FreeRange Verilog Foundation Modeling

James Mealy © 2019

V2.00

Table of Contents

Pretentions

Legal Stuff

Acknowledgements

The Purpose of this Book

The main purposes of this book are the following.

- To provide readers with an intuitive feel for Hardware Description Languages (HDL): HDLs are interesting animals. The syntax of HDLs looks like programming languages, which can lead to taking a bad approach to learning HDLs. This text with start readers out with the correct mindset so they can develop good digital circuit modeling practices using HDLs.
- To provide readers with a strong foundation of modeling digital circuits using one of the two common HDLs: Verilog (the other common HDL is VHDL).
- To provide readers with an introduction to using Verilog to verify circuits. This means using the Verilog HDL to verify digital circuits are working properly.

As many people know, when you read through a book on Verilog or digital design, the information is rarely presented in the most usable format. Books on Verilog provide you with tons of information starting on page one, which is not always the best approach to learning a new language. Books on Digital Logic that present an HDL do so in a piece-wise manner, which causes the reader to constantly search through the book to find what they need.

The goal of this book is to quickly get the reader up and going on using Verilog to model digital circuits. We achieve this quickness by providing only a subset of the Verilog language, as opposed to inundating the reader with excessive amounts of detail before they're ready to process the information. Our thought is to learn the basics of using an HDL (Verilog) to model digital circuits so that readers can quickly start modeling circuits. Once readers have the experience implementing actual circuits, they will be more quickly understand and utilize some of the more subtle features of the language.

This book is also meant to be a companion manual to another book: ***FreeRange Digital Design Foundation Modeling***, or ***FRDDFM***, which is available without cost from several places. This book presents digital design using a unique approach, which is quite different from approaches found in other digital design texts. I've include as much details as reasonably possible regarding FRDDRM, but you may want to obtain a copy for the full story.

Overview of Chapters

This section provides a brief overview of the information presented in the various chapters in this text. We provide this section to provide the brief big picture for readers and potential readers of this text.

Chapter 1: This chapter the basic purposes of HDLs: modeling for circuit design and circuit verification. This chapter provides an overview programming vs. circuit modeling. This chapter provides a basic guide to helping people new to HDLs start creating good models starting off by having a realistic grasp of the synthesizer's ability to create circuits.

Chapter 2: This chapter outlines the text's approach to introducing Verilog. This text assumes the reader already know digital design concepts and does not describe basic digital circuit operation. This text is also a companion text to a digital design textbook I wrote: *FreeRange Digital Design Foundation Modeling*, which introduces the topics in a unique manner.

Chapter 3: This chapter provides a high-level introduction to Verilog concepts in general. This introduction uses no circuit models but does provide a foundation for circuit modeling in later chapters.

Chapter 4: This chapter presents the first complete Verilog models using relatively simple circuits. We refer to this modeling approach as *direct modeling,* and we limit the models to gate-level designs. This chapter introduces the notion of bitwise operators, and scalar & vector data types.

Chapter 5: This chapter introduces *structural modeling*, which is Verilog's mechanism to support both modularity and hierarchical design. This chapter also introduces the concepts of *unary reduction operators*.

Chapter 6: This chapter provides the background behind behavioral modeling of digital circuits, thus this chapter is primarily about procedural blocks. This chapter specifically differentiates between modeling combinatorial vs. sequential circuits.

Chapter 7: This chapter presents the first Digital Design Foundation Modules with the description of two types of decoders: the generic decoder and the standard decoder. This chapter also covers Verilog issues regarding procedural blocks, variables, procedural programming statements, procedural assignment statements, and equality operators.

Chapter 8: This chapter presents another Digital Design Foundation Module with the description of the multiplexor (MUX). This chapter presents two different flavors of a 4:1 MUX.

Chapter 9: This chapter presents comparators, which his another Digital Design Foundation Module. In support of the comparator, this chapter also presents an overview of relational operators. The comparator is the first module we present with generic data widths, which allow designers to specify the data-width parameter as part of the module instantiation.

Chapter 10: This chapter briefly discusses low-level implementation of ripple carry adders (RCA). The RCA is another Digital Design Foundation Module and is the second module we present in generic format that allows data-width specifications to be part of module instantiation. This chapter also introduces the use of Verilog's arithmetic operators.

Chapter 11: The primary purpose of this chapter is to present how Verilog models sequential circuits. There are two primary methods based on latches and synchronous circuits; this text uses the synchronous circuit approach only. A previous chapter introduced the notion of blocking vs. non-blocking assignments statements; this chapter discusses the topic again in a different context.

