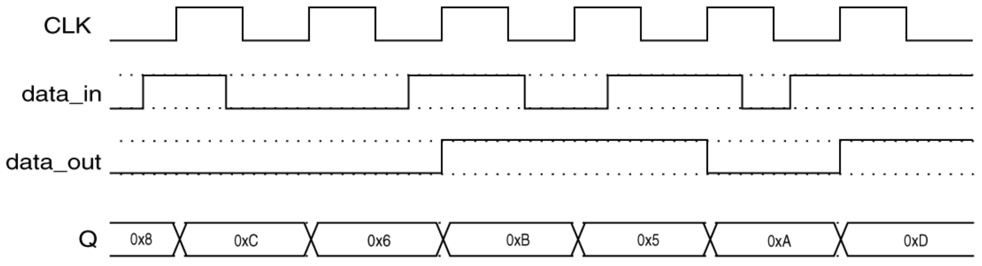


**Figure 27.4: The solution to this example.**

## 27.3 Universal Shift Registers

Shift registers that only shift in one direction are not overly; typical shift registers perform other operations such as shift left, shift right, parallel load, parallel clear, hold, etc. The term in digital design for shift registers containing many features is *universal shift register*, or *USR*. 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. You must consult the datasheet or designer as to what exactly. The following example is a USR with arbitrary functionality.

**Example 27.2: A Simple Universal Shift Register**

Provide a model for an 8-bit universal shift register that supports the following operational characteristics, hold, shift right, shift left, and parallel load. For this problem, assume that all shift register operations are synchronous (meaning they 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.

**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 wondering. All of these operations are synchronous.

- **Hold**: The shift register's contents do not change state on active clock edge.
- **Shift Right**: A typical shift right operation; there needs to be a single-bit input to become the next left-most bit.
- **Shift Left**: A typical shift left operation; there needs to be a single-bit input to become the next right-most bit[4].
- **Parallel Load**: Implies that there needs to be an 8-bit bundle input that simultaneously loads all shift register elements.

From these 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 in Figure 27.5. Here is some other fun stuff.

- The shift register has four unique operations: hold, shift-right, shift-left, and parallel load. This means we somehow need to control which operation occurs. We do this by adding a control signal that "selects" the desired operation. This signal is an input to the shift register and allows some external device to control the shift register. Since the shift register has four operations, we need a two-bit control signal that selects the desired operation.
- We know all the inputs and outputs to the shift register. The problem states that the outputs 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 input.

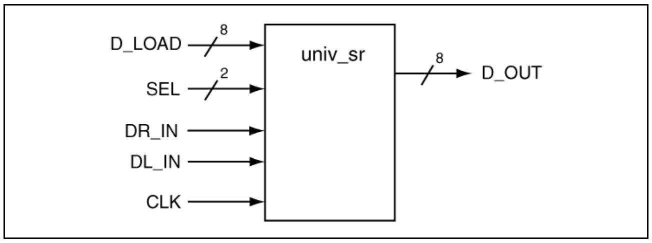


**Figure 27.5: A black box diagram of the universal shift register.**

Figure 27.6 repeats Figure 27.1 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 decide upon what value loaded on the next active clock edge.

[4] 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". We won't do this again.

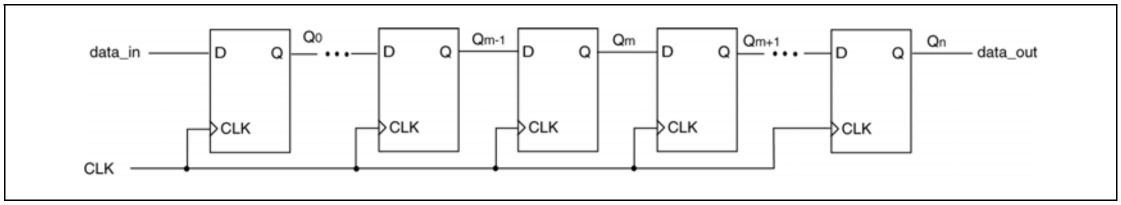


**Figure 27.6: A typical n element shift register.**

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 loaded to the storage element. Each shift register storage element needs to decide which signal loads into the element. Figure 27.7 shows the schematic for the single shift register storage element that you're probably imagining.

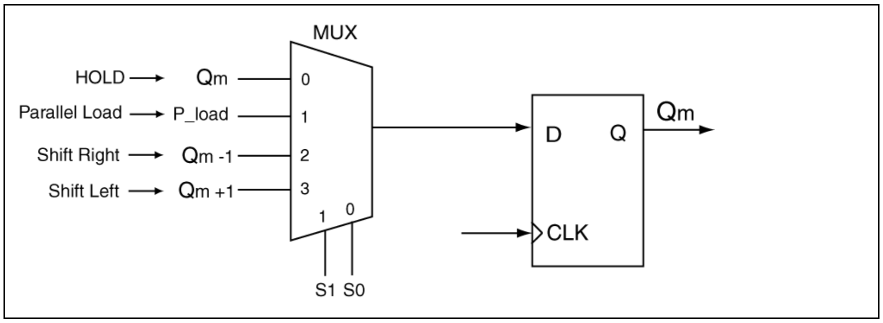


**Figure 27.7: A shift register element with an attached MUX for data selection.**

Figure 27.7 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. We describe the MUX data signals in Figure 27.7 with the following. Table 27.1 summarizes the information in the previous list.

- Qm (0): The input to the D flip-flop is the current output of the current storage element. Qm is an internal signal and ensures that the storage element does not change state by "reloading" its current value, so the present state of the D flip-flop becomes the next state.
- P_load (1): The D flip-flop is a bit from the parallel loading bundle input.
- Qm-1 (2): The input is part of a shift right operation, which indicates the input to this storage element is the first storage element to the left of this storage element (check out Figure 27.6 for the details on 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 27.6 for clarification).

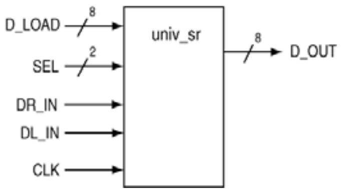
| S1 | S0 | D | Comment |
|---|---|---|---|
| 0 | 0 | Qm | hold |
| 0 | 1 | P_load | parallel load |
| 1 | 0 | Qm-1 | shift right |
| 1 | 1 | Qm+1 | shift left |

**Table 27.1: Summary of the SR element functionality.**

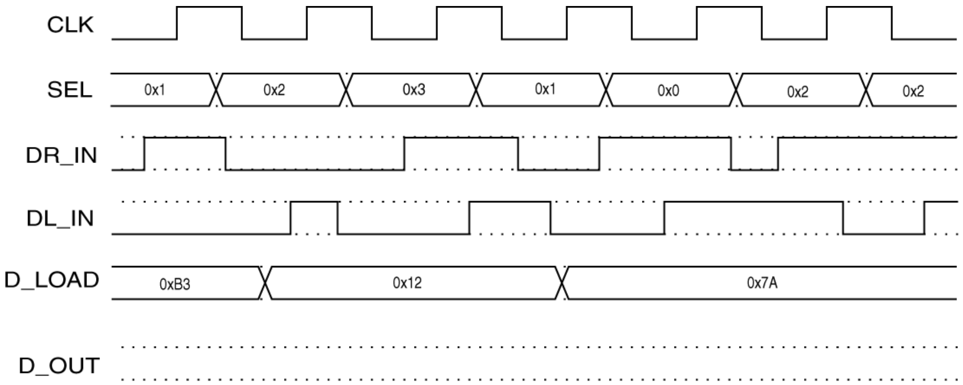
We could proceed with at this level, but let's instead call this example down and bump up to a higher level of abstraction on later problems.

**Example 27.3: Universal Shift Register Timing Diagram**

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 rising edge triggered
- Propagation delays are negligent.
- Initial D_OUT value is 0x45



**Solution**: The first step 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.

A good approach to problems such as these is to list what actions the **SEL** signal is selecting throughout the timing diagrams. Figure 27.8 shows a partially annotated timing diagram highlighting the operations selected by the SEL signal. We synchronize all of these annotations with the rising clock edge.

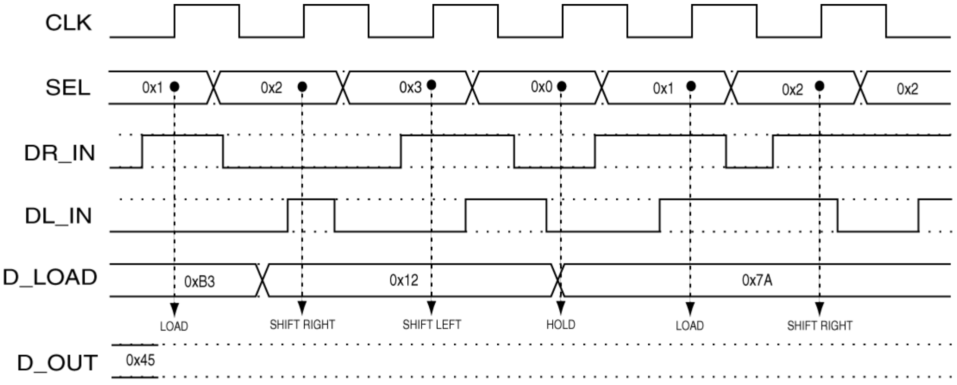


**Figure 27.8: A black box diagram of the universal shift register.**

Figure 27.9 shows the final timing diagram. Most of the changes in the **DR_I**N, **DL_IN**, and **D_LOAD** signals have no effect on the final output.

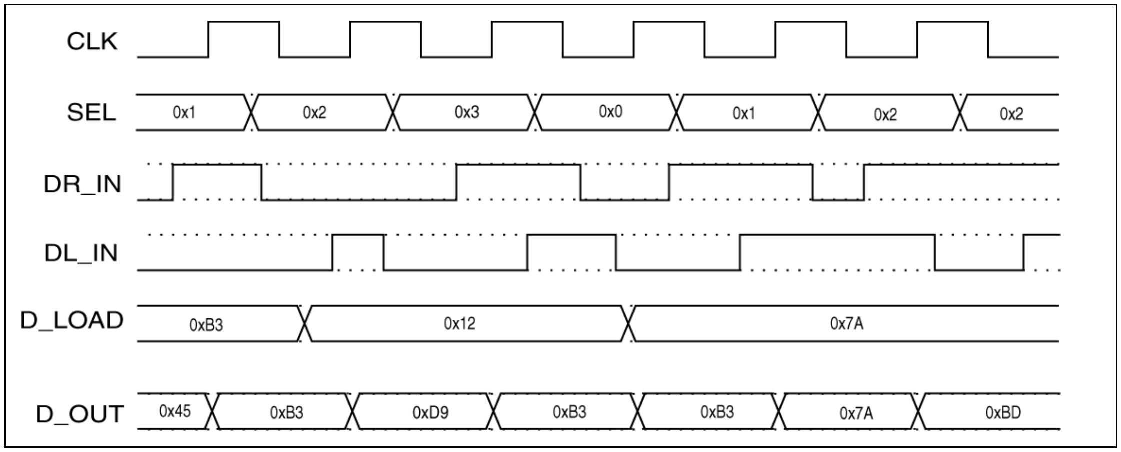


**Figure 27.9: A black box diagram of the universal shift register.**

## 27.4 Barrel Shifters

Another operation associated with shift registers is a “barrel shift”. While simple shift registers only performed one shift per clock cycle, barrel shifters are capable of performing more than one shift per clock cycle.

We can consider shifting left and right as forms of multiplying (left shift) or dividing (right shift) by two. Thus, barrel shifters are associated with multiplying and dividing by “powers of two” (such as 4, 8, 16, 32, etc.). These operations provide super-fast (namely, one clock cycle) multiply and divide operations. Multiplying and dividing binary numbers is time consuming relative to other computer operations (such as logic operations); barrel shifters provide a fast, but limited alternative.

We use barrel shifters in arithmetic applications where we do not require 100% accuracy of results primarily due to truncation of shift right operations. 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 retaining the full precision of values. Using integer math generally causes a loss in precision, but the increase in speed is desirable so long as the loss in precision is tolerable.

Table 27.2 shows two examples 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 2-bit right barrel shift that arbitrarily inputs 0’s on the left side of the register. The (b) row shows a 2-bit left barrel shift that arbitrarily inputs 1’s from the right side of the register.

| | Description | Example |
|---|---|---|
| **(a)** | barrel shift right 2x; stuff in a two 0’s from the left side. | 0 → 1 0 1 1 0 1 0 0<br>0 0 1 0 1 1 0 1 |
| **(b)** | barrel shift left 2x; stuff in a two 1’s from the right side. | 1 0 1 1 0 1 0 0 ← 1<br>1 1 0 1 0 0 1 1 |

**Table 27.2: Examples of barrel shifting operations.**

The examples in Table 27.2 are arbitrarily 2-bit barrel shifts. The barrel shifter is “shifting two times” in one clock cycle. There is only one shift, which implies there are connections between each shift register element and the element that is two shift register elements away. The barrel shifter thus requires the proper signal routing in order to accomplish this shift, so barrel shifters are typically to only a few shift lengths because routing resources are expensive in digital-land.

## 27.5 Other Common Shifts

Two more common shifting operations are rotates and arithmetic shifts. These operations are also simple in their basic states. Rotate operations can be useful in many applications, though there is not one slam-dunk great example I can think of; in theory, these operations fall into the category of "bit tweaking". Arithmetic shift operations are similar to simple shift operations but they work correctly with signed binary numbers.

### 27.5.1 Rotates

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 register are lost by "shifting them out" of the register as is the case with simple shift registers. Specifically, for a rotate right 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. Table 27.3 show examples of rotate left and right operations on an 8-bit register.

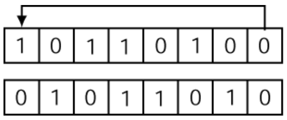
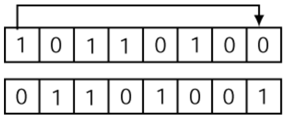
| | Description | Example |
|---|---|---|
| (a) | rotate right; the LSB is transferred to the MSB; | <br>1 0 1 1 0 1 0 0<br>0 1 0 1 1 0 1 0 |
| (b) | rotate left; the MSB transfers to the LSB. | <br>1 0 1 1 0 1 0 0<br>0 1 1 0 1 0 0 1 |

**Table 27.3: Examples of rotate-type shifts.**

### 27.5.2 Arithmetic Shifts

Arithmetic shifts are similar to simple shifts; the key difference is that arithmetic shifts work with signed binary number and preserve the "signedness" of the value they operate on. For an arithmetic shift left operation, the value of the sign bit does not change because 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, we both retain the sign bit as a sign bit and propagate the sign bit to the right with each shift.

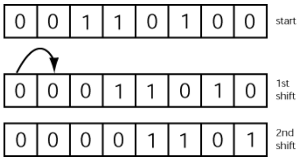
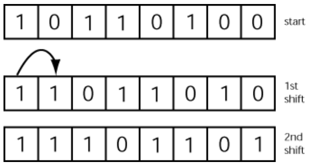
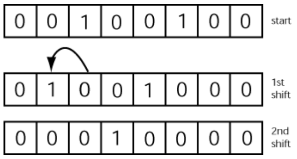
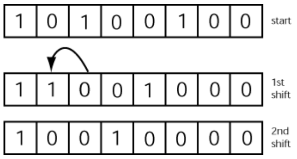
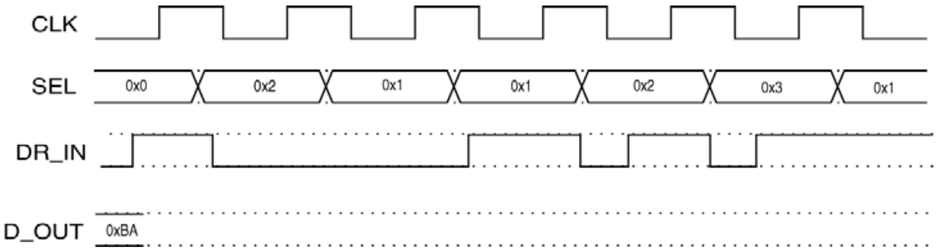
| | Description | Example |
|---|---|---|
| **(a)** | An arithmetic shift right of a positive number in RC format; the operation copies the sign bit from sign-bit position to the adjacent bit on the right with each shift. This is a divide by two on a positive signed number. | <br>0 0 1 1 0 1 0 0 start<br>0 0 0 1 1 0 1 0 1st shift<br>0 0 0 0 1 1 0 1 2nd shift |
| **(b)** | An arithmetic shift right of a negative number in RC format; the operation copies the sign bit from the 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 negative signed number. | <br>1 0 1 1 0 1 0 0 start<br>1 1 0 1 1 0 1 0 1st shift<br>1 1 1 0 1 1 0 1 2nd shift |
| **(c)** | An arithmetic shift left on a positive value in RC format; the left shift does not alter the sign; all other bits shift left and the operation arbitrarily stuffs a ‘0’ into the LSB. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a positive signed number. | <br>0 0 1 0 0 1 0 0 start<br>0 1 0 0 1 0 0 0 1st shift<br>0 0 0 1 0 0 0 0 2nd shift |
| **(d)** | An arithmetic shift left on a negative value in RC format. The left shift does not alter the sign bit; all other bits shift left and the operation arbitrarily stuffs a ‘0’ into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a negative signed number. | <br>1 0 1 0 0 1 0 0 start<br>1 1 0 0 1 0 0 0 1st shift<br>1 0 0 1 0 0 0 0 2nd shift |

**Table 27.4: Examples of many flavors of arithmetic shifts.**

**Example 27.4: A Shifting and Rotating Circuit**

Using the following specifications, complete the provided timing diagram. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. 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 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 27.10 shows the final block diagram for this example problem.

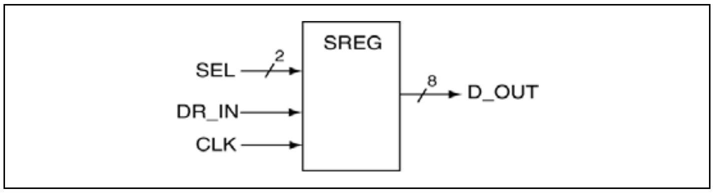


**Figure 27.10: A black box diagram of the universal shift register of** Example 27.4**.**

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 27.11 shows this intermediate helper step.

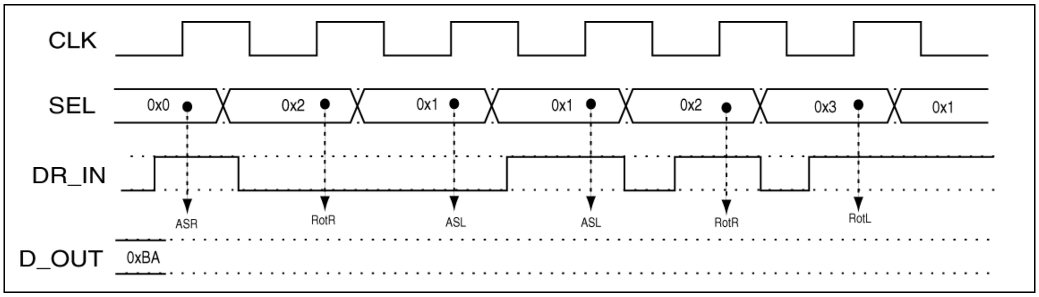


**Figure 27.11: A black box diagram of the universal shift register of Example 27.4.**

Without too much verbage, Figure 27.12 shows the final timing diagram solution to Example 27.4. One thing to note about this problem is that the circuit only uses the DR_IN input for arithmetic shift left operations.

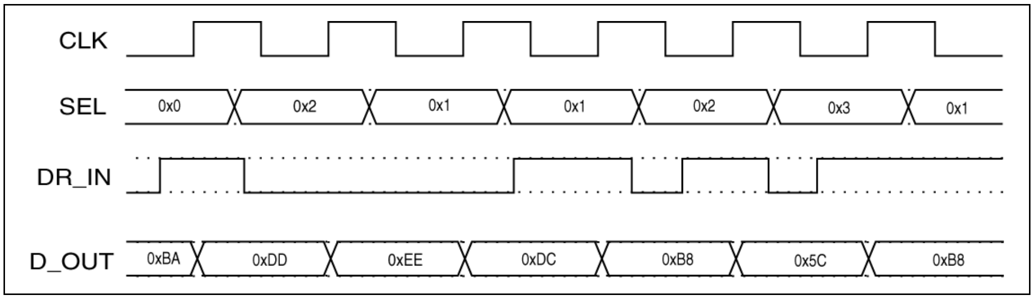
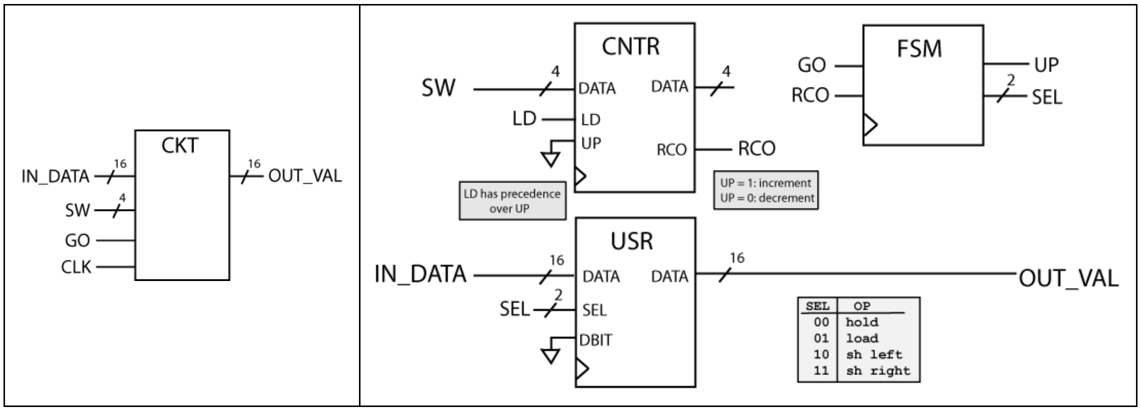


**Figure 27.12: A black box diagram of the universal shift register of Example 27.4.**

**Example 27.5: Design #1: Pre-Determined Shifting Circuit**

Design a circuit that shifts a 16-bit value left the number of times indicated by the value on the circuit's four switches (a binary value). When this circuit sees an asserted **GO** signal, the operation begins on the circuit's 16-bit input. Shift operations should input 0's into the circuit. The circuit's output is also a 16-bit value. The final value remains on the circuit's output until another assertion of the **GO** signal. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit's FSM. Also, state the forms of control the circuit uses. Minimize your use of hardware in the solution.

**Solution**: The problem description does not provide a BBD, so the first step is to generate one from the problem description; Figure 27.13(a) shows this first step.



**(a)** **(b)**

**Figure 27.13: A block diagram for circuit (a), and underlying circuitry (b).**

The next step in the solution is to make an inventory of the modules we need in the solution. Here is the thought process:

- The circuit needs to do some shifting, so we know the circuit requires a shift register. Even though we only need to shift in one direction, we use a 16-bit USR in our solution.
- The circuit needs to shift the number of times on the **SW** input (considering the **SW** input as a binary number).
- Since the USR only shifts once per clock cycle, we need some way of keeping track of how many times we shift. This functionality calls out for a counter; what we do is load the switch value into a counter, then decrement the counter until the counter is zero. We know the counter output is zero when the **RCO** output of the counter asserts.
- The circuit elements require a FSM to act as a master controller.

Figure 27.13(b) shows the lower-level BBD for the solution; here are a few of the more important points.

- We connect the **DBIT** input to zero; this is arbitrary, as the problem did not mention it. It's generally the safer approach to connect this input to zero as opposed to connecting it to one.
- We connect the **UP** input of the counter zero, which makes it always count down (we know this because of the annotation included in the BBD). We also include an annotation that states the precedence of the **UP** and **LD** signals of the counter.
- The counter's **RCO** output asserted when the counter reaches its terminal count in the down direction. The **RCO** is a status signal that is input to the FSM.

- The FSM controls the **SEL** control input of the USR. We include annotations of how the **SEL** input controls the USR operations.

Figure 27.14 shows the final state diagram for the solution. Here are the highlights of the state diagram.

- The state diagram has two states; in the “wait” state, the FSM is waiting for an asserted **GO** signal. When the **GO** signal asserts, the FSM transitions to the shift state. We configured the **SEL** and **LD** signals to be Mealy-type outputs, which was arbitrary. Using Mealy-type outputs allows the state diagram to have two states. If we had used Moore-type outputs for **LD** and **SEL**, the state diagram would have three states, accounting for the differences in the **LD** and **SEL** signals.
- We model the **LD** output in the “shift” state as a Moore-type output, which may seem confusing because **LD** was a Mealy-type output in the “wait” state. The final word is that **LD** is a Mealy-type output; we opted to model it as Moore-type output in the “shift” state to simplify the state diagram.
- We model the **SEL** signal as a Mealy-type output in the “shift” state, as its value depends upon **RCO**. As long as **RCO** is not asserted, we want the shift register to shift left and allow the counter to decrement. Once **RCO** asserts, the count is zero, which means we are done shifting. At that point, the state diagram directs the FSM to hold its state.

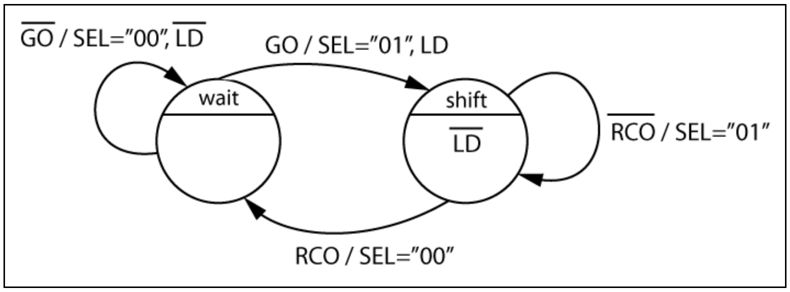

$\overline{GO}$ / SEL="00", $\overline{LD}$
GO / SEL="01", LD
wait
shift
$\overline{LD}$
$\overline{RCO}$ / SEL="01"
RCO / SEL="00"

**Figure 27.14: The state diagram associated with this example.**

Outputs from the FSM connect to the **LD** and **SEL** control inputs in the circuit’s modules. The **UP** control input is hardcoded. This circuit thus uses circuit (the FSM’s outputs) and internal (the **UP** signal) control. The **GO** signal is a form of external control.

**Example 27.6: Design #2: Even Parity Checker Circuit**

Design a circuit that determines if an 8-bit input value is even parity. When this circuit sees an asserted **GO** signal, the operation begins on the circuit’s 8-bit input. The final parity value remains on the circuit’s output until another assertion of the **GO** signal. Don’t use EXOR-type functions or a MUX in this design. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit’s FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

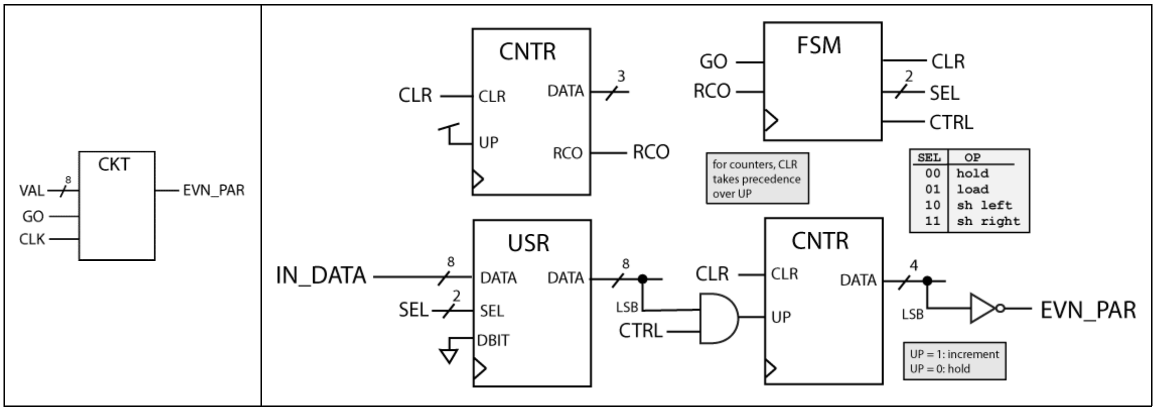
**Solution**: The first step is to generate a top-level BBD from the problem specification; Figure 27.15(a) shows the result of this step.

The next step in the solution is to create an inventory of the modules our circuit requires. The list below shows an example thought process for the solution.

- The problem states not to use XOR functions, which we know is a straightforward way to solve this problem. Our other approach to determining parity is to count the number of 1's in a number; if there are an odd number of 1's, then the number has odd parity; otherwise the number has even parity. We use a counting approach in this problem. This means that we need a counter. Since we are counting a 8-bit value, the maximum count possible is eight, which means this needs to be a 4-bit counter (a 3-bit counter has a maximum value of 7).
- This circuit also needs to counter to eight, which means our solution requires a second counter.
- We need a circuit that helps us do the actual count. There are several ways to do this; we choose a USR for this problem. The trick here is that we use the LSB output of the USR to control the counter's **UP** control input.
- The solution's final module is an FSM to control circuit operations.

Counting the set bits in a binary value is a trivial operation for humans. The issue in this problem is that we need to configure the hardware to do the counting for us. In other words, we have an algorithm that generates the solution; we must implement that algorithm in hardware. Figure 27.15(b) shows the final circuit for this problem; here is some other good stuff to chew on:

- Both counters share the same **CLR** signal. When this algorithm starts, both counters need to start at zero. One counter is counting the number of bits the circuit examines; the other counter is counter the number of set bits in the input value.
- The circuit uses the LSB of the shift register as a part of the **UP** control to the bit counter. We want complete control of the counting operation, which means we want to disable it if we choose, so we connect the LSB of the shift register to an AND gates. The other input to the AND gates is the **CTRL** signal, which gives the circuit the ability to disable the counter's **UP** control input. We do this because the problem states the parity output should be persistent; if we had connected the LSB directly to the **UP**, the counter could remain incrementing after the algorithm completes. Put this in your bag of tricks; it's typical in this flavor of design problems.
- The LSB of the bit count indicates the parity of the input value. The problem wants to know when the parity is even, which it is when the LSB of the bit count is zero. We thus invert the LSB and use that as the **EVN_PAR** output of the circuit.
- The BBD provides two annotations; one regarding both counter (the **CLR** comment) and the other regarding the right-most counter in the BBD. We don't annotate the left-most counter as the **UP** input is hardwired to '1', and there is no confusion as to how the counter reacts to different asserted **UP** values.
- The **DBIT** input of the shift register is hardwired to '0'. This is arbitrary for this problem, but it's a requirement in a later example problem.



**(a)** **(b)**

**Figure 27.15: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 26.16 shows the state diagram for the problem. There are several approaches to the state diagram; we opt for this approach, and provide some meaningful commentary:

- We model the **SEL** input in the as a Mealy-type output in the "wait" state and as a Moore-type output in the "shift" state. The **SEL** input is truly a Mealy-type output; it makes the state diagram more readable to model it as a Moore-type output in the "shift" state.
- We opted to use Mealy-type output for both the **SEL** and **CLR** signals. We could model these two outputs as Moore-type outputs, but that would have required adding an extra state to the state diagram.
- The **CTRL** output is a pure Moore-type output; it makes no sense to model it as a Mealy-type output. In the "wait" state, the **CTRL** always disables the bit counter's **UP** signal; in the "shift" state, the **CTRL** input always allows the LSB of the shift register to increment the counter.
- The shift register in this problem is holding, loading or shifting, the three different values of the **SEL** signal in the state diagram indicate. As with all shift register problems, the first order of business is to load the shift register, which we do when the **GO** input asserts.
- The shift register continues shifting until the **RCO** status signal of the counter asserts. Once the **RCO** signal asserts, the counter shifts one more time. We really don't care about the shift; we only care about the value of the LSB output of the shift register, as that determines whether the counter increments or not.

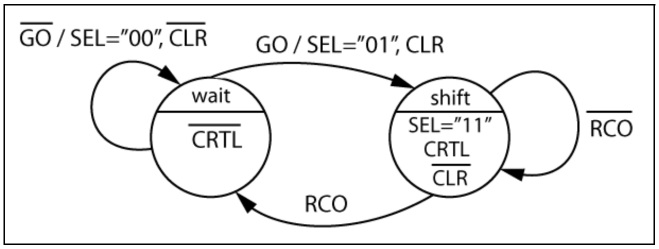


**Figure 27.16: The state diagram associated with this example.**

Outputs from the FSM connect to the **CLR**, **CTRL** and **SEL** control inputs in the circuit's modules. We hardcode the **UP** control input on the left-most counter in the diagram. This circuit thus uses circuit (the FSM's outputs), internal (the **UP** signal) control, and external control in the form of the **GO** signal.

This solution requires a deterministic amount of clock cycles to complete. After the **GO** signal asserts, the first clock cycle transition the FSM to the "shift" state; it stays in the "shift" state for seven clock cycles. On the eight clock cycle, the asserted **RCO** signal causes a transition to the "wait" state. Therefore, this algorithm always requires nine clock cycles.

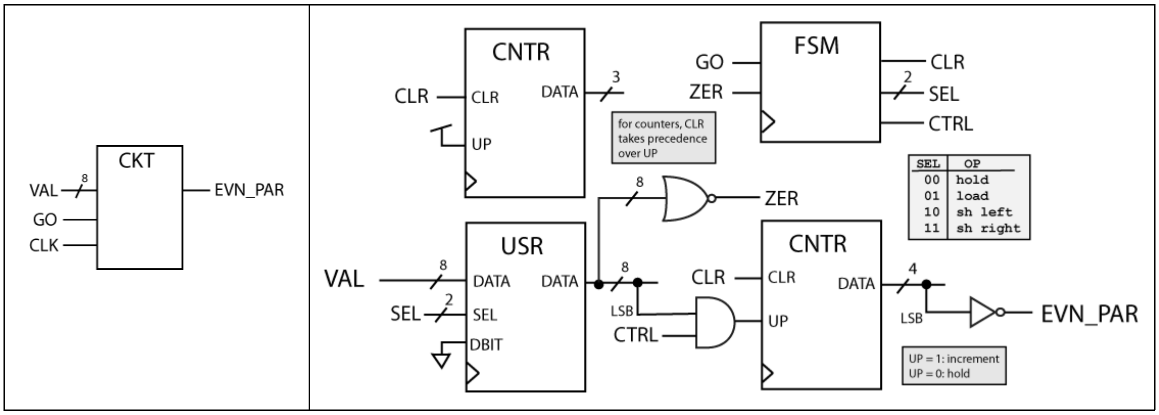
**Example 27.7: Design #3: Even Parity Checker Circuit (Fast Version)**

Design a circuit that shifts determines if the 8-bit input value is even parity. When this circuit sees an asserted **GO** signal, the operation begins on the circuit's 8-bit input. The final value remains on the circuit's output until another assertion of the **GO** signal. Don't use EXOR-type functions or MUXes in this design. Design this circuit so that it obtains the answer as quickly as possible in the average case. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit's FSM. State the forms of control the circuit uses. Make a comment how many clock cycles your circuit requires to generate the correct output. Minimize your use of hardware in the solution.

**Solution**: This is the same problem as the previous example, but now we need to increase the average operation time for the circuit. The top-level BBD is the same as the previous problem; we repeat it in Figure 27.17(a) for your viewing pleasure.

The next step in the solution is to generate an inventory of modules our solution requires. We make a guess here that the modules are the same, and then set out to figure out a way to make the algorithm run faster in the average case. The issue with the previous problem is that, no matter what, the algorithm always requires the same amount of time to complete because the left-most counter always must reach its terminal count before the algorithm terminates. The problem is that sometimes the circuit is sometimes examining only zero bits, which do not affect the final parity count. The solution to this issue is that we can constantly check the USR's output, because we can terminate the algorithm early if the remainder of the bits in the USR are zero. That is that the solution in Figure 27.17(b) does; here are some extra notes in addition.

- We no longer use the **RCO** from the bit counter as an indicator of when to terminate the algorithm; we now examine all the bits currently stored in the USR. When all the bits are zero, we terminate the algorithm. We can achieve this functionality using an 8-input NOR gate. We use a shorthand notation in Figure 27.17(b); this saves us drawing an eight-input NOR gate. The output of the NOR gates is the **ZER** signal, which is the new terminal count output.
- For this problem, we must connect the USR's **DBIT** input to '0'; out new approach to this problem would not work otherwise.



**(a)** **(b)**

**Figure 27.17: A block diagram for circuit (a), and underlying circuitry (b).**

Figure 27.18 shows the state diagram for our solution. This state diagram is identical to the previous solution except for the fact that **ZER** replaces the **RCO** signal.

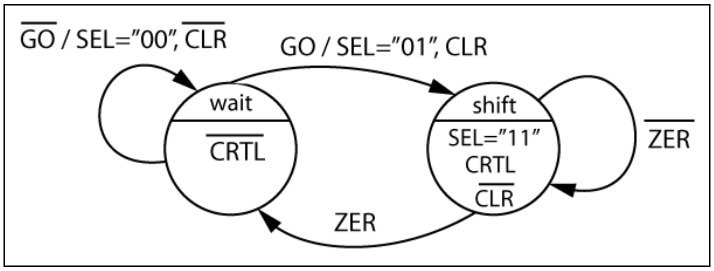


**Figure 27.18: The state diagram associated with this example.**

Outputs from the FSM connect to the **CLR**, **CTRL** and **SEL** control inputs in the circuit's modules. We hardcode the **UP** control input on the left-most counter in the diagram. This circuit thus uses circuit (the FSM's outputs) and internal (the **UP** signal) control. The **GO** signal is a form of external control

The number of clock cycles required to generate a solution depends upon the input data in this problem. It many take as few as two clock cycles (if the input value is zero) or as many as nine clock cycles (if the MSB of the input values is set). We consider this a better solution because the algorithm runs in fewer clock cycles in the average case.

## 27.6 Digital Design Foundation Notation: Shift Register

We consider the shift register a Digital Design Foundation module. The shift register is a controlled circuit. We consider all shift register operations synchronous, except for the **CLR** input, which is sometimes asynchronous. Because shift registers are straightforward to model in with an HDL, we typically only include (or connect) inputs and outputs as we need them. The width of the SEL input sufficient to support the shift register's operations. Figure 27.19 shows the foundation module for a shift register.

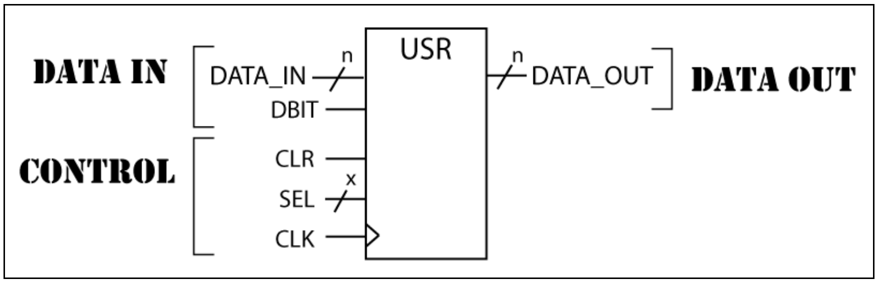


**Figure 27.19: Typical data, control and status signals for a universal shift register.**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **DATA_IN** | A counter is a register, so it can typically loaded data in to the counter's storage elements. The DATA_IN input is the data that is loaded to the counter. |
| | **DBIT** | The bit that becomes the left-most bit for a right shift operation or the right-most bit for a left-shift operation |
| OUTPUT DATA | **DATA_OUT** | The DATA_OUT signal is the data currently being stored in the counter's storage elements. |
| CONTROL | **CLK** | Registers are synchronous circuits; most operations are synchronized with the active edge of the clock signal. |
| | **CLR** | Latches 0's into the register's storage elements; can be synchronous or asynchronous. |
| | **DBIT** | The bit that shifts into the register on shift operations, which is the new left-most bit or the new right-most bit for shift right and shift left operations, respectively. |
| | **SEL** | These bits select the operation the shift register performs. These operations could include: shift left, shift right, hold, load, rotate left and/or right, barrel shifts, etc. The width of this input depends on the number of possible operations. |
| STATUS | **n/a** | - |

**Table 27.5: The foundation description for a universal shift register.**

## 27.7 Register Overview

The registers we've worked with include several common sequential circuits such as shift registers and counters. The main difference between the many types of register is their feature set. Table 27.6 shows a possible breakdown of the register types and their relation to each other. Many of the features in Table 27.6 can be either synchronous or asynchronous.

| Register Type | Sub-Types | Features |
|---|---|---|
| plain register | - | Synchronous load of input data |
| shift register | universal shift registers, barrel shifters | Synchronous load, preset, clear, load enable, shift left/right, arithmetic shift left/right, hold, rotate left/right, barrel shift, cascadeability |
| counters | up/down counters, decade counters | parallel load, preset, clear, load enable, increment, decrement, cascadeability |

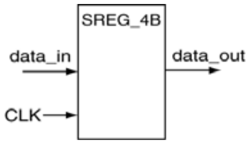
**Table 27.6: The feature progression of the device referred to as a register.**

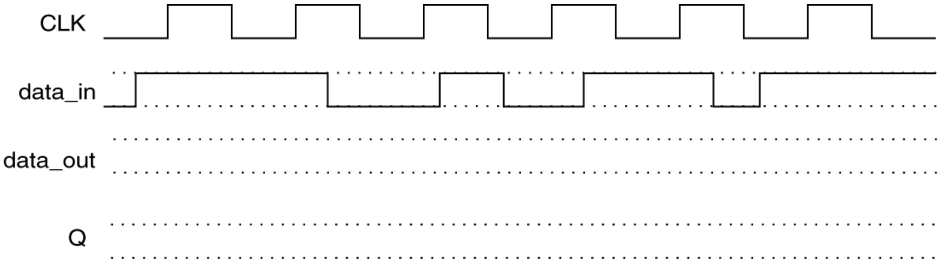
## Chapter Summary

- Shift Registers: Shift registers are in many ways similar to simple registers; 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.
- Rotates: These are similar to shift operations except the register retains all bits from the operation. For rotate right, all bits shift to the right and the LSB becomes the new MSB. For rotate lefts, all bits shift to the left and the MSB becomes the new LSB.
- Arithmetic Shifts: This type of shift retains the sign bit of the register. For right shifts, the sign bit propagates to the right; for left arithmetic shifts, the sign bit does not change and the register loses the bit adjacent to the sign bit.

## 27.8 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:

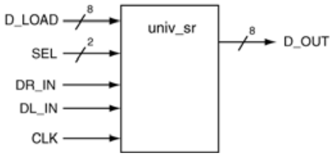
SEL = "00": hold

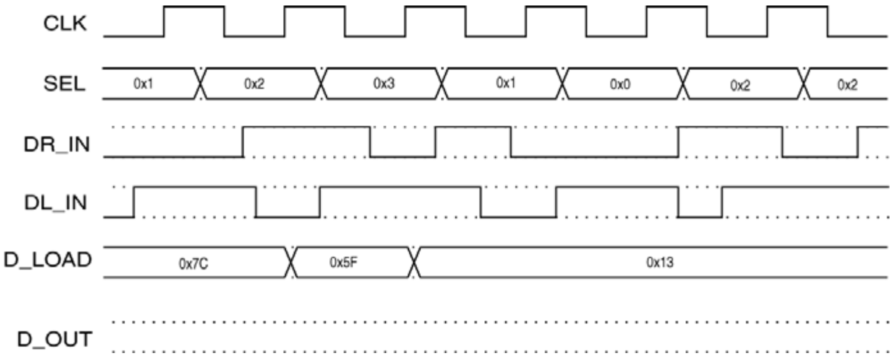
SEL = "01": parallel load of D_LOAD data

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





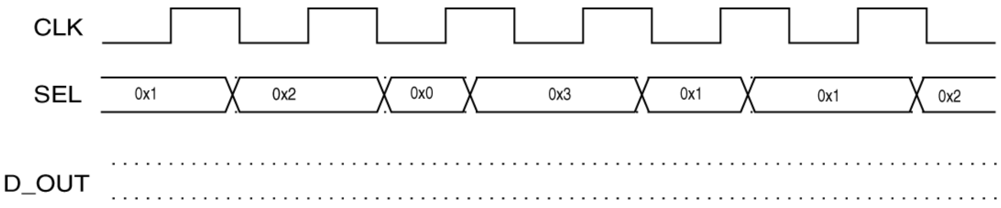
**3)** Complete the following timing diagram. The SEL inputs are the control inputs to an 8-bit universal shift register. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need 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

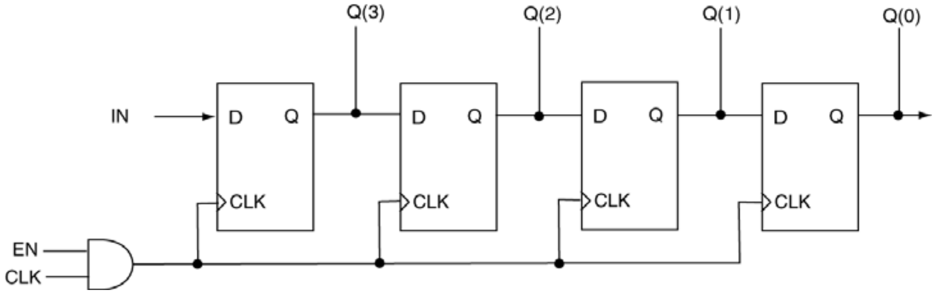
SEL = "01": rotate left

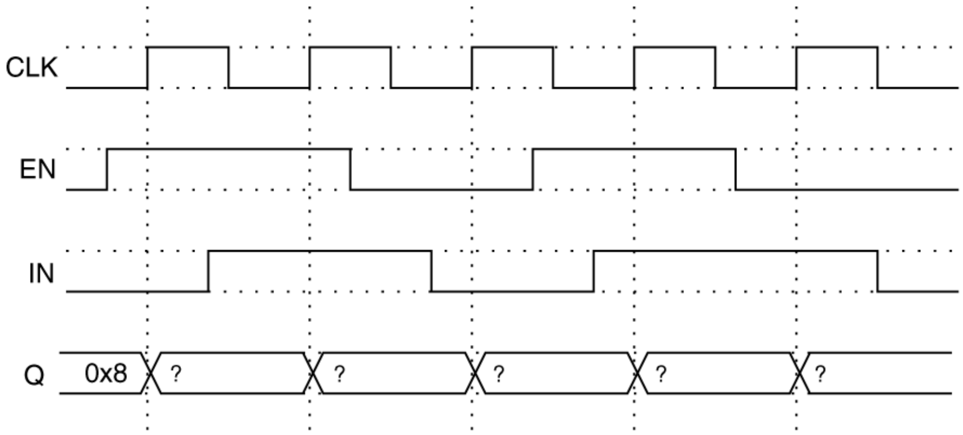
SEL = "10": divide by 8 (bit stuff 0's)

SEL = "11": multiply by 8 (bit stuff 0's)

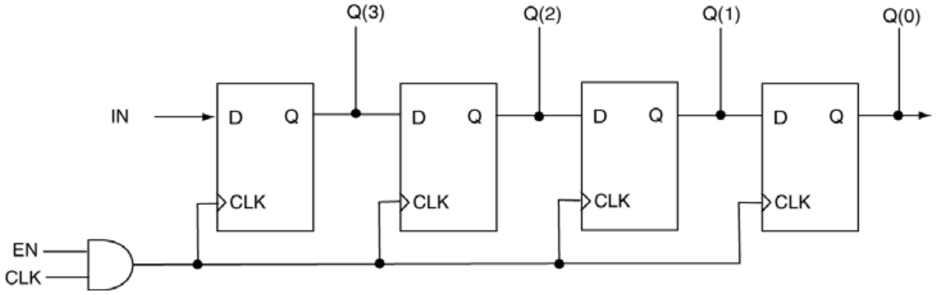


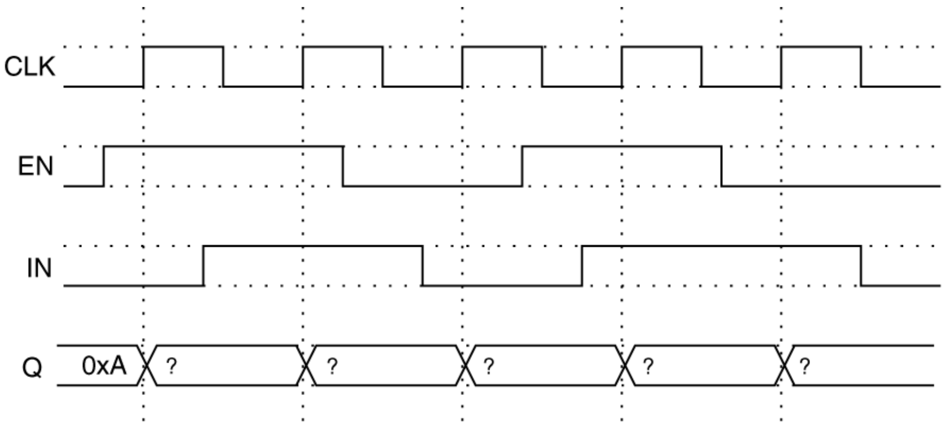
**4)** Use the schematic diagram to complete the **Q** output. The **Q** output is a 4-bit bundle; the starting state of Q is listed in the timing diagram as a hex value (4-bits).. Assume that propagation delays are negligent.