Chapter 12: This chapter is a continuation from previous chapter; because registers are parallel combinations of D flip-flops, we the Verilog models for registers are similar to the models for D flip-flops. Registers are another Digital Design Foundation Module; this chapter presents a generic model of a register, which requires module instantiations to specify data-widths.

Chapter 13: This chapter presents the standard finite state machine (FSM) model, and then presents a new model that we use to describe FSMs with using behavioral Verilog models. There are many approaches to modeling FSMs using Verilog; this chapter presents one of these approaches.

Chapter 14: This chapter present counters, another Digital Design Foundation Module. This chapter provides a Verilog model of a generic n-bit up/down counter with other control and status features; instantiations of this device must provide the data-widths.

Chapter 15: This chapter present shift register, the final Digital Design Foundation Module. This chapter provides a Verilog model of a generic n-bit universal shift register with four different operations; instantiations of this device must provide the data-widths.

Chapter 16: This chapter provides a fast overview of testbenches including their theory and some basic examples. Verilog language dedicates a significant portion of the language constructs to non-synthesizable code that finds use in verification. This chapter provides only a brief smattering of the testbench possibilities.

Appendix

Digital Design Foundation Module Templates: This part of the appendix provides templates for each of the various Digital Design Foundation Modules (parity circuits are omitted). The intent of these templates is to have one location for some of the most useful digital circuit to cut & paste and drop into their circuits.

- *Generic Decoder*
- *Standard Decoder (2:4)*
- *Multiplexor (MUX)*
- *N-Bit Comparator*
- *N-Bit Ripple Carry Adder (RCA)*
- *N-Bit Register*
- *N-bit Up/Down Counter with Asynchronous Clear*
- *N-Bit Universal Shift Register*
- *Finite State Machine*

Structural Modeling CheatSheet: Contains one simple example of structural model for simple circuit

Finite State Machine Modeling CheatSheet: Contains one example of Verilog model for simple FSM

Verilog Testbench CheatSheet: Contains two example of simple Verilog Testbenches

1 Introduction to Hardware Description Languages

1.1 Introduction

This chapter provides an overview of Hardware Description Languages (HDLs). Although this text primarily provides a basic understanding of Verilog, this chapter provides no details regarding the Verilog language.

1.2 Purposes of HDLs

The main purpose of any Hardware Description Language (HDL) is to model digital circuits. There are subsequently two main reasons why we want to model a digital circuit.

> **Synthesis**: If we have a model of a digital circuit, we can input that model to another piece of software that uses the model to generate a digital circuit. When we say "generate a digital circuit", we mean either on a *programmable logic device* (PLD) such as a *field programmable gate array* (FPGA), or on actual silicone (such as a digital integrated circuit (IC)). If you're programming a PLD, the software translates the circuit model to a working circuit on the existing digital circuitry embedded into the fabric of the FPGA. If you're designing a digital IC, the software uses your model to "create" actual transistors on silicon.
>
> **Verification**: The synthesis process can be complicated for reasons we'll delve into later. Anytime you use an HDL to synthesize a digital circuit, you're going to always need to test that circuit to ensure your original model was correctly synthesized. We typically use HDLs to create models that verify our other models are working properly.

To summarize, we use HDL for both synthesis and verification. As you'll soon find out, synthesis and verification represent two different worlds in HDL-land. Even though we're using the particular HDL language for both operations, we do so in different ways. Many of the constructs in the HDL exist for verification only; those constructs are thus meaningless for the purpose of synthesis.

1.3 Programming Computers vs. Modeling Digital Circuits

If you're reading this, you more likely than not have some experience programming computers. This being the case, you will notice similarities between the syntax of higher-level computer programming languages and HDLs. Despite these similarities, there are huge differences between the two languages. Moreover, if you're familiar with programming computers and don't understand these differences, you may end up using HDL constructs in the same manner as programming constructs, which usually results in digital circuits that don't work properly when synthesized. This section describes the differences you should be aware of before learning to model digital circuits using an HDL.

1.3.1 Programming Computers

You can program computers at several different levels, but the final step is machine code[1]. You can write machine code by hand, but only crazy people and computer scientists undertake such grunt work. The approach for programming at any level other than the machine code level is to input your program (text-based code) to some software that eventually provides you with the machine code that runs the hardware. The following is a brief overview of various levels of programming and issues associated with them. Yes, there is a point, so keep reading.

[1] Machine code is the 1's & 0's derived from programming instructions. Computer hardware only understands (is controlled by) these 1's & 0's.

1.3.1.1 Assembly Code

This is the step above programming in machine code. Assembly languages are device-dependent, which means every computer probably has a different assembly language. By different, we mean there are different instructions in the instruction set associated with the computer. When you write a program in assembly language, you then input your program to a piece of software we refer to as an *assembler*, which then outputs the machine code.

Assembly languages are quite syntactically structured compared to higher-level languages. As a result, the software that implements the assembler is relatively simple. If you're familiar with assembly code, you know that the assembly programs all sort of look the same, once again, due to the limited syntactic options associated with the language.

Programming in assembly language has many advantages, but also has two main drawbacks. First, the programs can become very long and tedious based on the simplicity of the instructions. Second, unlike higher-level languages, assembly languages are not portable, which means that if you change the computer hardware, you have to re-write the entire program in the assembly language associated with the new hardware.

1.3.1.2 Higher-Level Language Code

Higher-level languages lack the syntactic constraints of assembly languages. This new syntactical freedom solves the portability issues associated with assembly code, but it creates more complexity in the software that translates the higher-level language code into machine code, which we refer to as a *compiler*. Higher-level language code has its own syntax requirements, but they are relatively loose compared to assembly languages.

I like to refer to the compiler as having "magic". Think about it… the compiler takes a page full of text that is only lightly constrained by syntax rules, and converts the text to machine code. In other words, it takes something that is rather unstructured and imposes a meaningful structure on it. Note that compilers must be able to translate an expression that runs multiple pages long to a set of instructions on a processor that only implements a single relatively simple instruction at a time.

Is it magic? No (but it seems like it). Is it relatively complex? Yes. Are compilers written by humans? Yes. Do humans make mistakes? Yes. Do compilers always do the right thing (generate the correct machine code) in every instance? No. What a great world it would be if compilers generated the correct machine code 100% of the time[2]. The main point here is that as you pile more responsibility onto the entity that translates your code into machine code, the more chance you're going to have problems. There are work arounds for these problems in software-land, but that's a topic for another book.

1.3.1.3 Modeling Digital Circuits

Using an HDL to model a digital circuit is similar to programming a computer in that you start with writing text-based code into an editor, and eventually send the result off to another piece of software. Additionally, you HDL code may look similar to higher-level language code, but appearance is its only similarity.

After you an HDL to model your digital circuit, you input that model to a piece of software we refer to as a *synthesizer*. The synthesizer then translate that code to some other form that is on its way to becoming an actual digital circuit. Although a compiler had significant amount of responsibilities, the synthesizer has much more. Recall that when you're writing computer programming code, the code represents instructions to a computer; the computer interprets those instructions one at a time and in a sequential manner. The "one at a time" and the "in a sequential manner" significantly reduces the complexity of the compiler's task.

The synthesizer's main task is to interpret a page full of HDL code and deliver something close to a digital circuit. A digital circuit comprises of many modules simultaneously doing all their assigned tasks. Think about it: the HDL model uses a text on a page to describe a circuit that has many different modules operating in parallel. The point here is that the "magic" associated with a synthesizer is well-beyond the magic associated with a compiler.

[2] Anyone who's programmed embedded systems knows to never trust the compiler. When your embedded system program has a bug, you always consider that it may be a problem generated by the compiler.

The generation of hardware from text-based descriptions is daunting. As a result, the HDLs provide you with many "knobs"[3] to tweak in hopes that you can correctly instruct the synthesizer to generate the circuit you've described with the HDL. I still like to joke that the synthesizer is magic, but it not; it only seems that way. The moral of this story is that you must have a working understanding of how the synthesizer handles the various knobs associated with the HDL. If you understand the basics of the HDL's knobs, you're circuit is going to have a higher probability of working properly, and without too much random knob-tweaking and prayer. This text is going to tell you about the main knobs associated with the Verilog HDL. We're not going to tell you the entire story, but we'll give you a great start. You can use you newfound knowledge of Verilog to write your own happy ending to the story.

1.4 Two Purposes of HDL

HDLs have two main purposes that in intimately related. We use HDLs to model and verify digital circuits. A model is a description of a circuit that we can use to synthesize actual hardware. Once we have that hardware, we find ourselves hoping that the synthesizer correctly interpreted our model and generated a circuit that works as we expect. Once again, the synthesis process is complex, as the synthesizer must interpret many knobs in the model. So naturally, the next step in the process is to test your synthesized models to ensure they are working properly. We refer to this testing as verification. We verify circuits by writing other HDL models that allow "exercise" the circuit in such a way that we get a good feeling that the circuit is working properly.

1.4.1 Verification of Digital Circuits

This is the funny part. Most HDLs first started out as a means to verify circuits. Not surprisingly, the HDLs are a significant amount of constructs that are meant only for verification purposes. The main purpose of these constructs is to design models that simulators use to facilitate the automatic verification of circuits. In other words, digital designers can't use these constructs to synthesize hardware. The even funnier part is that the HDL models in the verification process are surprisingly similar to higher-level language programming code, which is a direct result of the non-synthesizeability of the HDL code. The level of "magic" that we're joking about is much lower for a model used for verification than a model used for synthesis. Sort of strange.