**5)** Use the schematic diagram to complete the Q output. The Q output is a 4-bit bundle; the starting state of Q is listed in the timing diagram as a hex value (4-bits).





## 27.9 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the number of states in the associated state diagrams
- Minimize the use of hardware when problem require extra hardware
- Assume all inputs and outputs are positive logic unless stated otherwise
- Disregard all setup and hold-time issues
- For sequence detector problems assume the X input is stable when each clock edge arrives and that X can change no more than once per clock period.
- State all forms of control for your solution.

**1)** Design a circuit that upon the pressing of a button, outputs the value on the inputs multiplied by eight. The input is an 8-bit unsigned binary value; the output data width is as small as it can possibly be and still represent the largest possible result from this operation.

**2)** Design a circuit that upon a button press, divides an 8-bit unsigned binary input values by two for as many times as required to make the result of the input value less than 17. The result of this operation is also an 8-bit unsigned binary number.

**3)** Design a circuit that outputs a value 2.5 times greater than the input value upon the pressing of a button. Assume the input is an 8-bit unsigned binary value and the width of the output is minimized but the answer is valid. Don't worry about round-up for this problem.

**4)** Design a circuit that outputs a value 5.5 times greater than the input value upon the pressing of a button. Assume the input is an 8-bit unsigned binary value and the width of the output is minimized but the answer is valid.

**5)** Design a circuit that outputs a value 4.75 times greater than the input value upon the pressing of a button. Assume the input is an 8-bit unsigned binary value and the width of the output is minimized but the answer is valid. For this problem, the answer should be available in four clock cycles or less.

**6)** Design a circuit that upon a button press, calculates the parity of a 16-bit value. This circuit has an LED that indicates the number of bits in the input that are set and also a single LED that indicates parity (off for even and on for odd).

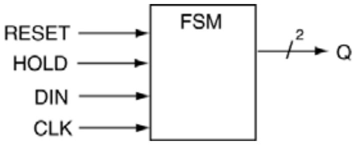
**7)** Design a circuit that upon the pressing of a button, divides an 8-bit signed binary input in RC format by two. For this problem, the circuit needs to make sure the answer is valid in every case. It is OK to truncate your division operation. The result is also an 8-bit value.

**8)** Design a circuit that divides an unsigned 8-bit signed binary input by four upon the pressing of a button. For this problem, apply a standard round-up to the result. The result is also an 8-bit value. Show a BBD of your solution; minimize the amount of hardware in your solution.

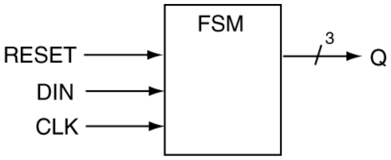
**9)** Design a circuit that can either divide or multiply a 32-bit unsigned binary input by the value of a 4-bit output upon the pressing of a button. The output is also a 32-bit unsigned value. The value that the input value is multiplied by is a two to the power of the 4-bit input; it multiplies if an input signal OP is set, or divides if OP is cleared. Assume that the OP signal does not change after a button press. Don't worry about this problem overflowing or underflowing; truncation for the divide is OK.

**10)** Design a circuit that can either divide or multiply a 32-bit unsigned binary input by the value of a 4-bit output upon the pressing of a button. The output is also a 32-bit unsigned value. The value that the input value is multiplied by is a two to the power of the 4-bit input; it multiplies in a single-bit input one, or divides otherwise. Don't worry about this problem overflowing or underflowing; truncation of the divide is OK. This circuit also has two outputs that indicate when the calculation is complete and if the answer is correct. Assume that the OP signal does not change after a button press.. For this problem, division always output a valid number, but multiplication does not.

**11)** You can use a FSM to model 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.



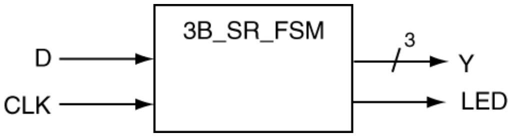
**12)** You can use a FSM to model a shift register. For this problem, provide a state diagram that could be used to model a 3-bit shift register. Consider the Q output to be a 3-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs.



**13)** You can use a FSM to model a shift register. For this problem, provide a state diagram that models a 2-bit shifting left shift register. In addition to DIN (data in) and a CLK (clock) input, this shift register has two asynchronous reset inputs, RST0 & RST1, which place the shift register in "00" & "01" states respectively. The shift register also has a HOLD input that when asserted, keeps the same shift register output in the FSM, but effectively puts the FSM in a hang state. Exiting any hang state is thus done using one of the two asynchronous reset inputs.

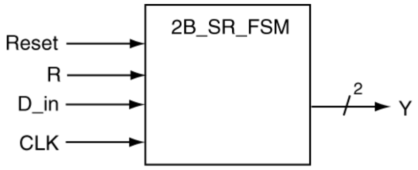
**14)** You can use a FSM to model a shift register. For this problem, provide a state diagram that models a 2-bit shifting left shift register. In addition to DIN (data in) and a CLK (clock) input, this shift register has two asynchronous reset inputs, RST0 & RST1, which place the shift register in "00" & "01" states respectively. The shift register also has a HOLD input that when asserted, causes the FSM to not change state for one clock cycle. After that one clock cycle, the HOLD input is ignored for one clock cycle.

**15)** Provide a state diagram that models the operation of a 3-bit shift register that shifts right on each clock edge. Also, this FSM controls an LED that blinks so long as the shift register value is "111"; the blink rate is equal to half the clock frequency. Assume all set-up and hold-times are met and that circuit delays are negligible.



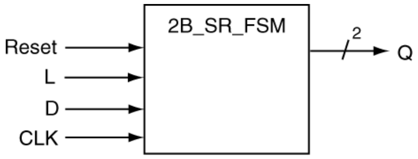
**16)** Provide a state diagram that models the operation of a 2-bit shift register that shifts either right or left. Be sure to state any assumptions you make for this problem. *Minimize the number of states in your design.*

- The D_in input represents the value that is shifted into the shift register for both shift directions
- The Reset signal is an asynchronous input that places the input into the "00" state
- The Y output is the value of the shift register
- If the R input is a '1', the FSM shiftS right; otherwise the shift register left.



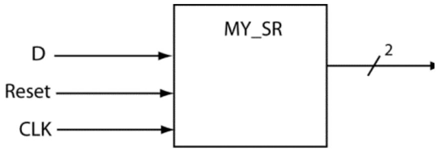
**17)** Provide a state diagram that models the operation of a 2-bit shift register that shifts either right or left. Be sure to state any assumptions you make for this problem. *Minimize the number of states in your design.*

- If the L input is a '1', the FSM shifts left; otherwise the shift register shifts right
- The D input represents the value that is shifted into the shift register for both shift directions
- The Reset signal is an asynchronous input that places the input into the "11" state
- The Q output is the value of the shift register



**18)** Provide a state diagram that models the operation of a 2-bit left shifting shift register. For this design, the shift register should ***not*** have the same output for more than three clock cycles. In order to avoid this condition, the "00" state transitions to "11", and the "11" state transitions to "00" in such a way as to avoid this condition. Be sure to state any assumptions you make for this problem. *Minimize the number of states in your design. Don't use a counter in this design.*

- The D input represents the value that is shifted into the shift register
- The Reset signal is an asynchronous input that places the input into the "11" state
- The Q output is the value of the shift register



**19)** Design a circuit that inputs a 10-bit unsigned value (A) and five times that value (5A) after a "GO" signal is asserted.

# 28 Structured Memory: RAM and ROM

## 28.1 Introduction

The previous chapters dealt with basic memory elements in digital design, but on a relatively small scale (flip-flops and registers). While those types of memory are important, you typically find other types of memory in digital systems. We classify flip-flops and registers as "incidental" memory; this chapter introduces the notion of "structured[1]" memory, which has significantly more storage capacity than incidental memory. You must learn a new set of skills and vernacular when you deal with structured memory; this chapter discusses some of the more basic aspects of memory.

**Main Chapter Topics**

**OPERATIONAL OVERVIEW OF MEMORY**: This chapter provides an overview of the basic operational and performance characteristics of memory as well as common terminology associated with memory.

**MEMORY TYPES:** This chapter introduces the two accepted main types of memory, RAM and ROM, by describing their differences and similarities.

**MEMORY INTERFACE METRICS**: This chapter describes the basic interface issues involved in structured memory device.

**Chapter Acquired Skills**

- Be able to describe the difference between incidental and structured memory
- Be able to name and describe the basic memory types
- Be able to describe basic performance parameters of structured memory
- Be able to use basic structured memory devices to solve digital design problems

## 28.2 Memory Introduction and Overview

There are many different types of memory out there; most of them are beyond the scope of a basic digital design course. If you ever need to work with a new memory device, you'll be ready because you're familiar with the basic operation of structured memory.

Before we start, we need to make one clarification. Often time when we discuss the notion of memory, we sometime use the terms "data" and "information" interchangeably. In most cases, this is no big deal, but you need to understand there is a distinct difference. In the context of digital design, data is nothing more than a bunch of 1's and 0's, while information relates to the interpretation of the 1's & 0's. We often refer to data as having information content; there is actually a unit used to measure the information content of data[2]. It is up to the user to interpret data as having certain information content or not. For example, consider a memory unit; if

[1] I've adopted this term from the notion of "regular structures", which roughly refers to larger semiconductor devices that have a large and repeated structure that is dedicated to a single purpose. In this case, the purpose is memory.

[2] Somewhat unfortunately, we use the term "bit" to measure the information content of data. This metric is a function of probability and is not related to the "binary digit" definition of bit that we use in this text.

the stored data represents instructions to a computer, then you could consider the data to be information. On the other hand, if you have a memory that you have never written to, the memory is still full of 1's and 0's, but the data has no meaning.

### 28.2.1 Basic Memory Operations: READ and WRITE

The two operations associated with memory are *reading* and *writing*. The notion of a "memory read" or "reading from a memory" refers to the action of retrieving data currently stored in memory. Retrieving data specifically means that you're copying the data from memory to another place, but not changing the data in memory. The notion of a "memory write" or "writing to a memory" refers to the action of placing new data in memory, which means you are changing the data stored in memory. Reading and writing memory are the copying of data from memory (reading) and the transfer of data into memory (writing), respectively.

### 28.2.2 Basic Memory Types: ROM and RAM

There are many different flavors of memory in digital-land; each of these memory types has their own acronym describing them. Despite this relatively high number of memory types, we classify all of them as either RAM or ROM, , which are acronyms for *random access memory* and *read only memory*, respectively. These terms are rather misleading, particularly in regards to the attributes of modern memory. In an effort to classify memories as either RAM or ROM, these two acronyms have rather loose definitions. Here is the information embedded in those acronyms.

- The notion of a "read only memory", or ROM, implies that you'll only be reading from a memory, and never writing to it. Because the memory is a "read only" memory, you can only retrieve data from that memory; you cannot "easily"[3] alter the data in that memory.
- The notion of a ROM brings up the issue of whom or what put the data into the ROM. This starts delving down into the various sub-types of ROM; we don't want to go there because we want to keep this discussion general. Writing to a ROM is a "special" operation performed by "something". All we're interested in is that there is data in the ROM.
- The term *random access* refers to the fact that it requires the same amount of time to access (either reading or writing) each "chunk" of memory stored in the device. While this notion seems rather simple, not all memory devices fall into the category of "random access". The two most obvious notions of non-random access memories are "hard drives" and "tape drives". The time required to access data in your hard drive is different depending on the physical location of the data on the disk and the current location of the read/write heads. Recall that the hard drive is a mechanical storage device that requires motors to move a physical device (the read/write head) radially across the spinning media to access the data. If the heads are close to the data, it require less time to access the data than if the heads must move a long way to access the data.
- Although the term ROM refers to read only memory, ROMs are also random access devices. Thus, you can access any of the chunks of data stored on a ROM in an equal amount of time.
- All memories have the notion of being either *volatile* or *non-volatile*. If a particular memory is volatile, the data stored in that memory is lost when you remove power from that circuit. Conversely, the data in non-volatile memory is not lost when you remove power. It is generally accepted that RAMs are volatile and ROMs are non-volatile.

Despite all these misleading terms and acronyms associated with structured memory, RAM and ROM do have accepted definitions. Table 28.1 lists these accepted differences and similarities.

[3] Meaning that many types of ROM can be written to; we'll not discuss those cases.

| Memory Type | Random Access | Operations | Volatility |
|---|---|---|---|
| RAM | yes | read & write | volatile |
| ROM | yes | read | non-volatile |

**Table 28.1: Accepted attributes of RAM and ROM.**

## 28.3 Software Arrays vs. Hardware Structured Memories

The notion of structured memory is not as new as it seems, as there is a direct analogy to the use of arrays in programming languages. Recall that an array in computer programming is a data structure that allows you to store values and later access those store values.

**Accessing values in an array:** This operation is analogous to a read of a memory. In computer programming, when you access a value in an array, your program must provide an index that indicates which value in the array you want to access. The array "returns" the requested value without changing that value in the array. In hardware, the circuit must provide value (the address) that indicates which address in the memory you want to read from. The memory then outputs that value; the read operation does not change the value.

**Changing values in an array:** This operation is analogous to a write of memory. In computer programming, when you place a new value into an array, your program must provide an index that indicates which value in the array you want to change. The array then replaces that value with the new value. In hardware, they circuit must provide a value (the address) that indicates which value in the memory you want to write to and the new data. The memory then changes the value at that address to the new value.

## 28.4 Memory Operation Details: Reading and Writing

Figure 28.1 shows a high-level diagram of a generic memory device. We can classify the various signals associated with interfacing with a memory device into three categories: address lines, data lines, and control lines[4]. The following is a general overview of these lines. In general, the widths of these bundles are associated with the specific capacity attributes of the memory; we deal with those issues soon.

**Data Lines:** The data lines are a set of signals that route the bits you're writing or reading into or out of the memory device. The arrow associated with the data lines has an arrowhead on each end, which signifies that data on those particular lines can travel either into the memory (for read operations) or out of the memory (for write operations)[5]. The data lines can be either serial or parallel; the bundle notation in Figure 28.1 means the data lines are parallel. Figure 28.1 happens to show only one set of data lines; memories often separate input and output data lines.

**Address Lines:** The address lines are a set of signals that provide the memory with a "location" within the memory to write to or read from. The address lines are the method that the memory uses to differentiate between chunks of memory on the interior of the device.

**Control Lines:** The control lines are a set of signals that determine and direct the various operations associated with the memory. The best example of the responsibility of the control lines are with RAM devices that are both readable and writeable; the control lines allow the user

[4] In this context, the notion of "lines" refers to a bundle of wires or signals. You often hear the term "lines" associated with standard bundles such as "data", "address", and "control" lines.

[5] But not both directions at the same time.

to control which operation occurs. The underlying notion of control lines is that simple memories have few control lines; more complex memories have more control lines[6].

We soon delve further into the details of memory interfacing; for now, you can consider the general interfacing operation of a memory read as: 1) give the memory an address, 2) tweak the control lines, and 3) wait for the data. For memory writes, you generally 1) give the memory an address, 2) give the memory the data, 3) tweak the control lines, and 4) wait for the data to write to memory.

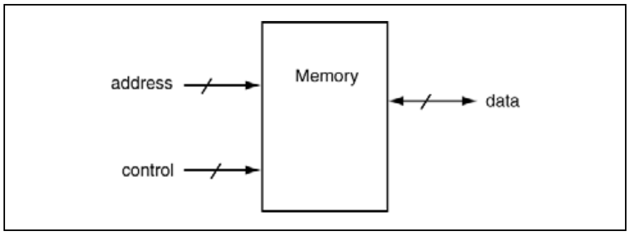


**Figure 28.1: A general diagram of a memory integrated circuit.**

## 28.5 Memory Specification and Capacity

When working with memory and memory systems, the two most important pieces of information are the capacity and the speed of the memory. The memory capacity refers to how much data the memory can store while the memory speed refers to how fast you can access (read or write) that data.

People in digital-land describe memory capacity in many different ways. As is typical in any human oriented pursuit, people attempt to make their "thing" sound better than it really is; the same idea applies to memory capacity specifications. While these statements are not lies, they are misleading. You, the digital designer must see through the smoke and hand waving and understand the characteristics of the memory you're working with.

We know that memory stores bits, and these bits are stored at certain addresses within the memory, but memories are rarely bit-addressable. In other words, specific memory devices only allow you to access larger chunks of data. If you need to read or write a single bit, you must start with the minimum chunk of addressable data specified by the device. Making memory bit-addressable would create an inefficient device, so memories generally compromise by providing data only in chunks.

Memories usually store data in groups of bits, which we refer to as a *word*. The official definition of a word is the smallest addressable unit (or chunk of bits) in a memory. This term is important because we typically described memories and memory systems in terms of words rather than bits. Referring to memory in terms of words is the honest approach.

Figure 28.2 shows a diagram of a generic memory including some typical memory characteristics. The metrics in the diagram are typical of most memory devices. Here is an overview of the most important aspects of Figure 28.2 while Table 28.2 summarizes all the gory details.

- The by "$2^m$ x S" notation is how we state the capacity of a memory. The underlying notion is that we are modeling the memory as a two-dimensional grid, as the "x" in "$2^m$ x S" indicates.
- Everything having to do with memories relates to binary. The term "m" refers to the width of the address bus or number of address lines, which is the number of memory chunks that a memory can access is two raised to the number of address lines. The true capacity of a memory (the amount of data it can store) relates to the number of address lines.
- The term "S" is the width of the data bus or data lines, or the word width for the memory. Datasheets often state this metric in bits, but should state it in word capacity.

[6] In an effort to increase memory capacity while keeping physical size small, interfacing some modern memories have become rather complicated and thus have a relatively large number of control signals.

- The total word storage capacity for the memory is how many words the memory can store. For this particular memory, the word storage capacity is thus $2^m$.
- The total bit storage capacity for the memory is a product of the number of words and the number of storage locations in the memory. Thus the bit storage capacity is given by $2^m$ x S.
- We don't include a bundle width indication on the control lines in order to keep the discussion general. The notion of $2^m$ x S is common; the control lines for memory modules tend to vary greatly across different devices.

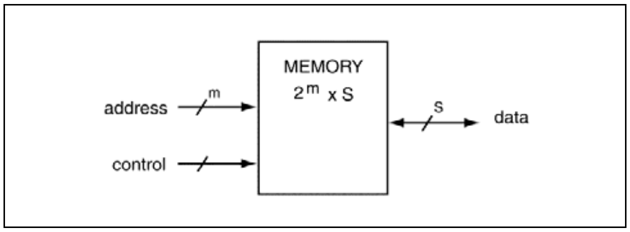


**Figure 28.2: A diagram of memory indicating notions of storage capacity.**

$$capacity\ (in\ bits) = 2^m \cdot S$$

**Equation 28.1: Closed form formula for memory storage capacity in bits.**

| Symbol | Definition |
|---|---|
| m | Bit-width of address bus |
| S | Bit-width of data bus (word size) |
| $2^m$ | Memory capacity in words |
| $2^m$ x S | Memory capacity in bits |

**Table 28.2: Summary of memory definitions and properties.**

## 28.6 Memory Interface Details

This section examines the control lines and their relation to the data and address lines for basic read and write operations on a generic memory. Recall that a memory write transfers a word to be stored in memory while a memory read prompts a memory to output the contents of memory. The reading and writing of memory is controlled by the "control lines" of the memory device. Every memory has its own method of reading and writing; specifically, each memory has its own protocol for tweaking the control lines in such a way as to obtain the desired function from the memory device.

**Memory Writes:** For a memory write operation, you provide the memory with data that overwrites data currently stored in the memory. The information on the address lines provides the location of where the word is stored. The bits on the data lines provide the data that we transfer and store on the memory device. The write operation overwrites the data currently stored at the address indicated by the address lines.

**Memory Reads:** For a memory read operation, you prompt the memory device to output the data currently stored at a specific location in memory. The information on the address lines provides the location in memory of where you want to read from. Thus, the address lines provide the memory location of the word that transfers out of the memory; the transfer occurs by placing the data at the specified address onto the data lines. Read operations don't alter values stored in the memory device.

| Steps for Memory Writes | Steps for Memory Reads |
|---|---|
| Apply the information representing the memory location of where you desire to store the given word to the address lines.<br>Apply the information representing the actual data bits to be written to the data lines.<br>Tweak the control lines to make the write operation occur.<br>Wait for data to write | Apply the information representing the memory location of where you desire to retrieve the given word to the address lines.<br>Tweak the control lines to make the read operation occur.<br>Wait for valid data to be output |

**Table 28.3: Summary of generic steps required for memory reads and writes.**

## 28.7 Memory Performance Parameters

When we speak about memory devices, we're talking about actual physical electronic devices. This means that read and write operations require finite amounts of time to happen. Most of the associated performance parameters are outside the scope of this discussion, but some are basic enough for an overview here.

Figure 28.3 shows a BBD for a simple RAM. This RAM has two control inputs: **CLK** and **WE**, where **WE** is a common acronym for *write enable*. The BBD for this RAM does not completely describe how the device operates; you need more information, as we use this device in several examples. Here is what we need to know about the device in Figure 28.3:

- The RAM in has an asychrounous read. This means that the RAM outputs the requested data as soon as it is physically capable after it receives a new address value; the read operation is not dependent upon the clock signal. The **WE** enable remains unasserted for read operations.
- The RAM in has a synchronous write. This means that write operations are synchronized with the active edge of the clock, which we assume is the rising edge in this example. The device initiates the write operation when it detects an asserted WE signal at the same time as a rising clock edge. The write operation requires a finite amount of time to complete.

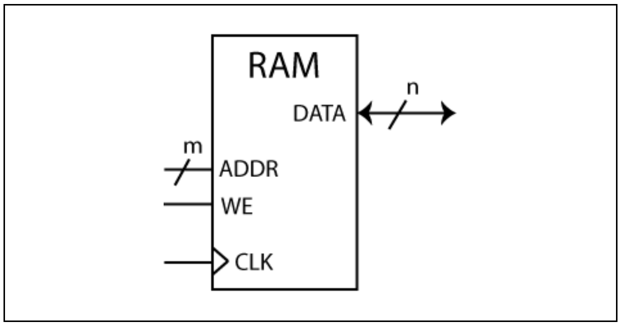


**Figure 28.3: A typical control sequence for a memory read operation.**

Figure 28.4 and Figure 28.5 show generic timing diagrams associated with typical read and write operations, respectively. For this device, the number of address and data lines does not matter for this discussion

Figure 28.4 shows a timing sequence for a memory read operation. Because the reads are synchronous, we don't need to show the **CLK** input. The one control input of interest is the **WE**, which remains unasserted for the read operation. Once a valid address appears on the **ADDR** input, the RAM outputs the data at that storage address after a finite amount of time, which we refer to as the *read access time*.

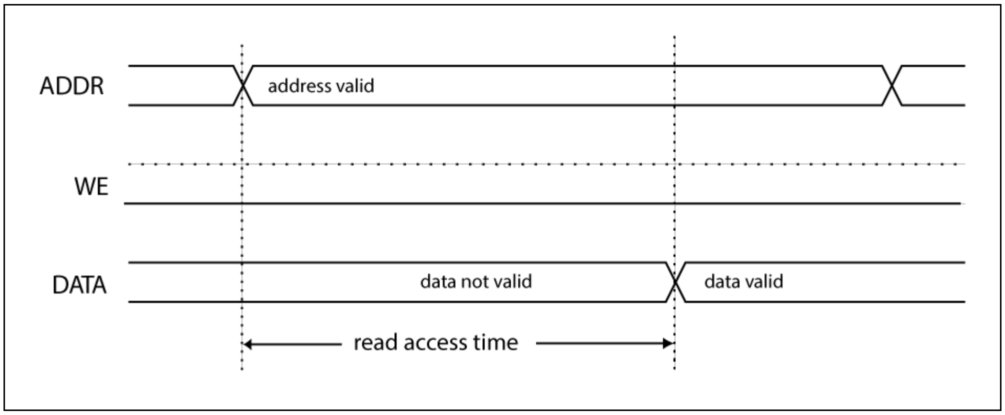


**Figure 28.4: A typical control sequence for a memory read operation.**

Figure 28.5 shows a timing sequence for a memory write operation. Because this RAM has synchronous writes, we include a **CLK** signal in the timing diagram. The writing of new data to the RAM is initiated by two control signals: **CLK** and **WE**. For a write to initiate, the **WE** control input must be asserted when a rising edge appears on the **CLK** input. The physical writing of data to the RAM occurs a finite amount of time later, which we refer to as the *write cycle time*.

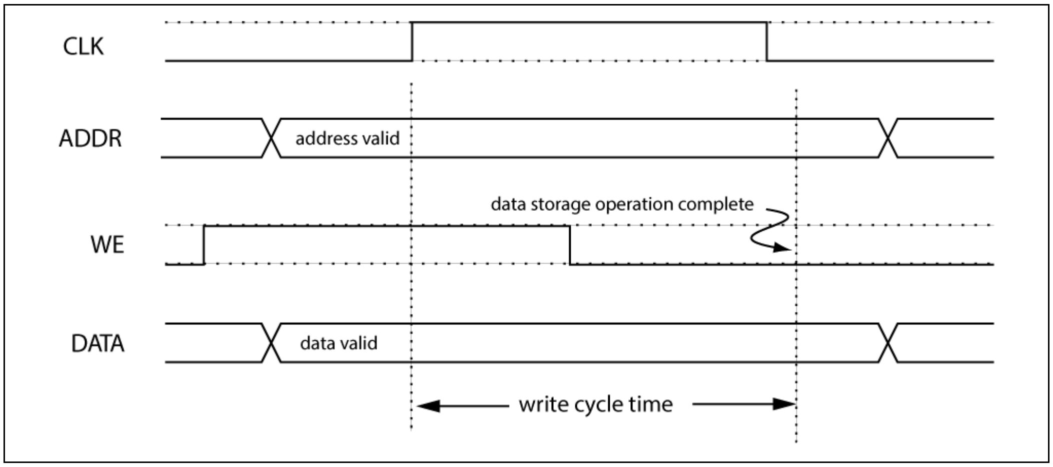


**Figure 28.5: A typical control sequence for a memory write operation.**

We use three main parameters to describe memory performance, which states how fast you can read from memory (read access time), how fast you can write to memory (write cycle time), and roughly how much data you can pass back and forth to and from the memory (bandwidth). Figure 28.4 and Figure 28.5 show graphic examples of the read access and write cycle times, respectively. The list below provides a more detailed description of these three performance parameters.

**Memory Read Access Time:** The minimum time required to access a word from memory. This is the amount of time measured from the application of a valid address to the address lines to the appearance of the valid data on the data lines.

**Memory Write Cycle Time:** The minimum time required to write a word to memory. This is the time measured from the application of a valid address lines to the completion of the internal operations required to successfully store the data in memory.

**Memory Bandwidth:** The maximum data transfer ***rate*** for a memory device. Since both read and write operations require finite amounts of time, it's worthwhile knowing the amount of data that we can physically transfer to and from memory in a given amount of time.

As with just about everything in digital-land, the faster something can operate, the more highly regarded that devices. This is maybe even more so true with structured memory devices as they are typically a major component in many digital systems, particularly computer systems. Moreover, in many digital systems, more than one device in the system must access memory. Often times more than one device must simultaneously access memory; this situation creates what we refer to as a *bottleneck*. This condition is undesirable in the one or more devices must wait to access memory[7]. The notion of "waiting" in digital-land means your device is probably doing nothing, thus probably lowering the overall throughput of your system. Roughly speaking, the faster your memory operates, the less chance of a bottleneck; or the less problematic that bottleneck is if you had a slower memory.

Any time you work with a new memory device, you'll find yourself concerned with the above parameters. Probably one of the most informative items regarding working with memory devices is the associated timing diagram, which you can find in the associated datasheet. There is almost a special language used to specify all the timing parameter associated with memory devices, once you start working with memories, you'll quickly get the hang of things.

## 28.8 Memory Address Ranges

Anytime you work with memories, you run into similar sets of numbers having to do with ranges and maximum values. Table 28.4 shows a set of values that you inherently become intimately familiar with once you spend some time working with memory. The values in Table 28.4 are systematic; familiarizing yourself with these values is not a big deal.

Table 28.4 shows the relation between the number of address bits of a given memory and the associated address range. The first column in Table 28.4 shows the number of address bits associated with a given memory while the other three columns show the zero-based address ranges possible from those given address bits. The decimal representations quickly become barely perceptible. We don't even bother writing out the binary equivalents, as we would quickly inundate your brain with 1's and 0's.

There are a few other important things to realize about Table 28.4. The "Address Range" column provides the associated address range in an 8-digit hexadecimal format. Note the maximum address in any range is associated with all the address bits being at a '1' value. This subsequently provides the "1→3→7→F" format associated with the first non-zero digit reading from left to right. Also note for both the third and fourth columns of Table 28.4 that the number ranges double as you proceed downwards in the table. This is a by-product of the underlying binary nature of memories.

[7] There is a notion of "multi-port" memories. These memories typically allow some type of parallel operation such that two devices can simultaneously read from two different memory locations. These types of memories become expensive and certainly exercise the inherent trade-offs in digital systems designs.

| # of Address Bits | Decimal Range | Address Range (hexadecimal) | Abbreviated Range |
|---|---|---|---|
| 1 | 0-1 | 0-00000001 | - |
| 2 | 0-3 | 0-00000003 | - |
| 3 | 0-7 | 0-00000007 | - |
| 4 | 0-15 | 0-0000000F | - |
| 5 | 0-31 | 0-0000001F | - |
| 6 | 0-63 | 0-0000003F | - |
| 7 | 0-127 | 0-0000007F | - |
| 8 | 0-255 | 0-000000FF | - |
| 9 | 0-511 | 0-000001FF | - |
| 10 | 0-1023 | 0-000003FF | 0-1k |
| 11 | 0-2047 | 0-000007FF | 0-2k |
| 12 | 0-4095 | 0-00000FFF | 0-4k |
| 13 | 0-8191 | 0-00001FFF | 0-8k |
| 14 | 0-16383 | 0-00003FFF | 0-16k |
| 15 | 0-32767 | 0-00007FFF | 0-32k |
| 16 | 0-65535 | 0-0000FFFF | 0-64k |
| 17 | 0-131071 | 0-0001FFFF | 0-128k |
| 18 | 0-262143 | 0-0003FFFF | 0-256k |
| 19 | 0-524287 | 0-0007FFFF | 0-512k |
| 20 | 0-1048575 | 0-000FFFFF | 0-1M |
| 24 | 0-16777215 | 0-00FFFFFF | 0-16M |
| 32 | 0-4294967295 | 0-FFFFFFFF | 0-4G |

**Table 28.4: Number of bits and associated number ranges.**

**Example 28.1: Design #1: RAM Summation**

Design a circuit that sums the values in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The summation begins when a **GO** signal asserts. The final sum remains on the circuit's output until another assertion of the **GO** signal. Assume the circuit contains numbers in unsigned binary format. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit's FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

**Solution**: The first step in your solution is drawing the top-level BBD. The problem statement generally states the exact characteristics of outputs in problems such as these (though sometime not overly explicit), but this problem requires some extra thought and calculation. We need to show the width of the output, which represents a summation of the 16 values in the RAM. The width of the data in the RAM is 8-bits, and we know they are unsigned values. This means the largest value of the sum is 16 x ($2^8$-1). We could break out the calculator, but it's better to note that we're working with powers of two, so the maximum summation is $2^4$ x $2^8$, or $2^{12}$. Therefore, the width of the summation is 12 bits. Figure 28.7 shows the top-level BBD for this problem.

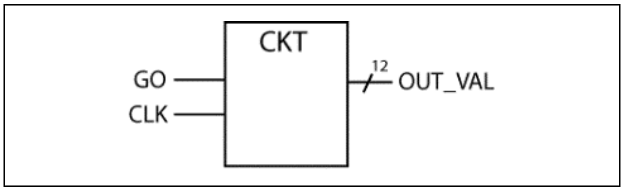


**Figure 28.6: The top-level BBD for this example.**

The next step in the solution is to create an inventory of the modules our solution requires. The following is an outline of our thought process.

- We know this problem has a RAM because the problem description says so.
- Any RAM we work with in this text uses the output of a counter to provide an address input to the RAM. Many different circuits or modules can provide the address inputs, but the simplest approach for this text is to use a counter output provide the address.
- The circuit also is summing all the values in the RAM. Because the RAM can only output one value at a time, we need a circuit that keeps a running total of the RAM's stored values. This calls out for an accumulator, which is a combination of an RCA and a register. The accumulator's register provides a persistent output.
- Something must control this circuit, and this control is non-trivial, which calls out for a FSM.

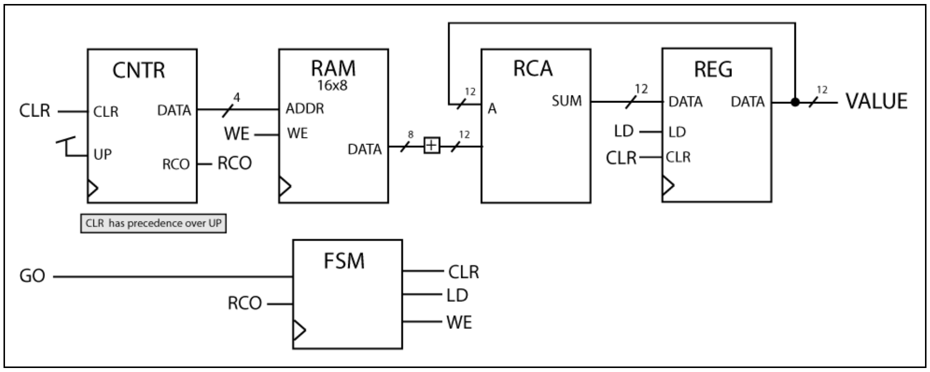


**Figure 28.7: The lower-level BBD for this example.**

Figure 28.7 shows the final circuit for this problem; meaningful commentary follows the diagram.

- The counter always counts up when it's not loading.
- We need to zero-extend the RAM data to make it 12 bits, which makes the RAM output compatible with the output of the accumulator's register, and the other input to the RCA. We use the square symbol with a "+" in the center to do this (which is arbitrary).
- We had to include an annotation stating that the counter's **CLR** input has precedence over the **UP** control input.

Figure 28.8 shows the state diagram for this example; here are a few items of interest to note about the state diagram.

- We drew the state diagram using two states, which requires treating **CLR** as a Mealy-type output. This approach was arbitrary, but it saved drawing an extra state.
- In the "wait" state, the register's **LD** input is disabled; we enable it while the circuit is summing.

- We always disable the RAM's WE input as this problem requires no writing to the RAM.
- The FSM remains in the "sum" state until the counter asserts **RCO**.

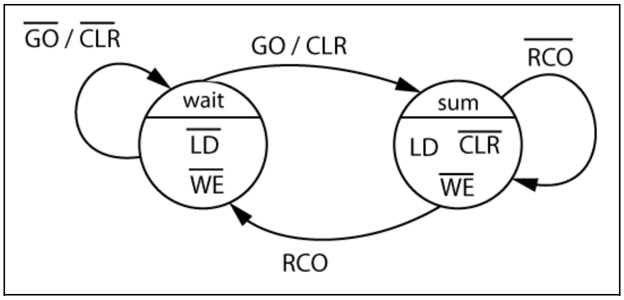


**Figure 28.8: The state diagram associated with this example.**

The FSM controls both the **LD** and **CLR** inputs, while the **UP** input of the counter is hardwired to always count up. The **GO** signal is a form of external control. This circuit thus has external, circuit, and internal controls.

The counter has 16 unique count values that it steps through after receiving a **GO** signal. The first clock cycle causes the FSM to transition from the "wait" state to the "sum" state. The summing operation for this circuit thus requires 17 clock cycles.

**Example 28.2: Design #2: Minimum Value & Address Displayer**

Design a circuit that finds the smallest value in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The summation begins when a **GO** signal asserts. The circuit's output shows the minimum value as well as the address where that value resides in RAM. Both the value and the address remain on the circuit's output until another assertion of the **GO** signal. Assume the circuit contains numbers in unsigned binary format and that every value in the RAM is unique. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit's FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

**Solution**: This is another problem that requires iterating through all the values in a RAM. In this case, the circuit outputs the minimum value in RAM as well as the address of that minimum value. Figure 28.9 shows the top-level BBD for this solution.

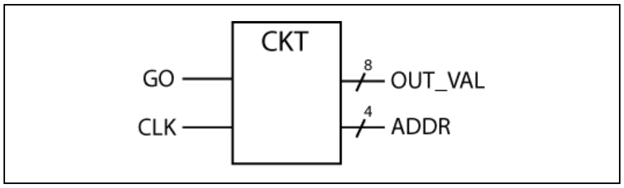


**Figure 28.9: The top-level BBD for this example.**

The next step in the solution is to create an inventory of the modules our solution requires. Here is the general thought process.

- The problem description states that the circuit contains a RAM; we then know that the circuit then uses a counter to generate an address for the RAM. There are 16 values in the RAM, so the width of the counter's output is 4-bits.
- The circuit needs to store two values: the smallest value in the circuit and the location in RAM of the smallest value. These values both need to be persistent after the algorithm completes, so we know that the circuit requires two register. The register storing the smallest value is eight bits while the register storing the address of that value is four bits.
- This circuit needs to do continual comparisons to find the smallest value, so we also require an 8-bit comparator.
- In an effort to make this circuit generic, we first pre-load the 8-bit register with the minimum possible unsigned 8-bit value. The first step in the algorithm is then to load "all 1's" into the register that holds the minimum value, which we do in order to reduce the complexity of the overall circuit. This is somewhat of a trick, but it is something you see often.
- We use a MUX to select what value appears on the minimum value register's **DATA** input. We first need to load the register with the maximum 8-bit value; after that, we need to be able to load the register with the current RAM value when the comparison result dictates.
- We need to state that **CLR** has precedence over the **UP** input for the counter, and that the **CLR** input has precedence over the **LD** inputs for the two registers.

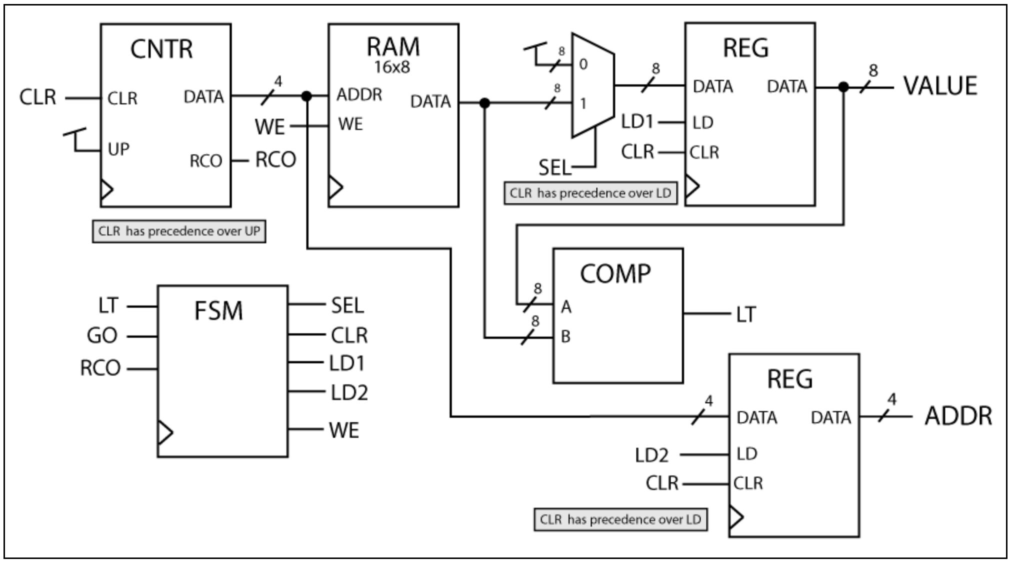


**Figure 28.10: The lower-level BBD for this example.**

Figure 28.11 shows the state diagram for this example. Although it looks quite busy, it's actually very structured, as the following items indicate.

- We model the **LD1** and **CLR** as Mealy-type outputs in the "wait" state, which is arbitrary. We did this in order to save a state in the state diagram.
- When the **GO** signal asserts, the FSM clears the address register and counter, and loads the minimum value register with the largest possible 8-bit unsigned binary value.
- This circuit does not write to RAM, so we always disable the **WE** signal.
- The "search" state appears busy, but it's actually structured. Two things are happening. First, when the **LT** signal is not asserted, we don't load any new values to either register (**LD1** & **LD2** are not asserted). When the **LT** signal is asserted, we load the current address (the output of the

counter) to the address register, and load the current RAM data output to the minimum value register. One of these two operations always happens no matter whether the **RCO** signal is asserted or not. When the **RCO** signal is asserted, that means the counter's output is at its maximum value and we must terminate the algorithm by transitioning back to the "wait" state.

- There are four arrows leaving the "search" state; each of these arrows has the four different possible combinations of the **RCO** & **LT** inputs.

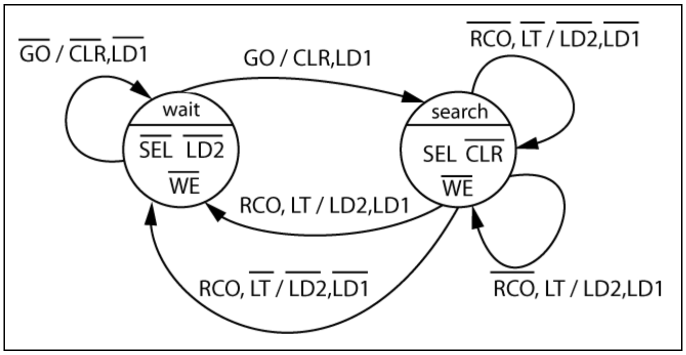

$\overline{GO}$ / $\overline{CLR}$,$\overline{LD1}$
GO / CLR,LD1
$\overline{RCO}$, $\overline{LT}$ / $\overline{LD2}$,$\overline{LD1}$
wait
$\overline{SEL}$ $\overline{LD2}$
$\overline{WE}$
search
SEL $\overline{CLR}$
$\overline{WE}$
RCO, LT / LD2,LD1
RCO, $\overline{LT}$ / $\overline{LD2}$,$\overline{LD1}$
$\overline{RCO}$, LT / LD2,LD1

**Figure 28.11: The state diagram associated with this example.**

The FSM controls both the **LD** and **CLR** inputs for both registers, while we hardware the **UP** input of the counter to always count up. The **GO** signal is a form of external control. This circuit thus has external, circuit, and internal controls.

The counter has 16 unique count values that it steps through after receiving a **GO** signal. The first clock cycle causes the FSM to transition from the "wait" state to the "search" state. This circuit thus requires 17 clock cycles to locate the minimum value for the circuit.