1.5 Digital Design and Modeling Digital Circuits with an HDL

The two main tenets of modern digital design are that designs are modular and hierarchical. Modern digital designs are thus more about increasing the efficiency of the digital design process by keeping designs abstracted away from low-level design. Both VHDL and Verilog fully support both modularity and hierarchy with what we refer to as the *structural modeling*, a topic we cover in a later chapter. Here are some more details to be aware of:

> **Modularity** -Put Your Design in a Box: Digital designs are modular in nature. This of course means that you don't have to make your designs modular, but doing so increases the readability, understandability, and testability of your models. As you'll soon see, we create complex digital circuits by placing and connecting modules; we typically don't have that many modules that we use in digital design, so we find ourselves re-using the same modules over and over. In this case, we strive to re-use previously designed modules as we typically know them to be fully functional.
>
> **Hierarchicality** – Put Your Boxes Inside Other Boxes: There are many approaches to modeling digital circuits. Software tools don't care how you model your circuits as they have their own approach to interpreting the circuits. But because digital circuits can quickly become large and complex, we use hierarchical models to makes these complex circuits more understandable to human readers. We refer to a non-hierarchical design as a *flat design*; the circuit is going to work as it's supposed to, but the models are hard for human readers to

[3] OK, "knobs" sounds good, but what we really mean by a knob is a feature included in the HDL language, which are necessarily syntax-based features. The HDL includes these features to provide you with flexibility in the description of the hardware.

understand. When humans can't understand designs, they can't efficiently correct problems with the design, they can't quickly and easily modify the design, and there is much less chance of the coder being re-used in other designs.

1.6 The Golden Rules of Modeling Digital Circuits with HDL

What we've been trying to tell you is that you, the digital circuit designer, must work with the tools in order for you to synthesize a circuit that works properly. When you know the shortcomings of the various design tools, you can work around them. If you don't know the various quirks of the tools, particularly the synthesizer, the tools will work you around.

It's not that big of deal to work with the tools. The good news is that the more designs you successfully complete, the more familiar you become with the tools. You quickly develop your own style and technique and working with HDLs and the associated tools. To help you along on this journey, here are the rules that you should always follow, or at least follow them until you develop some digital circuit modeling swagger

Golden Rule #1: Keep models simple by leveraging modular and hierarchical design.

Justifications:

- You can better control how the synthesizer interprets your model when it simple, which means when you use the simplest possible set of knobs in your design. Complex designs require more knobs; more knobs give the synthesizer more options, which means more ways to implement a circuit that does not work properly.
- We categorize good digital circuit models as boxes exchanging information with other boxes; these individual boxes make no attempt to do everything. HDL syntax is powerful enough to allow you to describe large complex circuits with a single box, but that design approach gives the synthesizer enough rope to hang itself with (and you too). Moreover, a design comprised of many smaller boxes is easier to test.

Golden Rule #2: Don't rely on the synthesizer to make your circuit work for you.

Justifications:

- The synthesizer is magic and powerful, but it is also somewhat stupid in that it has rules of how it handles different model constructs, but it has no idea what exactly you want you circuit to do. Don't rely on the synthesizer to properly interpret your models.
- Don't become comfortable when you see HDL constructs with similar syntax to programming constructs. In spite of the fact the constructs appear similar, they are performing two different tasks: HDLs model hardware, higher-level languages direct the operation of processors.
- Always have a general idea of how the circuit you're design appears on paper before you start modeling with HDL constructs.

Golden Rule #3: Design knowing that you'll need to verify the design (namely *verification*).

Justifications:

- The synthesizer follows a set of rules to do what it needs to do. As a result, you're always going to need to test your design. Because you always test your designs, you make the testing process easy for yourself by making your designs testable. Roughly speaking, the more modular and hierarchical you make your design, the easier it is to test.

Golden Rule #4: Neatness counts in HDL models.

Justifications:

- Part of testability means that you need to look back at your original models to fix any problems that the testing process may have discovered. If your code is well-structured, you can fix issues much faster than if your code is poorly formatted. Strive to make you code readable and understandable to humans, as humans are the one who fix your code when there are issues. Ask for a style file associated with your HDL for you to follow, and follow it.
- The size of your model does not necessarily correlate to the size of the synthesized circuit. It often does in the programming world, but it rarely does in the synthesis world. Use all tricks at your disposal to make your models readable by humans (such as utilizing whitespace and proper indentation of the code).