**Example 28.3: Design #3: Value Event Counter**

Design a circuit that finds the number of times the value 0x47 appears in a 16x8 RAM. Assume some external device previously placed the data into the RAM. The search for the given value begins when a **GO** signal asserts. The circuit's output persistently shows the resultant count value until another assertion of the **GO** signal. Assume the circuit contains numbers in unsigned binary format. Provide two levels of BBDs for your solution as well as a state diagram modeling the circuit's FSM. State the forms of control the circuit uses. Also, state how many clock cycles your circuit requires to complete the operation. Minimize the amount of hardware you use in your design.

**Solution**: This is another problem where we need to carefully choose the width of the output value. This problem asks that we count the number of value in the RAM that are equivalent to 0x47. The greatest count is when all the values in the RAM are 0x47, which is a count of 16. We thus require an output data width of five bits. Figure 28.12 shows the top-level BBD for this problem.

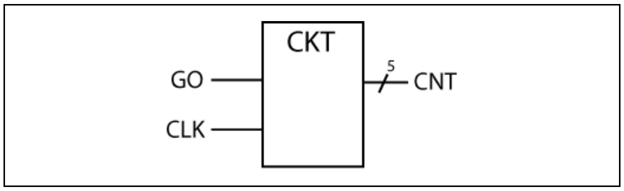


**Figure 28.12: The top-level BBD for this example.**

The next step in the solution is to create an inventory of the modules our solution requires; here is our module inventory thought process.

- We know the circuit requires a RAM, so we know the circuit then uses a counter to generate an address for the RAM. There are 16 values in the RAM, so the counter's output is 4-bits wide.
- We are looking for the value of 0x47, which means we need to compare the data at each RAM location with that value. Our circuit thus requires a comparator.
- We must determine the number of times the 0x47 appears in the RAM, so the first thought may be that our circuit requires an accumulator. We could use an accumulator, but we can satisfy our circuit's needs with an event counter, which is a counter that increments when it detects a certain event. The event we are detecting is the presence of 0x47 in the RAM.
- We need a FSM to control our circuit.

Figure 28.13 shows the lower-level BBD for our solution. Here are a few interesting items in that BBD:

- The comparator hardwires one the "event" value to one of its inputs.
- We don't need to provide a note for the event counter regarding the precedence of the **LD** and **CLR** inputs; the FSM handles that aspect of the circuit.
- The **CLR** signal on the two counters are physically the same signal.
- The **DATA** input to the RAM is hardwired to zero; when we find the value of 0x47 at a particular address, the circuit writes 0x00 to that address location.

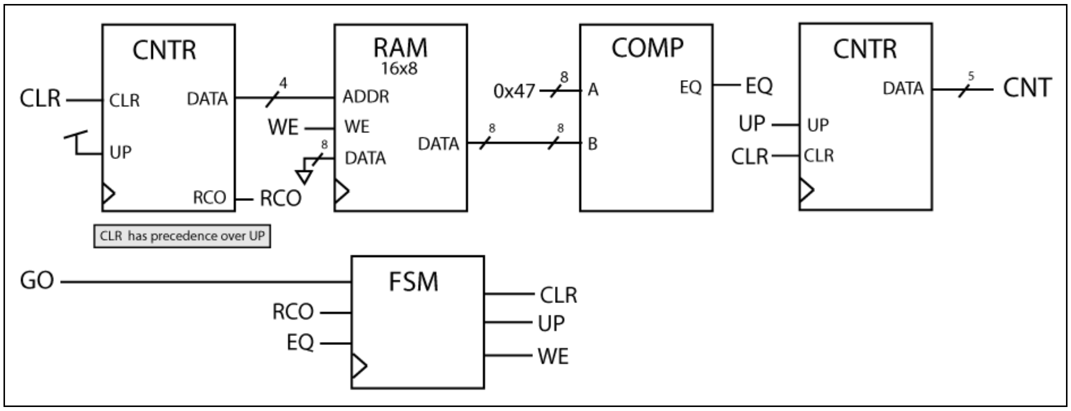


**Figure 28.13: The lower-level BBD for this example.**

Figure 28.14 shows state diagram for our solution. The state diagram looks rather busy, but once again, it is nicely structured. If you see and understand that structure, the state diagram seems relatively simple. Here the full story:

- We model the **LD** and **CLR** as Mealy-type outputs in the "wait" state, which is arbitrary. We did this in order to save a state in the state diagram.

- This **WE** input is always disabled in the “wait” state. The state of the **WE** signal in the “scan” state depends on the **EQ** input, where it writes a new value to RAM when the **EQ** is asserted, or does not change the RAM contents otherwise. We thus model the **WE** input as a Mealy-type output in the “scan” state and as a Moore-type output in the “wait” state.
- The “scan” state has four arrows leaving the state, where each arrow represents one combination of the two inputs (**RCO** & **EQ**).
- When the **RCO** is not asserted, the circuit either increments the count and clears that corresponding address in RAM, or it does nothing; it then transitions back to the scan state. When **RCO** is asserted, it performs the exact two actions, but the FSM then transitions to the “wait” state.

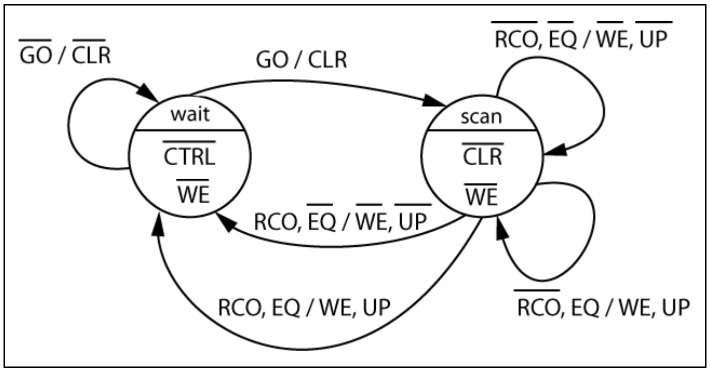

$\overline{GO}$ / $\overline{CLR}$
GO / CLR
$\overline{RCO}$, $\overline{EQ}$ / $\overline{WE}$, $\overline{UP}$
wait
$\overline{CTRL}$
$\overline{WE}$
scan
$\overline{CLR}$
$\overline{WE}$
RCO, $\overline{EQ}$ / $\overline{WE}$, $\overline{UP}$
RCO, EQ / WE, UP
$\overline{RCO}$, EQ / WE, UP

**Figure 28.14: The state diagram associated with this example.**

The FSM controls the **LD, CLR**, and **WE** inputs for the counters and RAM. The **GO** signal is a form of external control. Thus, this circuit has both circuit an external control.

The counter has 16 unique count values that it steps through after receiving a **GO** signal. The first clock cycle causes the FSM to transition from the “wait” state to the “search” state. This circuit thus requires 17 clock cycles to locate the minimum value for the circuit.

## 28.9 Digital Design Foundation Notation: RAM

We consider the RAM to be a Digital Design Foundation module. The RAM is a controlled circuit. Figure 28.15 shows the digital design foundation notation for the counter. This foundation module is both data inputs and data outputs, both of which are the same width. We use a simple device for the foundation model and consider read operations to be asynchronous and write operation to be synchronous. The **WE** signal controls whether the device is reading or writing, where **WE** is asserted for write operations and unasserted for read operations. We consider ROMs to be a subset of RAMs; ROMs are not able to write. Table 28.5 shows the foundation description for the RAM.

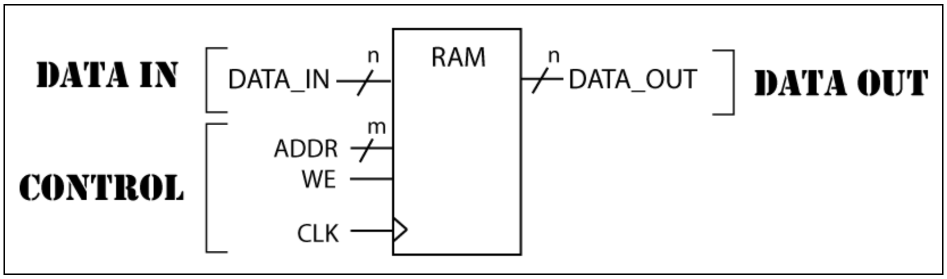


**Figure 28.15: Typical data, control and status signals for RAM. .**

| | Signal Name | Description |
|---|---|---|
| INPUT DATA | **DATA_IN** | Data to be synchronously written to RAM. |
| OUTPUT DATA | **DATA_OUT** | Data stored in the RAM at the address given by the ADDR input. |
| CONTROL | **CLK** | The **CLK** signal synchronizes the writing of data to the RAM |
| | **ADDR** | The RAM stores the value of **IN_DATA** at the address associated with the value of **ADDR** on the active clock edge (synchronously) when the WE signal is asserted. |
| | **WE** | When asserted, allows the loading of **DATA_IN** to the RAM location specified by ADDR, which is a write operation. When unasserted, the RAM outputs the data stored at the location specified by the **WE** input. |
| STATUS | **n/a** | - |

**Table 28.5: The foundation description for a RAM.**

## 28.10 Chapter Summary

- Memory is a form of a sequential circuit, but we further divide memory into two categories: “incidental memory” and “structured memory”. Incidental memory refers to items such as flip-flops and registers (relatively small) while structured memory refers to larger capacity regular structures.
- There are many type of memory in digital-land, but we can roughly classify them all as either ROM or RAM. ROM is “read only” memory while RAM is “random access” memory. Both of these memories have the random access attribute in that all of the data on the devices is accessible in the same amount of time. ROMs are considered non-volatile while RAMs are not. RAMs can be both written to and read from while ROM can only be generally read from.
- The notion of reading from a memory, or a memory READ, consists of making the data within the memory at a given address available to entities external to the memory. Memory reads generally do not alter the data stored in the memory. The notion of writing to a memory, or a memory WRITE, consists of overwriting data contained in the memory at a given address with data provided by some entity external to the memory.
- Interfacing with memory generally requires tweaking one the three types of I/O associated with memory. The three types of memory I/O are address lines, data lines, and control lines. The address lines provide an index into the memory and allow access to a particular chunk of data stored in memory. The data lines provide a path for data to flow into (write) or out of (read) memory. The control lines provide a structured approach to read from and/or writing to the memory device.
- Memories are generally rated by the capacity (how many bits they can store) and the speed (how fast you can read and/or write the memory). The term “word” is used to refer to the smallest chunk of memory available at a given address in the memory. Memory capacity can be stated in bits or words; any other approach is suspect as it can be misleading
- Memories typically store two raised to an integral power number of words. The integral power in this case is the number of address lines on the memory. The number of address lines is sometimes referred to as the width of the address bus.
- Memory speed is rated by how fast you can read from it and/or write to it. The term “read access time” refers to how fast you can read from a memory. The term “write cycle timing” refers to how fast you can write data to a memory. The term “memory bandwidth” refers to the maximum amount of data going to and coming from a particular memory in a given amount of time.

## 28.11 Chapter Exercises

**1)** In your own words, briefly describe what is meant by the term “random access” in the context of computer memories.

**2)** In your own words, briefly describe what is meant by the term “random access” in the context of computer memories.

**3)** In your own words, briefly describe what is meant by the term “random access” in the context of computer memories.

**4)** In your own words, explain how read and write access times affect the bandwidth of a given memory.

**5)** Describe a circuit situation where having a large memory bandwidth would be important.

**6)** Faster memories are typically more expensive than slower memories. Speculate on why you feel this would be the case.

## 28.12 Design Problems

For the following problems:

- Provide a top-level BBD and as many lower-level BBDs as necessary to describe your solution
- Minimize the number of states in the associated state diagrams
- Minimize the use of hardware when problem require extra hardware
- Assume all inputs and outputs are positive logic unless stated otherwise
- Explicitly state whether state diagrams have Mealy or Moore outputs where appropriate
- Disregard all setup and hold-time issues
- For sequence detector problems assume the X input is stable when each clock edge arrives and that X can change no more than once per clock period.
- State all forms of control for your solution.

**1)** Design a circuit that upon the pressing of a button, determines how many values in a 16 RAM are negative, and displays that value until another button press. The RAM contains 8-bit signed numbers in RC format.

**2)** Design a circuit that upon the pressing of a button, finds the maximum value in a 16x8 RAM, and displays that value until another maximum value is found after another button press. The RAM contains 8-bit unsigned numbers.

**3)** Design a circuit that upon the pressing of a button, finds the minimum value in a 16x8 RAM, and displays that value until another minimum value is found after another button press. The RAM contains 8-bit unsigned numbers.

**4)** Design a circuit that upon the pressing of a button, determines how many values in a 16x8 RAM are evenly divisible by eight, and displays that value a button press restarts the process. The RAM contains 8-bit unsigned numbers.

**5)** Design a circuit that upon the pressing of a button, determines how many values in a 16x8 RAM have a value of 15 or less, and displays that value until a button press restarts the process. The RAM contains 8-bit unsigned numbers. Don’t use a comparator in this problem.

**6)** Design a circuit that upon the pressing of a button, sums all the values in a 16x8 RAM and displays that value until a button press restarts the process. The RAM contains 8-bit unsigned numbers.

**7)** Design a circuit that upon the pressing of a button, determines if the value in a 16x8 RAM are in ascending order. If they are in ascending order, the circuit turns on an LED; otherwise it leaves the LED unlit. The circuit does this each time a button is pressed. The RAM contains 8-bit unsigned numbers.

**8)** Design a circuit that upon the pressing of a button, determines how many bits are set in a in a 16x8 RAM and displays that number on the output. The circuit does this each time a button is pressed.

**9)** Design a circuit that reads all the values in a 16x8 RAM. If the value is less than 26, the circuit changes that value to 0x00. The circuit does this each time the button is pressed.

**10)** Design a circuit that upon the pressing of a button, determines how many values value in a 16x8 RAM are even parity and how many values are odd parity. The circuit does this each time a button is pressed.

**11)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit counts the number of values in each even address location in a 16x8 RAM that are evenly divisible by 8 and stores that count in a register.

**12)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit finds the minimum value in a 16x8 RAM. Upon completion, the circuit continually outputs both the minimum value and the RAM address of that value until another GO signal is detected. The RAM contains unsigned 8-bit values.

**13)** Provide the hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit counts how many values in each even address location in a 16x8 RAM are evenly divisible by 8. Consider address “0000” to be an even address location.

**14)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit stores the largest value in a 16x8 RAM into an 8-bit register.

**15)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit sums the values in each memory location of a 16x8 RAM if they are less than 63 and stores the result in a register. The final result should not be changed until another GO signal is detected. The RAM contains unsigned 8-bit values.

**16)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task. Upon receiving a “GO” signal, the circuit counts number of values in each memory location of a 64x8 RAM that are less than 32 and stores that count in a register. The final result should not be changed until another GO signal is detected. The RAM contains unsigned 8-bit values.

**17)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit sums the values in two 8x8 RAMs and outputs that sum until it receives another GO signal. Design your circuit for either ***minimum operating time*** or ***minimum hardware***; **state which approach you are taking**. The RAM contains unsigned 8-bit values.

**18)** Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit finds the maximum value in a 16x8 RAM, ***and then clears that value in RAM***. Upon completion of this operation, the circuit waits for another GO signal. The RAM contains unsigned 8-bit values.

# Appendix

# Mealy's Laws of Digital Design

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

> Justification: You always know enough to draw the top-level BBD interface. When you start drawing black boxes and listing what you know, you generate ideas on how to solve the problem.

**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.

> Justification: There is never one correct circuit to solve a digital design problem, which means there are many paths to take when working on a digital design problem. You're inevitably going to take the wrong path, so be ready to realize as much, and switch to a different path.

**Mealy's Third Law of Digital Design:** Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.

> Justification: Digital circuitry is inherently flexible, which allows you to solve digital design problems in different ways. The digital design must be familiar enough with digital circuitry to be able to create their designs to satisfy the design criteria and to then verify their designs work as expected.

**Mealy's Fourth Law of Digital Design**: The digital design process is circular, not linear. If you think you're going to generate the correct solution with the first pass, you're bound for disappointment. The digital design process is circular; always make going backwards a few steps to fix issues part of the design process. Don't try to make your design perfect from the get-go, make it simple to understand so that you can fix issues as they arise.

> Justification: Based on Mealy's Second and Third Laws, you always need to be willing to go temporarily backwards on your designs. Design, go back and make necessarily refinement, design some more, repeat.

**Mealy's Fifth Law of Digital Design**: Model circuits using many smaller sub-modules as opposed to fewer larger sub-modules; as this approach supports testing and increases the chances module reuse.

> Justification: Large designs are harder to understand and test, particularly if you're first passing off your models to an HDL synthesizer. Make your designs reliant upon a strong foundation of basic digital modules is always the best approach.

**Mealy's Sixth Law of Digital Design:** Don't rely on the HDL synthesizer; create your HDL models by having a remote vision of what underlying hardware should look like in terms of standard digital modules.

> Justification: Although HDLs give you the ability to model digital circuits, they are not magic. The HDL synthesizer's task is to convert pages of text into circuits; the more the options you give to the synthesizer, the less probable the synthesizer will successfully generate a circuit that works as you intended.

**Mealy's Seventh Law of Digital Design**: Always first consider modeling a digital circuit or part of a digital circuit using some type of decoder. Decoders in digital design are anything we can describe in a tabular format, so they are essentially look-up tables (LUTs).

> Justification: The basis of all digital design is defining circuits in a tabular format, whenever possible. Although this approach represents low-level design, HDL tools have strong support for table-based models.

# Requiem for the Digital Logic Designer

Digital design is the process where you create a digital circuit to solve a given problem. There are many approaches you can use to solve given problems, designing a digital logic circuit is one of them. What makes digital design so useful is that the design can generally interface with other digital circuits such as computer-type circuits. The two basic tenets of digital logic are:

Digital logic circuits are hierarchical: We can describe a digital circuit at various levels; the level at which we describe digital logic is generally the one that allows us to transfer as much useful information as possible. Abstracting digital designs to higher levels aids in understanding and designing circuits.

Digital logic circuits are decomposable into a few basic digital circuits: Although there are many ways to describe digital circuits, we strive to make the descriptions an aggregate compilation of standard digital circuits in able to help us understand the circuits.

A given digital design solves problems by having the outputs react to the inputs in a manner such that it solves the given problem. There are two basic types of digital logic circuits:

Combinatorial Circuits: circuit outputs are a function of the circuit's inputs.

Sequential Circuits: circuit outputs are a function of the sequence of the circuit's inputs.

The main ramification of sequential circuits is that they can "remember" the previous "state" of the circuit. Sequential circuits can store (remember) bits; we refer to the bits the circuit is remembering as the "state" of the circuit. Combinatorial circuits, by definition, do not have state.

Figure 28.16shows a digital logic circuit containing both sequential and combinatorial modules. We can thus model digital circuits as a controlled interaction between a set of sequential and combinatorial circuits. Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it provides a solution to the given problem.

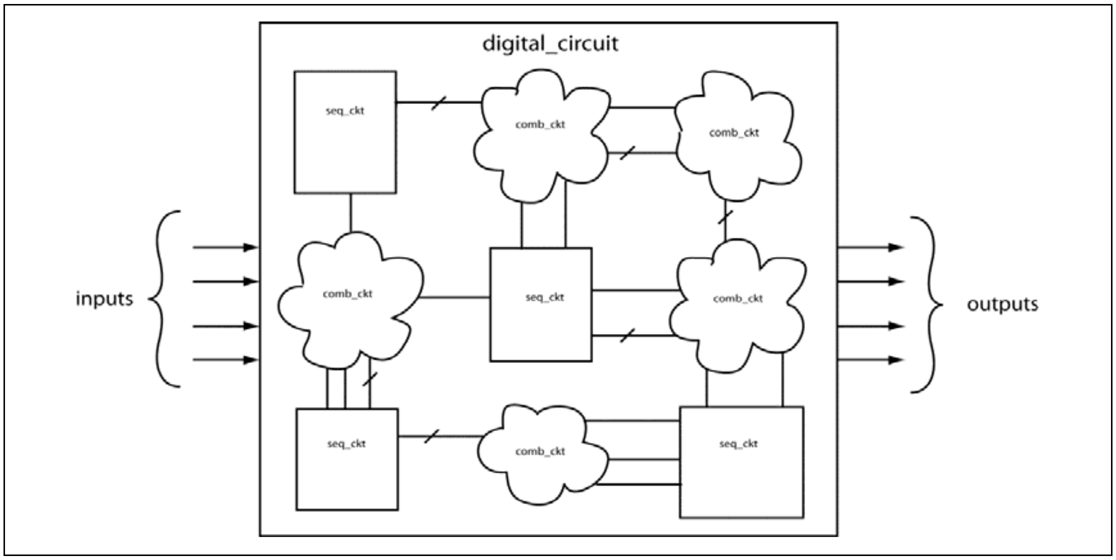
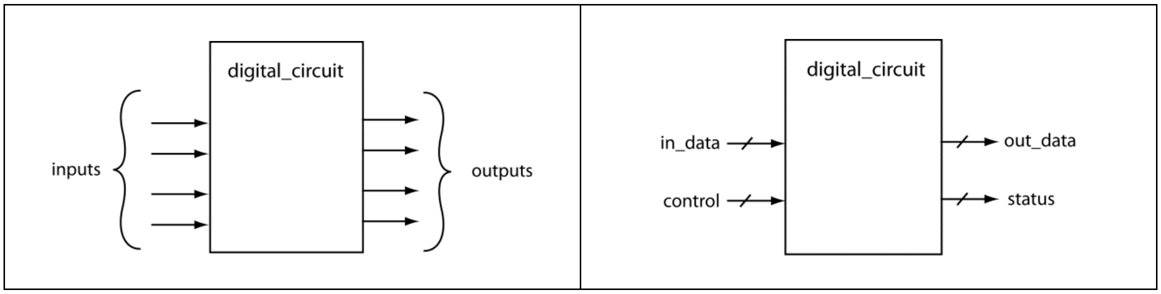


**Figure 28.16: A basic logic circuit.**

Figure 28.17(a) shows the basic model of a digital logic circuit; we characterize the signals that the outside world sees as either inputs or outputs. Because we need to control the flow of data through the digital circuit, we must more specifically define the inputs and outputs of a basic digital circuit module. Figure 28.17(b) shows that we further classify the inputs as either "data" or "control" and classify the outputs as either "data" or "status". This means the various circuit elements in Figure 28.17(b) are able to 1) pass data from their inputs to their outputs under the direction of the "control" inputs and, 2) output characteristics of the data transfers using the status outputs.



**(a)** **(b)**

**Figure 28.17: Models for a basic logic circuit (a), and a more refined basic digital logic circuit (b).**

Something must control the flow of data through the generic digital circuit. We therefore must have some other entity that interprets the status signal outputs of the circuit modules and issues control signals to those circuit modules. For the purpose of this discussion, we consider this circuit to be a finite state machine (FSM). The important thing to remember is that something controls the circuit, whether it is an FSM, a computer, or a herd of confused academic administrators.

Figure 28.18 shows a generic model of an FSM. The FSM interprets the status signal outputs from various digital modules and then outputs the appropriate control signals that are the various digital modules use as control inputs. Other interesting characteristics to note include:

FSMs generally do not have data inputs and data outputs. You can design FSMs with data inputs and outputs, but they tend to be klunky and non-generic. Non-generic FSMs require modifications if the data widths within the controlled circuit change.

The FSM is a sequential circuit because it has the ability to store bits. The FSM only stores bits to represent the "state" of the FSM, which it does in its "state variables".

The underlying model of the FSM includes three primary elements: 1) the next state decoder, 2) the output decoder, and, 3) the state variables. The next state decoder is a combinatorial circuit that decides the next state based on the given state and status inputs. The output decoder is a combinatorial circuit that generates control outputs based on either state only (Moore machine) or state and status inputs (Mealy machine). Figure 28.19 shows models for the Moore and Mealy-type FSMs.

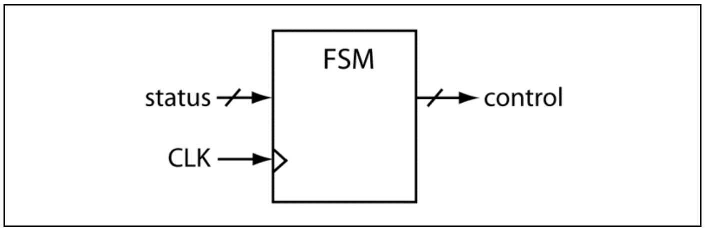


**Figure 28.18: A black box model of a FSM.**

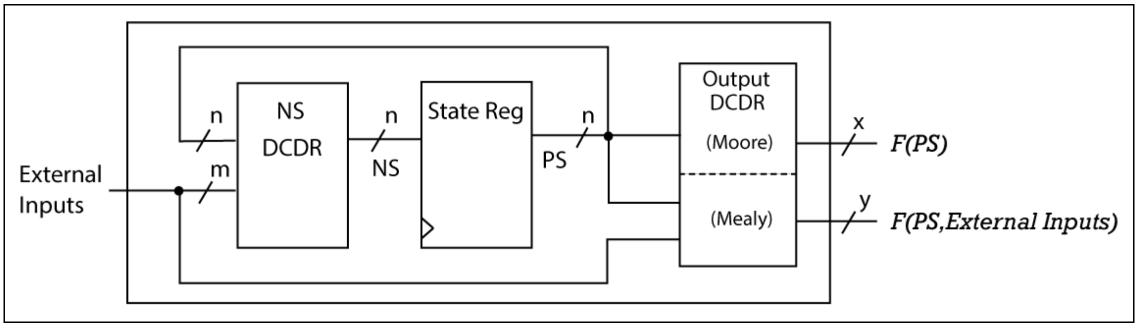


**Figure 28.19: The FSM model showing the two types of outputs (Mealy and Moore).**

Figure 28.20 shows a modified version of Figure 28.17 that includes an FSM as a control element. Figure 28.21 shows that we can further this abstraction. Figure 28.21 shows that the circuit control elements can either be hardware (FSMs) or software (microcontrollers). Additionally, Figure 28.21 shows that the modules that we can control in a digital circuit include "computer peripherals" as well as the low-level digital modules Figure 28.17. Figure 28.21 represents computer peripherals using circles.

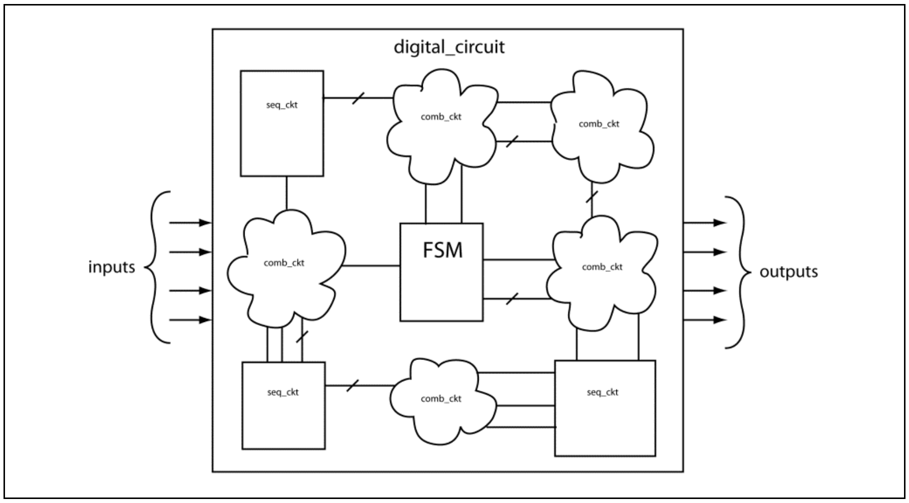


**Figure 28.20: A basic logic circuit controlled by FSM**

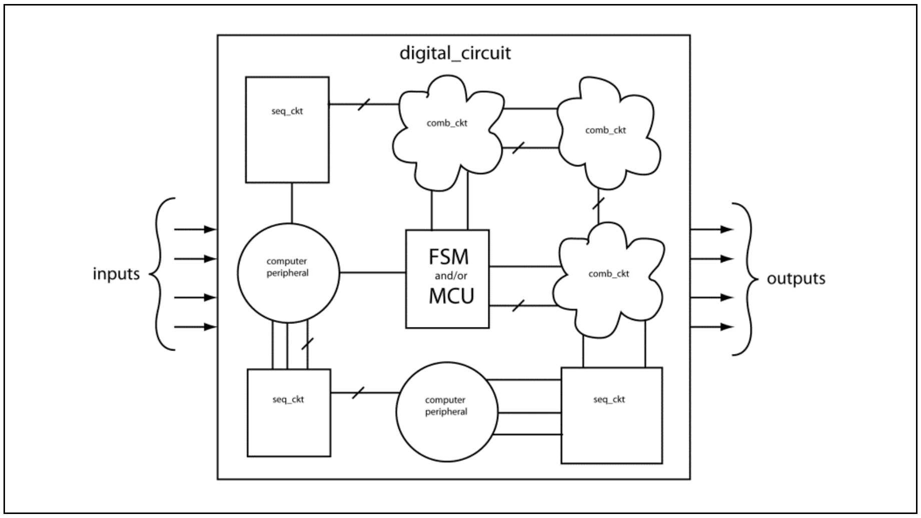


**Figure 28.21: A basic logic circuit with peripherals and various control circuits.**

## Ripple Carry Adder (RCA)

The RCA is a combinatorial module that performs addition. We often model the RCA as a series of Full Adders (FAs) connected in series such that the Co from one module connects to the Cin of the next higher bit location. The RCA can also perform subtraction by changing the sign of one addend before performing the addition.

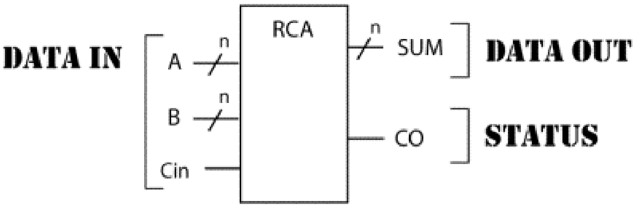
| | **RCA: Device Summary** |
|---|---|
| **Foundation Notation** |  DATA IN<br>A<br>B<br>Cin<br>n<br>n<br>RCA<br>n<br>SUM<br>CO<br>DATA OUT<br>STATUS |
| **Input/Output** | **DATA IN:** A, B, Cin. A & B are the addends; Cin is the carry in.<br>**CONTROL:** none<br>**DATA OUT:** SUM. Summation of: A+B+Cin.<br>**STATUS:** Co. The Carry out; indicates if addition operation generated a carry out |
| **Usage** | • Circuits use RCAs when they require addition or subtraction operations<br>• An RCA is a primary component of an accumulators (an register is the other component)<br>• The RCA's carry out (CO) is effectively the (n+1)th bit of a n-bit RCA<br>• The CO indicates "validity" of the SUM output when using unsigned binary numbers |

**Figure 28.22: The RCA Foundation Module overview.**

## Multiplexor (MUX)

The MUX is a combinatorial circuit that selects which of many (more than one) data inputs appear on the circuit's single data output. The SEL signal determines which signals transfers to the output, which requires that it have a width of at least: $\lceil log_2(\text{number of data inputs}) \rceil$. The width of the data inputs and outputs are equivalent. The most generic forms of MUXes include 2:1, 4:1, 8:1, 16:1, etc.

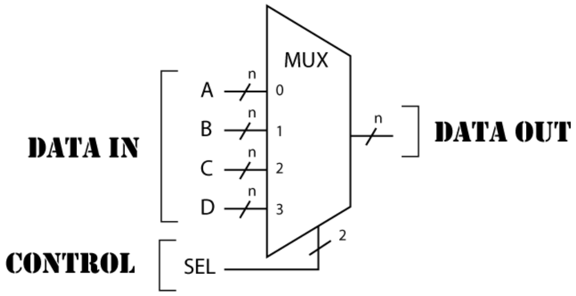
| | **MUX: Device Summary** |
|---|---|
| **Foundation Notation** |   MUX<br>DATA IN<br>A<br>B<br>C<br>D<br>n<br>0<br>1<br>2<br>3<br>DATA OUT<br>CONTROL<br>SEL<br>2 |
| **Input/Output** | **DATA IN:** A, B, C, etc; (MUXes have two or more data inputs)<br>**CONTROL:** SEL. selects which data input appears on the DATA OUT. The width of the SEL signal is such that $2^{SEL} \geq$ the number of data inputs.<br>**DATA OUT:** A single output, which is one of the inputs (selected by the SEL signal)<br>**STATUS:** none |
| **Usage** | • Circuits use MUXes when they need to make decisions. The general hardware approach to decision making is to generate valid values of all MUX inputs and then select one of the values as an output to the MUX.<br>• The width of the data inputs and data outputs generally match<br>• MUXes can have almost any number of inputs (greater than one); the constraint is:<br>**$2^{(\text{width of SEL})} \geq$ the number of data inputs** |

**Figure 28.23: The MUX Foundation Module overview.**

## Comparator

The comparator is a combinatorial circuit that generates an equality-type relationship between the two inputs. The comparator has outputs of EQ (equal), LT (less than), and GT (greater than) which are characteristics of the relationship between the two input. The comparator's two input values are typically bundled values of equal width.

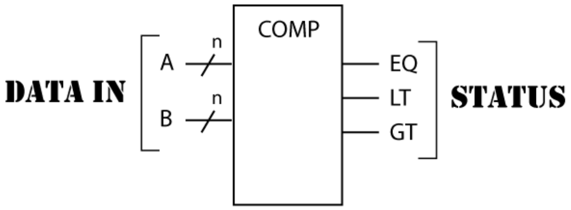
| | **Comparator: Device Summary** |
|---|---|
| **Foundation Notation** |  DATA IN<br>A<br>n<br>B<br>n<br>COMP<br>EQ<br>LT<br>GT<br>STATUS |
| **Input/Output** | **DATA IN:** A, B. the two bundled values to be compared.<br>**CONTROL:** none<br>**DATA OUT:**: none<br>**STATUS:** EQ (A=B), LT (A<B), GT (A>B) |
| **Usage** | • Circuits use comparators when they need to establish equality relationships between two number<br>• The data width of the two inputs is generally the same<br>• When appearing in circuits, comparators don't need to include every status output |

**Figure 28.24: The Comparator Foundation Module overview.**

## Generic Decoder

The generic decoder is a combinatorial circuit that establishes a functional relationship between the module's data inputs and data outputs. The generic decoder is a digital circuit implementation of a look-up-table (LUT). The generic decoder's inputs and outputs are hard to classify because inputs can include data and/or control and outputs can contain at and/or status. We thus describe this circuit using the DATA IN input for all the inputs (whether they be data or control) and the DATA OUT for all outputs (whether they be data or status). Both DATA IN & DATA OUT can be bundles or single bits, which allows us to classify basic logic gates as generic decoders.

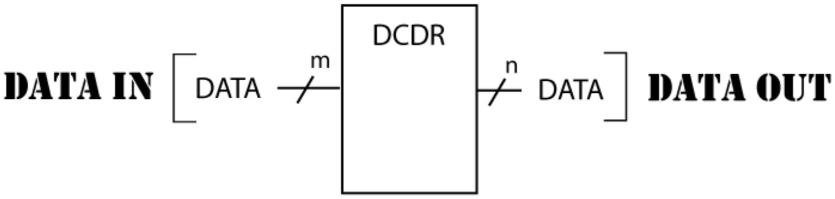
| | **Generic Decoder: Device Summary** |
|---|---|
| **Foundation Notation** |  |
| **Input/Output** | **DATA IN:** DATA; the function's independent variables<br>**CONTROL:** none<br>**DATA OUT:** DATA; the function's dependent variables<br>**STATUS:** none |
| **Usage** | • Circuits use generic decoders: 1) as true LUTs where we havepre-calculated values (DATA OUT) indexed by DATA IN, or 2) as a replacement for logic functionality<br>• Generic decoder DATA IN & DATA OUT data widths must be at least one bit<br>• You must include a adequate description of a generic decoder if you use it in a circuit |

**Figure 28.25: The Generic Decoder Foundation Module overview.**

## Standard Decoder

The standard decoder is a combinatorial and is a subset of generic decoders. The standard decoder has a special relationship between the SEL inputs and the outputs. The number of single-bit outputs = $2^{(width\ of\ SEL)}$. The output bits have either a one-hot (only one output bit is set) or one-cold (only one output bit is cleared) form. We often describe standard decoders using the notation: 1:2, 2:4, 3:8, 4:16, etc.

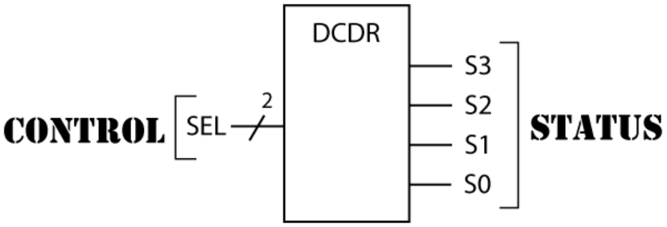
| | Standard Decoder: Device Summary |
|---|---|
| Foundation Notation |   DCDR<br>CONTROL<br>SEL<br>2<br>S3<br>S2<br>S1<br>S0<br>STATUS |
| Input/Output | **DATA IN:** none.<br>**CONTROL:** SEL; selects the form of the output<br>**DATA OUT:** none<br>**STATUS:** Sx; the set of outputs in one-hot or one-cold form |
| Usage | • Circuits use standard decoders when they need to select only one of several outputs to actuate<br>• Standard decoders are a subset of generic decoders |

**Figure 28.26: The Standard Decoder Foundation Module overview.**

## Parity Generator

The parity generator is a combinatorial circuit that establishes a given parity for the aggregate combination of the DATA inputs and PAR output. In other words, the parity generator assigns the parity bit such that the DATA & PAR bits are either odd or even parity. Parity “checkers” circuits are similar to parity generators, where the PAR output indicates the DATA bits are either odd or even parity.

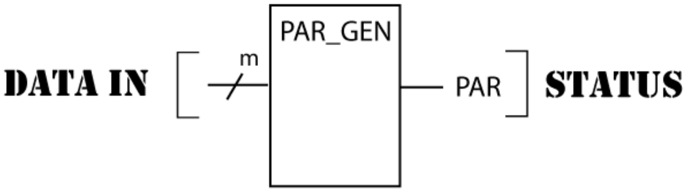
| | **Parity Generator: Device Summary** |
|---|---|
| **Foundation Notation** |  |
| **Input/Output** | **DATA IN:** data used for establishing parity<br>**CONTROL:** none<br>**DATA OUT:** none<br>**STATUS:** PAR bit provides information regarding the parity of the DATA IN bits, which the circuit can then use to establish a given parity |
| **Usage** | • Circuits use parity generators when they need to: 1) establish the parity of the DATA IN bits, or 2) when they need to include a bit with DATA IN to ensure the aggregate set of bits (DATA IN & PAR) is of a given parity<br>• The notion of parity is also associated with a serial stream of bit; in this case, the circuit must have some mechanism to store the DATA IN bit stream |

**Figure 28.27: The Parity Generator Foundation Module overview.**

## Registers

The register is a sequential circuit that to stores bits of data. Registers generally store multiple bits of data; we refer to a 1-bit register as a flip-flop. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. Any data loaded to the register appears on the register's OUT_DATA output after a given propagation delay. Registers also have inputs such as CLR, which clears each of the bit storage elements in the register. Signals such as CLR are often asynchronous, which means the given action occurs immediately upon asserting the CLR signal.

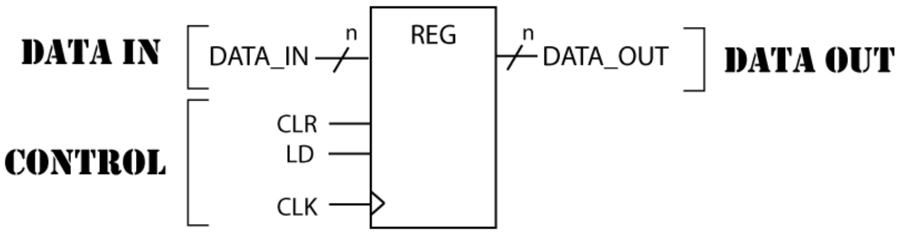
| | **Register: Device Summary** |
|---|---|
| **Foundation Notation** |   DATA IN<br>DATA_IN<br>n<br>REG<br>n<br>DATA_OUT<br>DATA OUT<br>CONTROL<br>CLR<br>LD<br>CLK |
| **Input/Output** | **DATA IN:** data to be synchronously loaded into the register.<br>**CONTROL:** CLK, LD, CLR; The CLK signal synchronizes the loading of data into the register, which happens when both an active clock edge occurs when the LD input is asserted. The CLR input clears each bit storage element in the register (can be either synchronous or asynchronous).<br>**DATA OUT:** the data previously loaded to the register.<br>**STATUS:** none |
| **Usage** | • Circuits use registers when they need to store values<br>• Register loading is always synchronous, while clear-type inputs can be either asynchronous of synchronous depending upon design requirements<br>• A register is a primary component of an accumulators (an RCA is the other component) |

**Figure 28.28: The Register Foundation Module overview.**

## Counters

The counter is a sequential circuit and is essentially a special type of register, which means it retains all the control inputs associated with a register. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. The counter as the ability to count up (adds '1' to stored value) or count down (subtracts '1 from stored value). As with the LD signal, changes to the stored register value are synchronized with the active clock edge. Counter typically have inputs such as CLR, which serves to clear each of the bit storage elements in the register. Counter also have inputs such as CLR, which clears each of the bit storage elements in the register. Counters generally automatically "roll over" when they reach their terminal values, which means that the count transitions from its maximum value to zero when counting up and from zero to its maximum value when counting down.

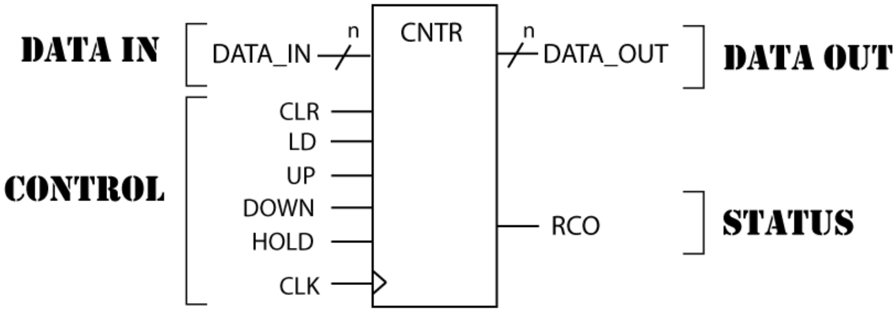
| | Counter: Device Summary |
|---|---|
| Foundation Notation |   DATA IN<br>DATA_IN<br>n<br>CNTR<br>n<br>DATA_OUT<br>DATA OUT<br>CLR<br>LD<br>UP<br>DOWN<br>HOLD<br>CLK<br>CONTROL<br>RCO<br>STATUS |
| Input/Output | **DATA IN:** parallel data for synchronous loading.<br>**CONTROL:** CLK, LD, CLR, UP, DOWN, HOLD. The CLK signal synchronizes the loading of data into the register and the changing of the count, which can either be up or down as controlled by the UP, DOWN, & HOLD signals. The LD signal controls the synchronous loading of new data into the register. The CLR input clears each bit storage element in the register either synchronously or asynchronously.<br>**DATA OUT:** OUT_DATA; the data previously stored in the circuit and possibly modified as loaded to the register. The OUT_DATA signal is the "count" value of the counter.<br>**STATUS:** RCO; establishes when the counter outputs are at the counter's terminal value, where the terminal value depends on the count direction. |
| Usage | • Circuits use counters primarily in two ways: 1) to keep track of how many times something has happened (event counter), or 2) to ensure a circuit does some action a given number of times<br>• There is flexibility in the UP, DOWN, & HOLD functionality as the circuit design requires<br>• The RCO signal is count direction dependent<br>• Not all counter implemenations require the entires set of inputs & outputs<br>• The DATA IN & DATA OUT data width are equivalent |

**Figure 28.29: The Counter Foundation Module overview.**

## Shift Registers

The shift register is a sequential circuit that is a special type of register. The register's SEL input choose operations such as loading of the DATA_IN value to the register, holding, and various flavors of left and right shifts. Most operations are typically synchronous, though the CLR input is often asynchronous. The DBIT signal serves as the new input bit for shift operations, namely the new MSB for right shift or the new LSB for left shifts. We refer to shift registers that do more than one operation as universal shift regsiters (USRs).

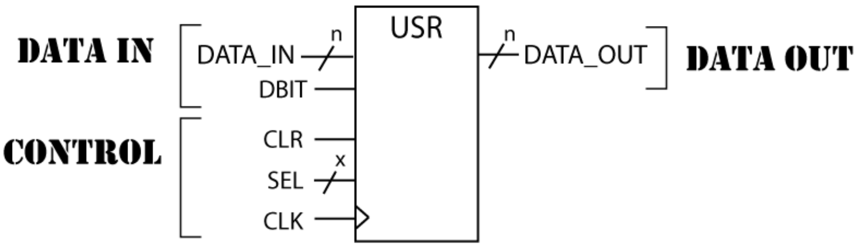
| | **Shift Register: Device Summary** |
|---|---|
| **Foundation Notation** |   DATA IN<br>DATA_IN<br>n<br>DBIT<br>CONTROL<br>CLR<br>x<br>SEL<br>CLK<br>USR<br>n<br>DATA_OUT<br>DATA OUT |
| **Input/Output** | **DATA IN:** DATA_IN is the multibit signal to be loaded into the shift regsiter; DBIT is the single bit of that becomes the new left-most or right-most bit of the stored value for right or left shifts, respectively<br>**CONTROL:** CLK, CLR, SEL. The SEL input selects the functionality of the shift register. Typical shift registers include hold, load, shift-left, and shift-right functionality. The CLK signal synchronizes the loading of data into the register, as well as both left and right shifts. The CLR input clears each bit storage element in the register either synchronously or asynchronously.<br>**DATA OUT:** OUT_DATA; the data stored in the shift register.<br>**STATUS:** none |
| **Usage** | • Circuits use shift registers when they require 1) fast division by two (right shift) or fast multiplication by two (left shift), or 2) when they requires special shift-type bit manipulation<br>• DATA_IN & DATA_OUT bit widths always match<br>• A shift register is a special type of register |

**Figure 28.30: The Shift Register Foundation Module overview.**

## Random Access Memory (RAM)

The RAM is a sequential circuit that allows for the storage of large amounts of data relative to registers. RAM contains three types of input/output signals: address, data (input and output), and control. The IN_DATA signal is the data that will be written to the RAM; the OUT_DATA is the data that is read from RAM. All data reads and write occur at the RAM location specified by the address inputs. The value of the control signals allow the data reads or writes. While memory modules often have many control signals, we only consider a CLK and a WE (write enable) signal in order to simplify this description. Reading data from the RAM is an asynchronously operation; writing to the RAM is a synchronous operation. Read Only Memories (ROMs) have most of the same features of a RAM except for the WE signal. Additionally, reading from ROMs can either be synchronous or asynchronous.