1.7 Chapter Summary

- HDLs provide one approach to modeling digital circuits. The two main purposes of modeling digital circuits using HDLs are to synthesize digital circuits or to verify digital models are working properly.
- Using an HDL to model a digital circuit often looks and feels like programming a computer, but there are significant differences between modeling digital circuits (HDLs) and programming computers (various computer programming languages).
- Modeling digital circuits using an HDL and using those models to synthesize a digital circuit is significantly more challenging than programming a computer. The complexity of a synthesizer is orders of magnitude greater than that of an assembler or compiler.
- Because synthesizers are so complicated, verification of the model is a significant part of the digital circuit design process using an HDL.
- The Golden Rules of HDLs:

  Golden Rule #1: Keep models simple by leveraging modular and hierarchical design.

  Golden Rule #2: Don't rely on the synthesizer to make your circuit work for you.

  Golden Rule #3: Design knowing that you'll need to verify the design.

  Golden Rule #4: Neatness counts in HDL models.

2 Digital Design & Digital Design Foundation Modeling

2.1 Introduction

Although the main topic in this text is Verilog, any HDL would be impossible to discuss without a proper context. While we're assuming the reader is already familiar with the rigors of digital circuit design, one of the goals of this text is to support Digital Design Foundation Modeling (DDFM), which is how we refer to our new approach to teaching digital design. This chapter provides an overview of both digital design and DDFM. We suggest that you obtain a copy of FRDDFM to obtain the full story on DDFM.

2.2 Digital Design Overview

Digital design is the process where you create a digital circuit to solve a given problem. A digital design solves problems by having the outputs react to the inputs in a manner such that it solves the given problem. Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it solves the 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.

2.2.1 Basic Tenets of Digital Design

The two basic tenets of digital logic are:

1) 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.

2) 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.

2.2.2 Digital Circuit Types

There are two basic types of digital logic circuits:

1) Combinatorial Circuits: circuit outputs are a function of the circuit's inputs.

2) 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 2.1shows 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.

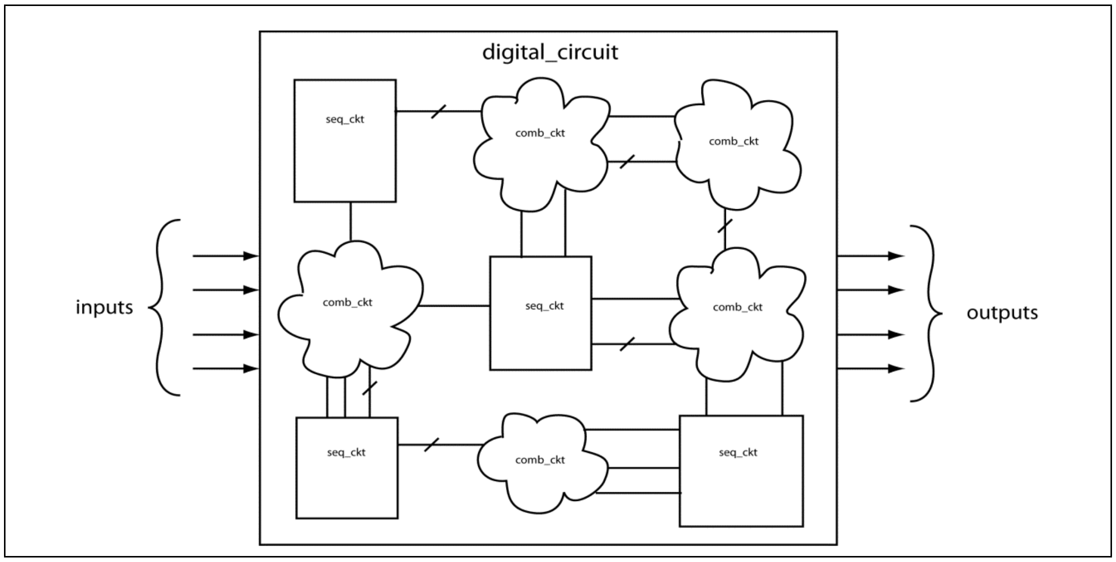
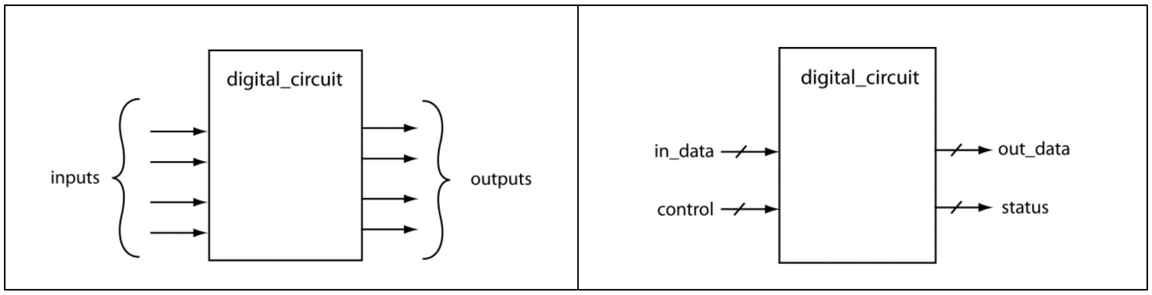