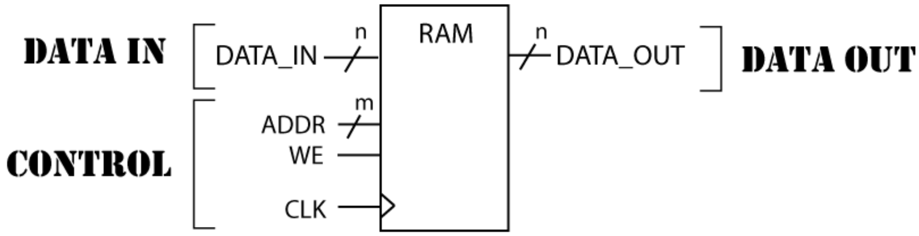
| | **RAM: Device Summary** |
|---|---|
| **Foundation Notation** |  DATA IN<br>DATA_IN<br>n<br>RAM<br>n<br>DATA_OUT<br>DATA OUT<br>m<br>ADDR<br>WE<br>CLK<br>CONTROL |
| **Input/Output** | **DATA IN:** data to be synchronously written to RAM.<br>**CONTROL:** CLK, WE, ADDR; The CLK signal synchronizes the writing of data to the RAM; the RAM stores the value of IN_DATA at the address associated with the value of ADDR on the active clock edge (synchronously) when the WE signal is asserted. Data read from RAM is asynchronous; data appearing on OUT_DATA is the data associated with the ADDR input.<br>**DATA OUT:** OUT_DATA; the data currently stored in the shift register.<br>**STATUS:** none |
| **Usage** | • Circuits use RAMs when they requires a significant amount of easily accessed data storage<br>• DATA_IN & DATA_OUT always have the same data widths<br>• In any given clock cycle RAMs either write or read, but not both |

**Figure 28.31: The RAM Foundation Module overview.**

## Finite State Machine (FSM)

The FSM is a sequential circuit that controls other digital circuits. The FSM reacts to status inputs and issues appropriate control outputs. The FSM's control inputs are "status" outputs form other digital modules, while the FSM's status outputs become "control" inputs to other digital modules. The FSM uses the value of the status inputs to transition through the various states of the FSM on the active edge of the CLK signal. The control outputs can either Moore (outputs a function of state only) or Mealy-type (outputs are a function of both state and status inputs).

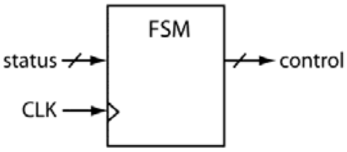
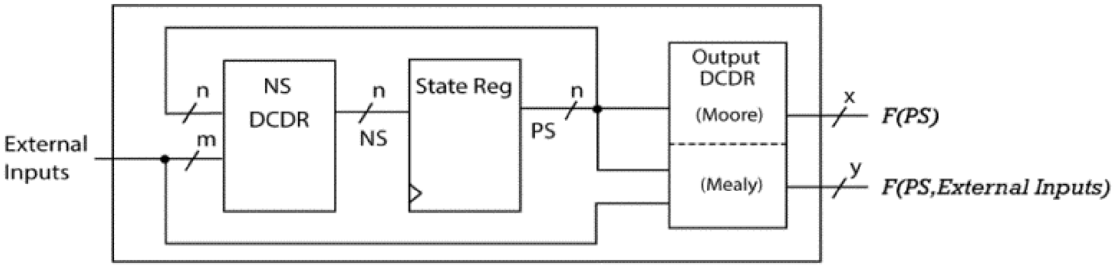
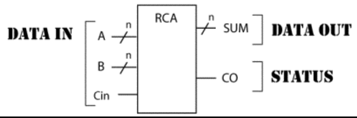
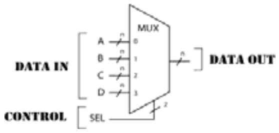
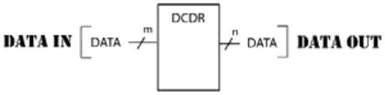
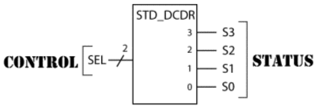
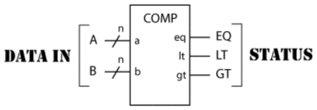
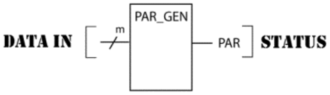
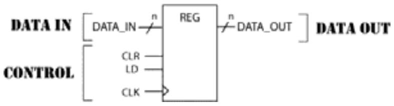
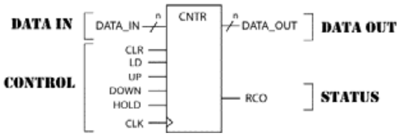
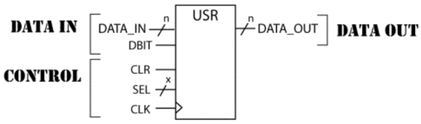
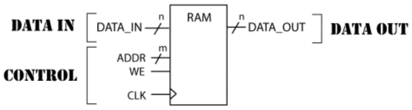
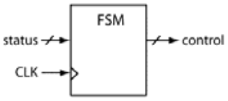
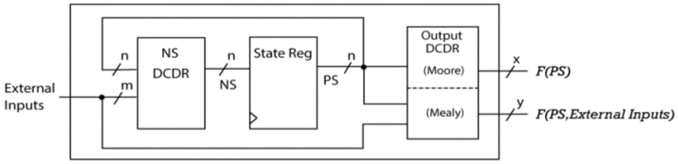
| **Diagram** | **Input/Output** |
|---|---|
|  | **Data In**: none<br>**Inputs**: CLK, status; The CLK signal synchronizes state transitions of the FSM; status inputs are the status outputs of modules external to the FSM. Status input values determine the FSM's state transitions.<br>**Outputs**: control; circuit elements external to the FSM use the control outputs to facilitate data handling; control outputs can be either Mealy or Moore-type. Mealy-type outputs are a function of present state and external inputs, while Moore-type outputs on a function of present state only.<br>**Status** Out: none |
| | |
|  External<br>Inputs<br>n<br>m<br>NS<br>DCDR<br>n<br>NS<br>State Reg<br>n<br>PS<br>Output<br>DCDR<br>(Moore)<br>(Mealy)<br>x<br>F(PS)<br>y<br>F(PS,External Inputs) | |

**Figure 28.32: The Finite State Machine.**

# Digital Designer Foundation Model Cheatsheet

| | | Circuit Diagram | Data IN | Control IN | Data OUT | Status OUT |
|---|---|---|---|---|---|---|
| CombInatorial | **RCA** |  DATA IN<br>A<br>B<br>Cin<br>RCA<br>n<br>SUM<br>CO<br>DATA OUT<br>STATUS | A<br>B<br>Cin | - | SUM | Co |
| | **MUX** |  DATA IN<br>A<br>B<br>C<br>D<br>CONTROL<br>SEL<br>MUX<br>n<br>2<br>DATA OUT | Multiple<br>DATA_IN | SEL | DATA_OUT | - |
| | **Generic Decoder (LUT)** |  | DATA | - | DATA | - |
| | **Standard Decoder** |  CONTROL<br>SEL<br>2<br>STD_DCDR<br>S3<br>S2<br>S1<br>S0<br>STATUS | - | SEL | - | STATUS |
| | **Comparator** |  DATA IN<br>A<br>B<br>n<br>COMP<br>a<br>b<br>eq<br>lt<br>gt<br>EQ<br>LT<br>GT<br>STATUS | A<br>B | - | - | EQ<br>GT<br>LT |
| | **Parity Generator** |  | DATA_IN | - | - | PAR |
| SequentIal | **Register** |  DATA IN<br>DATA_IN<br>CONTROL<br>CLR<br>LD<br>CLK<br>REG<br>n<br>DATA_OUT<br>DATA OUT | DATA_IN | CLK<br>LD<br>CLR | DATA_OUT | - |
| | **Counter** |  DATA IN<br>DATA_IN<br>CONTROL<br>CLR<br>LD<br>UP<br>DOWN<br>HOLD<br>CLK<br>CNTR<br>n<br>DATA_OUT<br>RCO<br>DATA OUT<br>STATUS | DATA_IN | CLK<br>LD, CLR<br>UP/DOWN<br>HOLD | DATA_OUT | RCO |
| | **Shift Register** |  DATA IN<br>DATA_IN<br>DBIT<br>CONTROL<br>CLR<br>SEL<br>x<br>CLK<br>USR<br>n<br>DATA_OUT<br>DATA OUT | DATA_IN,<br>DBIT | CLK, SEL<br>CLR, DBIT<br>DATA_IN | DATA_OUT | - |
| | **RAM** |  DATA IN<br>DATA_IN<br>CONTROL<br>ADDR<br>m<br>WE<br>CLK<br>RAM<br>n<br>DATA_OUT<br>DATA OUT | IN_DATA<br>- | CLK<br>WE<br>ADDR | OUT_DATA<br>- | IN_DATA<br>- |
| | | | | **Inputs** | | **Outputs** |
| | **FSM** |  | - | CLK<br>status | - | control |
| FSM Model |  External Inputs<br>n<br>m<br>NS DCDR<br>n<br>NS<br>State Reg<br>PS<br>n<br>Output DCDR<br>(Moore)<br>(Mealy)<br>x<br>F(PS)<br>y<br>F(PS,External Inputs) | | | | | |

# Digital Design Dictionary

-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 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. Simulations can also use *relative time* (see "relative time").

**Academic Administrators**: A term referring to alien D-bags representing the largest obstacle to actual learning in academia.

**Academic Exercise:** Any amount of work that looks good and keeps you busy but actually has no meaningful purpose in life in general.

**Academic Purposes:** Any process or endeavor that requires time but has no real lasting meaning or lasting effect.

**Academic-Types**: That special type of person who is intent on being successful in academia at any cost and without regards to anyone or anything they damage in the process. The hallmark of an academic-type student is that they 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; they primarily forcus their efforts 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 overpaid academic existence.

**Academonic**: The rallying cry for those who dare to expose the endemic corruption in academia.

**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 synchronizes changes in a circuit's state.

**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 generic term referencing a device that adds numbers. There are many forms of adders in digital design, each with their own set of characteristics.

**Address Lines**: A set of signals associated with an address. Most often, address lines are associated with memory devices, with the address lines being one of the three types of signals associated with memory (data and control lines are the other two). In this case, the address lines provide an index into the memory to read from or write to that particular memory location.

**Adjacency Theorem:** One of the basic theorems associated with Boolean algebra. This theorem facilitates the use of Karnaugh Maps to reduce Boolean functions. We sometimes refer to this theorem as the Co*mbining Theorem*.

**Administrator:** A person who purposely creates problems and/or purposely prevents others from solving existing problems. And if you manage to solve known problems despite the efforts of administrators, they 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 you can solve the problem 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. The world we live in is primarily analog, but computers are primarily digital. Digital design allows the successful interaction between computers the analog world.

**Analog:** A description of something that (such as a signal or data) that we express 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 single 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 of Boolean functions.

**AND/NOR Form:** One of the basic eight logic forms based but not commonly used in digital design. We derive this form 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. We also refer to this form as *sum of product* form or *SOP* form.

**Annotations:** This word relates to "notes". Any time you're describing something, you should include as many annotations as possible. Good designers always include annotations 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 pairs with the VHDL entity to model a digital circuit.

**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. 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 (ALU):** The ALU is generally a datapath submodule, which in turn is a submodule of CPU. The ALU performs all standard bit operations such as arithmetic and logical operations (and shifts and any other way you can think of to tweak bits). The ALU typically generates status of various operations (zero, negative, overflow, carry, pointlessness, parity, etc.) which are individual bits stored outside of the ALU.

**Arithmetic Shift:** A shift register operation on signed binary numbers that protects the 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.

**Assemblers**: A software program that translates assembly language programs into machine code.

**Assembly Language Program Parts**: There are three types of information found in assembly language programs: 1) comments, 2) assembler directives, and, 3) assembly language instructions.

**Assembly Language**: A computer language that uses mnemonics to represent the instructions available to the programmer (the instruction set) for a given computer architecture. The mnemonics give hints as to what the instruction does in terms of the underlying hardware. Assemblers translate assembly language programs into machine code by use of a software program referred to as an assembler. Assembly language is non-portable because a given computer has a finite set of assembly instructions specific to that computer.

**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 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 affects the circuit any time the signal is asserted as opposed to being synchronized to some other signal in the circuit such as a clock signal.

**Asynchronous:** An operation that is asychrounous occurs independent of any clock signal in a given circuit.

**Augend:** A number that adds to another number.

**Automatic Verification**: A term that refers to the notion of a HDL testbench's ability to discern whether a particular HDL model is working properly. The testbench designer can construct the testbench such that the testbench directly states whether the model is working or not; this is opposed to "manual verification" which is the other approach to HDL model verification; (see "manual verification").

**Axiom:** A statement that is universally accepted as true.

## -B-

**Background Task**: A term describing the program code associated with an interrupt service routine.

**Barrel Shift:** A shift operation that shifts more than one bit location in one clock cycle. Barrel shifts come in both left and right-shifting flavors.

**Base:** A synonym for the radix of a given number system.

**BBD:** An acronym used for *black box diagram* (see "black box diagram").

**BCD:** An acronym used for *binary coded decimal*; (see "binary coded decimal").

**Behavioral Style:** A term that refers to using behavioral models in HDL.

**BFD:** An acronym that referring to "brute force design"; this is essentially a pejorative synonym for the "iterative design".

**Bi-Directional Register:** A term that typically describes registers that use the same set of signals for both inputs and outputs. These registers necessarily have extra signals the register uses to control the bi-directionality of the signals to prevent the condition of the signals being simultaneously used as both inputs and outputs.

**Bi-Directional Signals**: A term that refers to the notion that data can flow through a line in two directions (though not at the same time) rather than only one direction. Bi-directions signals are associated with tri-state outputs because a given device cannot generally simultaneously drive a signal and read from that signal.

**Bi-Directional:** A term commonly associated with signals in digital circuits that can support data flowing in two different directions, but not two different directions simultaneously.

**Binary Coded Decimal:** Or BCD, is a number system that uses four bits 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 one of many different methods used to encode the state variables in a finite state machine (FSM). Using binary encoding minimizes the width of the state registers compared to other coding methods (such as one-hot encoding).

**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 values. These symbols are typically '0' and '1' for digital design and computer applications.

**Bit Addressable:** A term that refers to the notion that each bit in a memory has a unique address. More often, bits are only available for reading or writing as part of a larger chunk of data such as a word.

**Bit Mask**: A term that describes a value that "selects" certain bit locations of a word while disregarding other bit locations. The disregarded bits are generally cleared by the bit-masking operation. Microcontrollers require bit masking because most operations in microcontrollers implement operations on the the words only.

**Bit Stuffing:** A phrase used to describe the notion of adding extra bits to a number to increase its width without changing the value of the number. The stuffed bits could be either 1's or 0's depending on the signedness of the number.

**Bit-Banging**: The process of using bit-masking in word-based microcontrollers to use the outputs to control and/or communicate with external peripherals.

**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 often refer to bit-streams as *serial lines*; (see "serial lines").

**Black Box Diagram**: A term that refers to a schematic based model that promiarly uses black boxes to represent the modules.

**Block Diagram**: A modeling approach that uses boxes to quickly transfer high-level knowledge regarding a given system to a human reader of the diagram. Block diagrams are typically hierarchical in nature.

**Block-Style Comments:** A commenting style where multiple lines of code can be commented by using a comment start delimiter and a comment end delimiter such as "`/*`" and "`*/`" in the C programming language and Verilog HDL. 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. Academic administrators find this approach useful because people the control are generally to scared to do anything other than feign interest in the speaker.

**Board-level Digital Design**: A term referring to digital designs comprised primarily of discrete ICs populated on a printed circuit board and 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 digital design.

**Boolean Algebra:** An algebra 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 uses Boolean algebra; we also refer to these as Boolean expressions.

**Boolean Expression**: A term that refers to a Boolean equation.

**Boolean Variable:** A symbolic value that represents one of two values; in digital design, these values are typically '1' or '0'.

**Boring:** A term that describes 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 upper 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.

**Bottleneck**: A term referring to the notion that many devices are attempting access a single device. If the single device is not able to service all of the accessing devices simultaneously, the accessing devices remain idle until the single device is no longer busy. Bottlenecks in system lowers the overall throughput of a system by requiring devices that need services to remain idle.

**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 are later used by higher levels of abstraction in the design.

**Buffer:** A device that accepts a signal as an input and outputs the exact same signal without a change in logic levels. Circuits typically use buffers to increase a signals ability to drive more circuit inputs.

**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 Notation:** The act of showing or describing bundles in circuit models such as schematic diagrams and timing diagrams.

**Bundle:** A term that refers to a set of signals that we arbitrarily group together in a circuit and/or timing diagram; these signals generally share a common purpose.

**Bus Contention:** A term that describes the situation where more than one device is simultaneously driving a given signal or set of signals (such as a bus) on the same lines. Bus contention is a by-product of resource sharing in digital circuit. We avoid bus contention by using devices with tri-state outputs, which can effectively be "turned off" by unasserting the device's enable input.

**Bus**: A term that refers to a set of electrical signals that are grouped together because they share a common purpose. The term also refers to a standard data transmission protocol, which is why we generally refer to a group of signals as a bundle.

**By Inspection:** A term that refers to the notion that we can solve some problems in our heads, thus removing the need for expending extra time explicitly writing down solutions.

**Byte Addressble:** A term referring to the notion that each 8-bit chunk of data in a memory has a unique address, and is thus not bit-addressable.

## -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 most of my time writing this text.

**Central Processing Unit (CPU):** The CPU is generally considered the part of the computer that executes the instructions. Typical submodules of the CPU include the control unit, datapath, program counter, instruction memory, register files, accumulators, ALUs, secondary memory, roach motels, etc.

**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. The acronyms "CS" or "CE" are typically used to represent device chip enables.

**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.

**CISC**: This acronym officially stands for "Complex Instruction Set Architecture" and is generally used to describe computer architectures. CISC computers generally have the following characteristics:

- They contain relatively few general purpose registers
- The instruction word formats are of different lengths
- Instructions require a different number of clock cycles to complete execution
- Some instructions in the instruction set are complex (meaning they can generate a significant amount of processing internal to the architecture)
- System clock rates are generally slower than their RISC counter-parts.

**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 noun, this term refers to the notion that a signal or storage element has been set to '0'. This term is typically used in conjunction with sequential circuits; this term is synonymous with "reset"; (see "reset").

**Clear**: When used as a verb, this term refers to making the value of a signal or storage element a '0'. For example, *"we use the signal to clear the register"*. 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: Complementary 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.

**Combinatorial vs. Sequential Circuits:** The outputs of a combinatorial circuit are a function of the current inputs while the outputs of a sequential circuit are a function of the combination of past inputs. Stated differently, combinatorial circuits do not have the ability to "remember" bits while sequential circuits are able to store values and are this considered to have memory.

**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.

**Compiler**: A computer program that translates higher-level language code into machine code. Compilers generally also produce assembly language code listings which are specific to the target computer.

**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.

**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 I/O:** One of the three main subsections of a computer that allows the computer to interact with the outside world.

**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.

**Context Restoration:** A term describing what a CPU does upon completion of servicing an interrupt. In this case, context restoration refers to the notion that the CPU must return to the state it was in (flags, registers, etc.) before the CPU executed the interrupt service routine.

**Context Saving:** A term that describes what a CPU must do when an interrupt is acted upon. The general notion is that interrupts are asynchronous and can occur while the CPU is executing some important piece of code. In this case, the CPU will save the current state of the CPU (flags, registers, etc.) before processing executing the interrupt service routine.

**Control Lines:** A set of signals associated with controlling a device. Most often the notion of control lines is associated with memory devices, with the control lines being one of the three types of signals associated with memory (address lines and data lines are the other two). In this case, the control lines provide mechanism to read and write from the memory.

**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 implemented 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.

## -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 Lines:** A set of signal associated with some data. Most often the notion of data lines is associated with memory devices, with the data lines being one of the three types of signals associated with memory (address and control lines are the other two). In this case, the data lines provide a path for the data associate with the memory. Flavors of data lines include input data line, output data lines, and bi-directional data lines.

**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.

**Data:** The notion of data is an undefined set of bits (‘1’s and ‘0’s). Once a definition is given to these bits, the data officially becomes information. Once the data is classified as information, the data typically takes on other names such as “address”, “control”, “state”, “op code”, etc.

**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).

**Datapath**: The hardware module that is generally considered to do the number crunching associated with instructions. Submodules of the datapath generally include the ALU, register file, accumulator, various selection logic, etc.

**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’.

**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.

**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 professionals, particularly people who claim to be teachers.

**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 Polarity Indicators:** The use of parenthetical values (H) or (L) to indicated the logic level associated with a given signal.

**Display Multiplexing**: An approach typically used by LED-based 7-segment displays that allows the driving device to control many digits without dedicating a signal to each LED in each segment. The general approach is to connect each type of segments with one signal and give each individual display an on/off control. Using this configuration, display multiplexing only actuates one display at a time, but does so at a rate that makes it appear as if all displays are on at the same time. Multiplexing works for humans because of the notion of retinal persistence.

**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*").

**Driving the Bus:** A term associated with digital circuits that share routing resources. In these circuits, only one device at any given time can output its information to the shared resource. The actual device outputting this information is referred to as the device that is "driving the bus".

**Dumbtarted:** A term applied to technical people who go through life with blinders on; these people typically go into management (or administration in an academic setting) due to their complete lack of technical competence and ongoing reluctance and/or inability to learn new and useful 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-

**Elementary Operation**: A basic operation performed by a sequential circuit. Elementary operations are most often spoken of in terms of registers. Typical operations performed by registers include loading (generally a parallel load), setting (sets all bits in register), clearing (clears all bits in register), shifting/rotating (specifically for shift registers), and incrementing/decrementing (generally for counters).

**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.

**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.

**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.

**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.

**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).

**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. as well as a buttload of routing resources.

**Finite State Machine (FSM):** An abstract machine that defines a finite set of states, actions performed in those states, and a set of rules defining how the machine transitions from state to state. FSM are generally classified as either Mealy or Moore machines. FSMs are one of two major hardware devices that are typically used to control other hardware entities. In these cases, FSM inputs are considered status inputs while FSM outputs are considered control outputs.

**Firmware**: Firmware is a computer program that is written to run on a specific piece of hardware and is thus often associated with embedded systems. Firmware does not refer to the language-level in which the program is written thus can be written in machine code, assembly code, or a higher-level language.