Figure 2.1: A basic logic circuit.

Figure 2.2(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 2.2(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 2.2(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 2.2: Models for a basic logic circuit (a), and a more refined basic digital logic circuit (b).

2.2.3 Finite State Machines (FSMs)

We use a finite state machines (FSMs) to control the flow of data through digital circuits. The FSM interprets the status signal outputs of the circuit modules and issues control signals to those circuit modules. Figure 2.3 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.
- 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-type outputs) or state and status inputs (Mealy-type outputs). Figure 2.4 shows models for the Moore and Mealy-type FSMs.

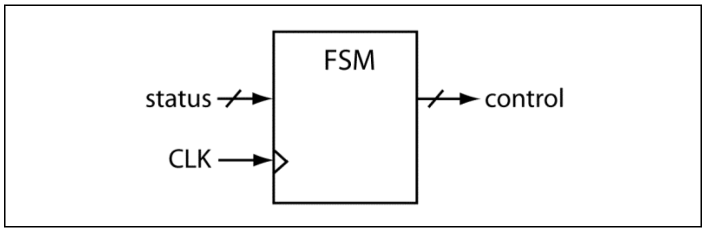


Figure 2.3: A black box model of a FSM.

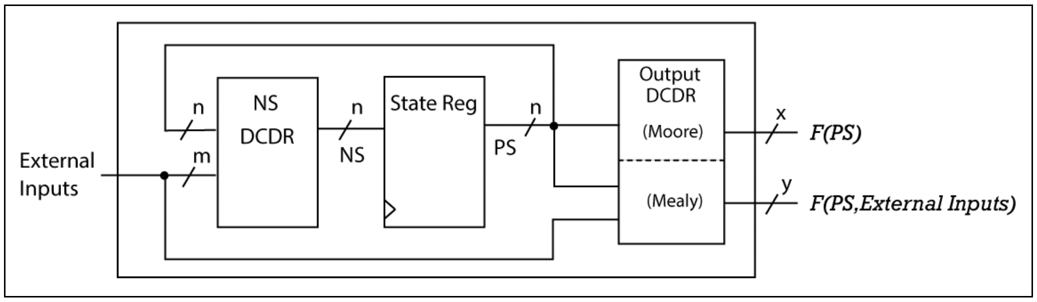


Figure 2.4: The FSM model showing the two types of outputs (Mealy and Moore).

Figure 2.5 shows a modified version of Figure 2.2 that includes an FSM as a control element. The main purpose of the FSM is to control the flow of data through the circuit in such a ways as to solver the given problem.

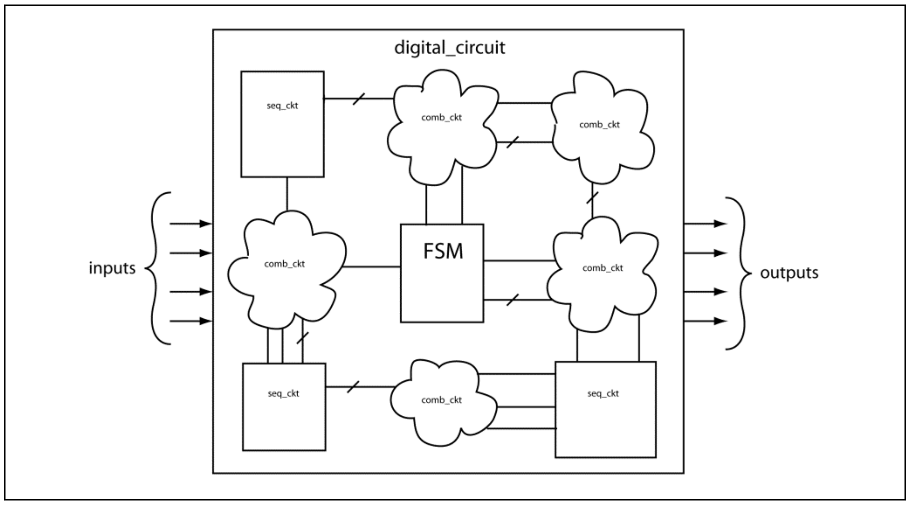


Figure 2.5: A basic logic circuit controlled by FSM

2.2.4 The Three Approaches to Digital Design

Part of DDFM includes categorizing digital design into three different approaches. 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.

2.3 Principles of 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.

2.3.1 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***[4] 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)**

Figure 2.6(a) shows the standard approach to modeling digital circuits, where we classify all digital circuit signals as either inputs or outputs. Figure 2.6(b) and Figure 2.6 (c) shows how DDFM further classifies inputs

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

and outputs by first separating digital modules into “controlled circuits” and “controller circuits”. Figure 2.6(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 2.6(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 2.6(c) inputs the status signals of controlled circuits and manages the controlled circuits by outputting the appropriate control signals to control the controlled circuits.

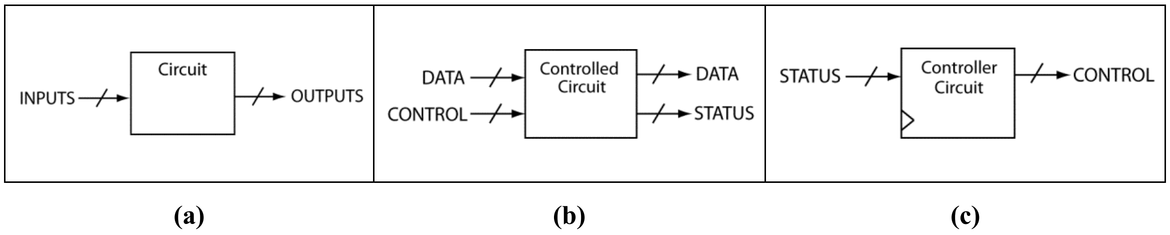


Figure 2.6: Old digital circuit model (a); models for controlled (b) and controller circuits (c).

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 2.7 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 2.7 includes three different module shapes showing that controllable modules can either be combinatorial or sequential circuits, as well as off-the-shelf computer peripherals.

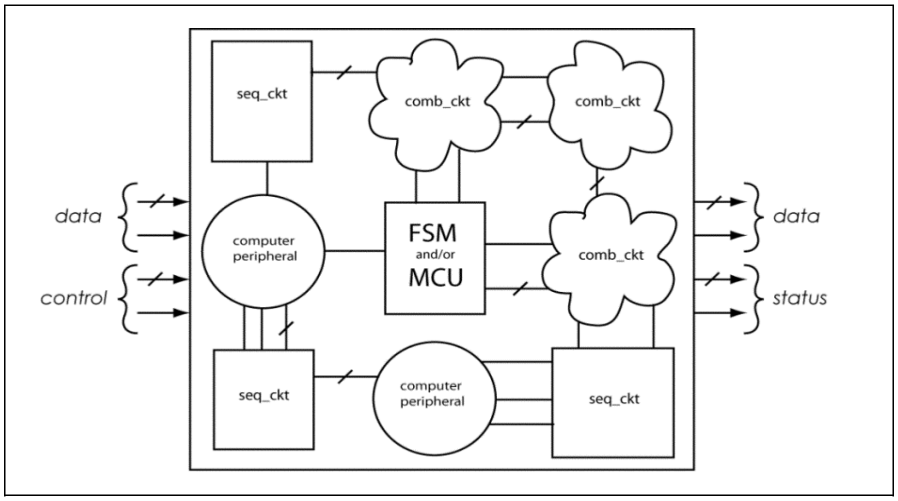


Figure 2.7: Our unifying digital circuit model.

2.4 Chapter Summary

This chapter outlined our approach to digital design; the outline was essentially a summary of how we present introductory digital design in the *FreeRange Digital Design Foundation Modeling* textbook. We presented this outline as it provides the justification for presenting a majority of the material in this text.

- The two basic tenets of digital logic are:
    1) Digital logic circuits are hierarchical
    2) Digital logic circuits are decomposable into a few basic digital circuits
- The two types of digital circuit are combinatorial circuits and sequential circuits. Sequential circuits can store data (thus have state); combinatorial circuits can't store data.
- We often use Finite State Machines (FSMs) to control digital circuits.
    - The three main subsections of a FSM are:
        1) Next State Decoder
        2) State Registers
        3) Output Decoder
    - FSMs can have Mealy-type outputs (a function of state and external inputs) or Moore-type outputs (a function of state only)

- 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**

3 Introduction to Verilog

3.1 Introduction

We previously presented a high-level conceptual view of important HDL issues and their relation to digital design. This chapter moves closer to actual Verilog by presenting some of the higher-level HDL concepts related to both Verilog and basic digital design.