**First Five Things for a New CPU**: When you first examine a new CPU, the five things you should initially examine are 1) the programmer's model, 2) the instruction set, 3) the interrupt architecture, 4) the memory model, and 5) the I/O architecture.

**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.

**Flicker**: An issue associated with display multiplexing where the multiplexing rate is slow enough for humans to note that displays are not "always on".

**Flip-Flop:** A classic sequential circuit that is functionally a synchronous 1-bit storage element. Changes in flip-flop state are synchronized to an edge input to the circuit (generally a clock signal). Flip-flops are also considered synchronous latches and 1-bit registers. The main types of flip-flops are D (data), T (toggle), and JK (who the heck knows) flip-flops.

**Flowchart**: A diagram that uses a few distinctive symbols to model the program flow associated with an algorithm. Computer programmers use flowcharts as an aid to program design and/or documentation support. Flowcharts can and should be hierarchical in nature when appropriate. The hardware analogy to a flowchart is the black-box diagram.

**Forbidden State**: A condition in a sequential circuit that is generally not allowed to happen to ensure an arbitrary characteristic of that circuit.

**Foreground Task**: A term used to describe the program code associated with the main loop in a program. The foreground task is generally all the code that is not initialization code or interrupt service routine code.

**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.

**Fragile**: A label attached to code that is unmaintainable. Fragile code breaks if you attempt to modify it, hence the name fragile. The roots of fragile code are a complete lack of planning of the code as well as modifications made by people who don't know what the f**k their doing.

**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 notion a given logic gate can perform each the three main logic functions: AND, OR, and inversion. NAND and NOR gates are functionally complete while AND, OR, XOR, and XNOR are not.

**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".

**Ghosting**: An issue associated with display multiplexing where an LED is on when it should be off resulting in dimly lit LED showing incorrect information.

**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".

## -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.

**Hand Waving**: A term used to describe literal and figurative gestures to call people's attention to something of hand-waver's choosing. Generally speaking, hand-waving serves to draw people's attention away from problems that were caused by the hand-waver or issues that need attention to other areas that people would not have a strong reaction to. Hand-waving is the approach academic administrators use to dupe the world into thinking they are actually doing something useful.

**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 Drive:** A mechanical storage device capable of store large amount of information. Information in hard drives is stored magnetically on a spinning disc made of a ferromagnetic material; this information is accessed by the classic "read/write heads". Hard drives are not "random access" devices. Hard drives are well known to crash when you need them most. Hard drives are also well known to make great mirrors after you disassemble the hard drive container.

**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.

**High-Z**: Yet another term to express the notion of high-impedance.

**Hi-Z**: Another term to express the notion of high-impedance.

**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 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.

**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'.

**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-

**IEEE Code of Ethics:** A set of guidelines that electrical engineering teachers are required to foist upon their students in order to have their programs accredited by ABET. This is a case of "do as I say; don't do what I do" as most electrical engineering instructors think that "fair" is nothing more than a four-letter word starting with "f".

**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.

**Incidental Memory:** A term used to describe relatively small pieces of memory in circuits such as flip-flops and registers. This term is used to differentiate small memory items from "structured memory" items such as ROMs and RAMs.

**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.

**Information Content**: Information is associated with a set of bits that has some sort of definable meaning. The notion of information content is loosely associated with generic information regarding bit. In the context of "information theory", information content is measured by the notion of "bits", which is based on probability and is thus not related to the familiar digital notion of 1's and 0's.

**Information Theory**: The study of information content of data. One of the key aspects of information theory is to quantify the information content of a data. The metric used for this quantization is the "bit", which is not the same as a "binary digit". Information theory defines a bit based on the probabilities of a data appear in a file, bit-stream, etc.

**Information**: A set of bits ('1's and '0's) that have been given some type of meaning. In other words, once you have more details about some bits, these bits can be then considered information.

**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.

**Interrupt: xxxx**One of the three approaches to performing digitps are typically not part of the IMD process.

**Interrupt Masking: xxxx**One of the three approaches to performing digitps 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”.

**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**: As a noun, this term describes 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.

**Latch**: As a verb, this term mean to the act of a sequential circuit storing data. For example: *“the data is latched into the register”*.

**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.

**Leading Zero Blanking:** Digit-based displays, such as 7-segment displays can display multiple digital. This term refers to the notion that the left-most digits of a given number are not actuated if the values are zero.

**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 every academic administrator on the planet.

**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 vs. Moore FSM Models**: There are two classes of finite state machine model which are referred to as Mealy and Moore "machines", or "models". The external outputs of a Moore machine are a function of state only and output changes are thus considered to be synchronized to state changes in the FSM. The external outputs of a Mealy machine are a function of both FSM state and the internal inputs. Changes in external outputs of a Mealy machine are not necessarily synchronized to the changes in FSM state since they are also a function of external inputs.

**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:** Every digital design problem can have many different but equivalent solutions; the absolute right solution is eternally elusive.

**Mealy's Fourth Law of Digital Design**: The digital design process is circular, not linear. If you think you're going to generate the correct solution with the first pass, you're bound for disappointment. The digital design process is circular; always make going backwards a few steps to fix issues part of the design process. Don't try to make your design perfect from the get-go, make it simple to understand so that you can fix issues as they arise.

**Mealy's Fifth Law of Digital Design**: Model circuits using many smaller sub-modules as opposed to fewer larger sub-modules; as this approach supports testing and increases the chances module reuse.

**Mealy's Sixth Law of Digital Design:** Don't rely on the HDL synthesizer; create your HDL models by having a remote vision of what underlying hardware should look like in terms of standard digital modules.

**Mealy's Seventh Law of Digital Design**: Always first consider modeling a digital circuit or part of a digital circuit using some type of decoder. Decoders in digital design are anything we can describe in a tabular format, so they are essentially look-up tables (LUTs).

**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 quantifiable; medium scale integration is generally known 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 Bandwidth**: Memory bandwidth refers to the amount of data that can be transferred to and from memory. The speed of memory reads and writes are constrained by physical attributes of the device as well as the system in which the device operates in which thusly allow for a maximum amount of information to be transferred to and from the device.

**Memory Capacity**: A term describing how much data can be stored in a sequential circuit. This term is most often used in conjunction with structured memories, in which case capacity is usually measured in terms of bits, bytes, or words.

**Memory Capacity**: The amount of storage a given memory contains. Memory capacity is stated in various forms such as total number of bits, total number of bytes, or total number of words.

**Memory Configurations**: This term refers to the notion that multiple memories can be configured in ways to obtain different memory capacities (number of accessible storage elements) and different storage characteristics (the width or word-length) of each storage element.

**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.

**Memory Levels**: A term that encompasses the various types of memory in a given system. Generally speaking, the lower-level memories are faster but more expensive than higher-level memories. Computer system deal with a trade-off between program execution speed and expense.

**Memory Model**: A term that describes the general way a given CPU utilizes the memory resources it has at its disposal.

**Memory Performance Measures**: Because systems rely heavily on memory, items such as read access times, write cycle times, and memory bandwidth are used to measure the specific performance of memory devices within the system.

**Memory Reading**: An operation that accesses the contents of memory without changing those contents.

**Memory Speed**: A term that refers to how fast a structured memory operates. Depending on the specific type of memory, this term is generally associated with how you can read data from a memory and/or write data to a memory.

**Memory Writing**: An operation that changes the contents of memory.

**Memory**: A term referring to the ability of a digital circuit to store bits. Sequential circuits are digital circuits defined as having memory. Memory in digital circuits can be categorized as either "incidental memory" (flip-flops and registers) or "structured memory" (ROMs and RAM).

**Metastability**: Digital circuits can become metastable when a set-up and/or hold time is not met. Metastability is a loose definition and means the circuit's output is neither high nor low and may remain in that state there for an unstated amount of time.

**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 single integrated circuit. Being complete computers (by definition of a computer), microcontrollers 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.

**Microoperations**: A microoperation is an elementary operation performed on data stored in a register. Microoperations can also include interactions with other registers such as storing the result of microoperations associated with other circuit elements. Microoperations are commonly used in higher-level descriptions of digital circuitry such as computers.

**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.

**Mnemonic**: A set of letters that represents a given operation. Generally speaking, mnemonics loosely describe, in an abbreviated manner, the operation they represent.

**Model**: A model is a representation of something. A more (definitive) descriptive description of a model is a description of something in terms that highlights the relevant information in that thing while hiding the less useful information. The purpose of a model is to quickly transfer important information to the entity reading the model (whether human, or computer, or member of the EE Faculty). Generally speaking, the quality of any model is determined by its ability to transfer information to the user.

**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.

**Mono-Stable Multivibrator**: A device that has one stable state; the stable state can either be the '0' or '1' state. The device's output is only in the non-stable state momentarily before transitioning to the stable state. This term is a fancy name for a device commonly referred to as a "one-shot"

**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 multiplexors 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".

**Narcissistic Personality Disorder (NPD):** A disorder inflicting most faculty members in academia. All faculty members must have this disorder if they plan on climbing up any academic ladder.

**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 your 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”).

**Non-essential 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.

**Non-Volatile**: A term that refers to a sequential circuit’s ability to retain its state (the values stored in memory) when power is removed from the associated circuit. Non-volatile circuits retain their state while *volatile* circuits lose their state information when power is removed from the associated circuit.

**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’.

**Ohm’s Law:** This law forms the basis of all electronic circuits and is commonly listed as “V-IR”, where V is the voltage (Volts), I is current (Amperes), and R is the resistance (Ohms). The equation states that the voltage is directly proportional to both the current and resistance.

**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 one 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 one 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'.

**One-Shot**: The common name for a mono-stable multivibrator. One-shots are used to synthesize fixed-length signals in response to signal events such as clock edges.

**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.

Op-code: A term that is short-hand for "operational code". Op-codes are the bits of an instruction that are used by the control unit to decode which instruction is being executed.

**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 massage the state variables (Mealy and Moore-type FSMs) and external inputs (Mealy-type FSMs only) into the correct output forms to control whatever the FSM needs to control.

**Output Enable**: A signal name that is commonly associated with a signal that allows a device to output a signal or set of signals. When the output is not enabled, the device's outputs typically go into a the high-impedance states. The acronym "OE" is most often used to represent the output enable.

**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 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").

**Periodic**: A term used to describe an attribute of a signal. A periodic signal is defined as having a set period, which represents the amount of time before the signal repeats itself.

**Pig**: A term that completely describes academic administrators.

**Pin Count**: A term referring 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*).

**Polling**: Processors use polling to interface with external devices where the process constantly evaluates the status of the external device in order to determine if the device is in need of services from the processor. Polling is considered to be used in "programmed I/O" and is one of three major types of computer related I/O. Polling is generally associated with inefficient embedded system design in that the system is considered to have low overall throughput when executing a polling loop

**Pop**: An operation associated with stacks where an item is removed from a stack; the stack pointer is appropriately adjusted.

**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.

**Princeton Architecture**: A computer architecture where data and instructions share the same memory space. This architecture is also known as a Von Neuman architecture.

**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.

**Program Counter (PC):** The program counter is a simple counter generally found in a computer's control unit and whose output is generally used as an address that points to the next instruction in program memory to be executed by the program. The PC is typically expected to do standard counter microoperations such as parallel load and increment.

**Program Flow Control Instructions**: Instructions that cause or potentially cause the CPU to execute an instruction other than the instruction following the current instruction. Examples of program flow control instructions are conditional/unconditional branches, and subroutine calls/returns.

**Program Flow Control**: For computer programs to do useful things, they must appropriately respond accordingly to important "events". This response at a low level includes executing different portions of the given computer program. Computer instructions that facilitate any computer operation other than simple incremental execution of instructions from the program memory are generally referred to as program flow control instructions. Program flow control is generally handled by clever manipulations of the program counter.

**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 (PLD):** Any integrated circuit used to create circuits in which the functionality of the internal circuit is not defined until the device is programmed (in this context, the term "program" does not typically refer to a computer programming language). One common type of PLD is the FPGA.

**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.

**Programmed I/O:** One of two main forms of computer I/O. Programmed I/O is characterized by dedicated instructions in the instruction set for performing data input and output. Programmed I/O synchronous in nature as it is associated with an executed instruction.

**Programming Language Levels:** Computer programs can be written on one of three general levels (listed from low to high): machine code level, assembly code level, or higher-level. Higher-level languages include C, C++, C#, Java, Wanker, etc.

**Programming Model**: The programming model, or programmer's model, describes the hardware resources available on a programmable computer-type device that the programmer is able to control via the program control. Program control is provided by the operations described by the device's instruction set and can either categorized as software or firmware.

**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).

**Push**: An operation associated with stacks where data is placed onto a stack; the stack pointer is appropriately adjusted.

**-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 the sign bit; if the sign bit is a '1', then the number is a negative number.

**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.

**RAM**: The acronym officially stands for Random Access Memory; a solid definition for RAM is fleeting due to advances in technology. RAMs are most often characterized as volatile, random access storage devices.

**Random Access**: A memory device is considered random access if it can access any of its contents in a constant amount of time. Devices such as flash drives are considered random access while devices such as tape drives and hard drives are not random access.

**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").

**Read Access Time**: The amount of time required for memory output data to become available after an address and the correct control signals have been provided to the device.

**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 File**: An abstract device that is used to model a given number of general purpose registers that are directly accessible by the given computers instruction set. Register files are typically modeled as multiport RAMs that can read and/or write multiple registers, roughly speaking, in a simultaneous manner.

**Register Transfer Language (RTL):** A syntactically loose approach to specifying a digital circuit that can be modeled as the synchronous transfer of data between sequential circuits such as registers. A RTL statement generally describes a microoperation (or set of micro-operations) generally associated with a digital circuit. The two parts of an RTL statement are 1) the register transfer specification, and 2) the specific conditions that are necessary for that transfer to occur. Generally speaking, only signals necessary for the stated transfer to occur are listed in the RTL statement while non-listed signals are assumed to be “properly handled” elsewhere. Unless explicitly stated, each RTL statement is assumed to occur in one clock cycle though the clock signal is rarely listed as part of the RTL statement. RTL is also known as register transfer notation (RTN).

**Register:** A register is a digital circuit that can store two or more bits of data (one bit of storage would be considered a flip-flop). Types of registers include simple registers, shift registers, and counters. When the term “register” is used, it typically refers to “simple registers” and not counters and shift registers. Registers are typically have both synchronous and asynchronous actions, but typically data storage is synchronous to an active signal edge.

**Register**: An n-bit wide sequential circuit that is primarily known for its ability to store bits. Registers are generally modeled as “n” D flip-flops which share a common clock. Register generally have synchronous parallel load inputs and sometimes other features (elementary operations) such as asynchronous or synchronous presets and clears. Specialized registers include shift registers and counters.

**Regular Structures**: A term that refers to large digital circuits that can be modeled and/or synthesized as a large circuit comprising of many smaller repeated circuit elements. This term is most often used in conjunction with circuits such as PLDs (FPGAs, PLAs, PALs, CPLDs, etc.) and structured memory (ROMs, RAMs, etc.).

**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”).

**Retinal Persistence**: The notion associated with the human visual system that does not allow humans to perceive an off-state of an LED at the exact time the LED is turned off. The notion of retinal persistence is what allows display multiplexing to work for humans.

**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.

**RISC vs. CISC**: The age-old computer argument of which is better that has never been solved. Generally speaking, RICS architectures require more instructions to complete a given operation than a CISC architecture would for that same operation, but those instructions are executed “more quickly” than a CISC architecture.

**RISC**: This acronym officially stands for “Reduced Instruction Set Architecture” and is generally used to describe computer architectures. In actuality, the term has little or nothing to do with the size of the instruction set. RISC architectures generally have the following characteristics:

- They contain a large register
- The instructions word formats all contain the same number of bits (no extended opcodes)

- The instructione execute in the same number of clock cycles
- The instructions generally are not complicated (they don't require great amounts of processing)
- They have higher system clock frequencies than non-RISC architectures

**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".

**ROM**: The acronym officially stands for Read Only Memory; a solid definition for ROM is fleeting due to advances in technology. ROMs are most often characterized as non-volatile, random access storage devices.

**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:** 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-Serving**: The defining characteristic of all academic administrators and most engineering faculty.

**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 specified 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 that sends or receives a contiguous set of bits over a given time period. We typically refer 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**: When used as a verb, this term refers to making the value of a signal or a storage element a '1'. For example, *"the signal sets the flip-flop"*.

**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-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”).

**Set-up & Hold Times**: Digital devices that are edge sensitive (circuit changes state on a rising or falling clock edge) must hold inputs stable (the inputs must not change state) for a certain amount of time before the active clock edge arrives; this time is referred to as the set-up time. Digital devices must also hold the inputs stable for a certain amount of time after the active clock edge which is referred to as the hold time. Failing to meet set-up and/or hold times leads to the circuit going metastable.

**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.

**Shadow Registers**: A term used to describe storage elements for the C and Z flags as part of the RAT MCU context storage mechanism.

**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 a 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.

**Shift Register**: A special flavor of register designed to perform contiguous bit-level transfers (or serial transfers) of data between the bit storage elements of the register. Shift registers generally shift all the storage elements to a contiguous storage element once per clock cycle.

**Short:** A short-hand notion referring to a short circuit; (see *short circuit*).

**Short-Circuit:** A circuit condition that describes a connection between two 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 Extension:** Refers to the act of increasing the bit-width of a signed number without changing the value of the number. Extending the bit-width is different for signed and unsigned numbers.

**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 synthesizable on a programmable logic controller (PLD).

**Software**: In the specific case, software is a computer program that is written in a generic way so that it can run on a more than one type computer. Software does not refer to the language-level in which the program is written and thus can be written in machine code, assembly code, or a higher-level language. In the less specific case, the term software is often means any code written to run on a computer.

**Sorting:** A typical hardware and/or software operation that arranges a set of values based on some pre-determined criteria such as magnitude.

**Spaghetti code**: Programming code that does not follow standard structured programming concepts. Spaghetti code is by definition fragile; it is hard to understand, maintain, modify, and reuse.

**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*).

**Stack pointer**: A term that refers to an entity that contains information that describes the “top of the stack”.

**Stack**: An abstract data type that implement a last-in/first-out (LIFO) queue (or list of things). Stacks can be implemented in hardware or software with hardware implementation of stacks employing the use of a stack pointer to increase efficiency of the device. Stacks are typically used in computer architectures to keep track of hierarchically-nested processes such as subroutines and interrupts.

**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 Decoder**: A standard decoder is a hardware device that implements a one-hot or one-cold output based on a given set of inputs. There is typically a binary relationship between the number of select inputs and the number of outputs and come in such flavors as 1:2, 2:4, 3:8, etc.

**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.

**Start-up code:** The code that is inserted automatically by the assembler as a result of declaring data in the program that requires initialization. The start-up code is typically comprised of instructions that initialize data memory.

**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 Capacity**: A term typically associated with memory devices that refer to how much data can be store within a particular memory or memory system. Storage capacity can be stated in many ways; the two most popular ways are the number of bits the memory can store or the number of words the memory can store.

**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 Code**: Code that can be decomposed into three basic structure: 1) sequence, 2) if-then-else, and, 3) iterative. Structured code is easily understood, maintained, modified, and reused.

**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 Memory:** A term referring to the notion of digital devices with regular structure that can store relatively large amounts of information, such as ROMs and RAMs. Smaller memory devices in digital circuits include "incidental memory items such as flip-flops and registers.

**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).

**Subroutine**: A set of instructions that a computer explicitly transfers to and returns from. In terms of program flow, the program transfers program execution to a set of instructions referred to as the subroutine. When the instructions in the subroutine have completed executing, control is returned to the instruction after the instruction, which caused the program to initially transfer to the subroutine.

**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").

**Switch Bounce**: A condition associated with all mechanical switches were upon actuation, the switch contacts make and break connections several times before the "settling" to the connected state. Switch bounce can last up to 20ms, depending on what source you consult.

**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 described; 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.

**Tape Drive:** A non-random access devices used to store digital data. Data in tape drives is stored on a magnetic media attached to some type of tape that is stored on some type of spool. Tape drives store large amounts of information but access to that information is slow relative to other mass storage devices such as hard-drives or flash drives.

**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.

**Three-State Device:** An electronic device that has the ability to be in a third state that is commonly referred to as the “high-impedance” state. The term “three-state” is synonymous with the term “tri-state”.

**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.

**Throughput**: The throughput of a system is the total amount of useful information processed or communicated during a specified time period. Note that this definition is general. Systems with high throughput are generally desired over systems with low throughput with the exception of administrative systems on university campuses.

**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 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 Gadgets**: 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.

**Top-of-stack**: A term that generally refers to the more recent item placed onto a stack.

**Tri-State Device:** An electronic device that has the ability to be in a third state that is commonly referred to as the "high-impedance" state. The term "tri-state" is synonymous with the term "three-state".

**Tri-State Register:** A register that has the ability to be place its outputs into a high-impedance state.

**Tri-State**: A term that refers to a devices ability to effectively remove itself from a circuit. Thus a tri-state device in a digital circuit can either be high, low, or high-impedance. The notion of tri-stating is used to share routing resources in a circuit; the only possible drawback of tri-stating is that only one device can drive the resource at a given time, otherwise the condition of contention will occur, which is ungood.

**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.

**Universal Shift Register**: A special flavor of shift register that performs actions other than simple one-directional shifts including some or all of the following operations: shift left, shift right, barrel shifts, arithmetic shift, and rotates.

**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".

**VHDL (Very High Speed Circuit Hardware Description Language):** VHDL is one of several modeling systems referred to as "hardware description languages", or HDLs. VHDL is typically used to model digital circuits; the resultant models can be used to simulate circuits, or synthesize circuit implementations on PLDs or silicone.

**VLSI:** An acronym for "very large scale integration"; (see "very large scale integration").

**Volatile/Non-Volatile:** A device is considered volatile if its contents are lost when power is removed from the device while non-volatile devices retain their memory when power is removed and subsequently returned. The term volatile is most often associated with memory devices and PLDs such as FPGAs.

**Volatile:** A term associated with sequential circuits (circuits having memory). The accepted definition of a volatile circuit is that the circuit loses the data it is storing when power is removed from the circuit.

**Von Neuman Architecture**: A computer architecture where data and instructions share the same memory space. The term Von Neuman machine is often used to mean Von Neuman architecture. Von Neuman architecture is sometimes referred to as a "Princeton" architecture.

## -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.

**Weightings:** This is roughly the same term as "weights"; (see *weights*).

**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".

**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.

**Word:** A term used to describe the smallest addressable unit (or chunk of bits) in a memory or memory system.

**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

**Write Cycle Timing**: The amount of time required for data to be written to memory after a valid address, valid input data, and the appropriate control signals have been provided to the device.

**Write Enable:** A name that is commonly associated with a signal that allows a sequential device to store new output's to the device's memory elements. output a signal or set of signals. The acronym "WE" is most often used to represent the output enable.

## -X-

**X:** The symbol typically used to represent input variables in finite state machines.

**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:** A letter that is used for two main purposes in digital circuits. This letter is used to refer to the notion of the "high-impedance" condition of a circuit's output. This letter is also used as a label in finite state machine (FSM) lingo to refer to external outputs.

**Z:** The symbol typically used to represent high impedance. This symbol is also used to represent output variables state machines.

**μ:** An abbreviation used for the metric prefix "micro"; this prefix is used in engineering notation.

# Index