3.2 History

Verilog was created sometime in the early 1980s as a way to standardize the process of designing digital circuits. Verilog started out as a method to simulate and verify digital circuits, but it later was developed into a tool used to synthesize digital circuits. Although it initially was a proprietary language, it entered the public domain in 1990. The continued development of Verilog eventually led it to be adopted as an official standard: IEEE 1364-1995 in 1995. An enhanced version of Verilog was released in 2001, which was also an official standard: IEEE 1364-2001. The newer version of Verilog provided many language enhancements, but remained backward compatible to older Verilog versions. System Verilog is the latest and greatest thing out there; this version support an extensive list of software-like features primarily designed for circuit verification.

3.3 The Concept of Concurrency

Modeling digital circuits is a matter of creating a bunch of boxes (digital circuits) and having those boxes work with each other in such a way as to solve the problem at hand. Note that an HDL model is simply a text file that containing various HDL constructs that follow the syntax of the given language. Our models are text files that roughly read from the top of the page to the bottom of the page; we send these files to the synthesizer and it generates a digital circuit.

The incredible importance of the previous paragraph probably snuck right past you. The moral of that paragraph is that the basis of all HDLs is the notion of *concurrency*. The various constructs in HDLs allow you to model boxes; these boxes are digital circuits that operate in parallel. This means that the order that you list the boxes in your HDL model does not matter. We necessarily need to list them in some order, as it is a text file, but there order of appearance of the boxes in the model does not matter. This is in stark contrast to programming a computer, where the order of the instructions does matter. That's the cool thing about hardware: it's naturally parallel. Attempting to program a computer with more than one processor is a pain in the arse.

To be perfectly clear, the order of the appearance of the boxes does not matter in HDLs, but the statements describing those boxes have an order that does matter. HDLs are full of various types of constructs that describe boxes, which you describe with various types of statements with that HDL; the order of these statements describing a single box does matter. The order of the box descriptions appearing in the text file does not matter. To be even clearer, the synthesizer does not care about the order of boxes, but a human reader of your code does, so you always try to make the order seem as meaningful as possible.

3.4 Modeling Digital Circuits with Verilog

The main purpose of Verilog is to model digital circuits. There are two major types of digital circuits: combinatorial circuits and sequential circuits. Roughly speaking, sequential circuits have the ability to store information while combinatorial circuits do not. This is an important distinction when using an HDL because of the unique way that HDLs represent the two types of digital circuits. We wait until a later chapter to get into those details.

There many approaches you can use to model a digital circuit using Verilog, but we won't discuss all of these ways. What we will do is discuss the two ways allow you to quickly start modeling digital circuits. The following is an overview of the two approaches to modeling we discuss in this text.

3.4.1 Direct Descriptions

The term "direct descriptions" in not associated with Verilog outside of this text. We created this term in order to clearly delineate the various design approaches in Verilog. We use the term "direct descriptions" to refer to Verilog models that are low-level descriptions of circuits, which means describing circuits on the gate level. There are two main approaches to describing gate-level logic using Verilog, we'll only discuss the more useful approach in this text.

The advantage of describing circuits using direct descriptions is that you provide the synthesizer with few options as to how it synthesizes the circuit. This allows your circuit to work as you planned (if you in fact correctly described the logic) without battling the synthesizer. Using our knob terminology, the direct modeling of circuits gives the designer no knobs to tweak, and thus the synthesizer has few decisions to make that it can potentially get wrong. The drawback of using direct descriptions of circuits is that it is an inefficient way to describe anything other than simple digital circuits. The power of HDLs does not lie in its ability to model digital circuits using basic logic.

The moral of this story is that you should use direct descriptions when you can, but quickly move onto more powerful modeling techniques when the circuit is anything but simple. Additionally, in this context we consider any sequential circuit to be non-simple. So if you need to model a sequential circuit, don't bother using direct descriptions. Moreover, this text only works with synchronous sequential circuits; we'll fill in the details in the chapter that discusses sequential circuits.

3.4.2 Behavioral Descriptions

The main point behind behavioral descriptions of digital circuits is that you are no longer constrained to describing circuits using logic. The power of HDLs lies in behavioral descriptions of circuits. The various constructs in HDLs provide you ways to describe how a circuit should "behave" rather than describing the circuit directly with gate-level logic. The classic example of this is the circuitry behind the edge-triggering of a synchronous circuit. It's an interesting circuit, for sure, but I would not want to describe it using a direct description.

Part of the power of behavioral modeling comes from the various constructs that the HDL provides to help you adequately describe digital circuits. What this means is that HDLs provide you with many *knobs* you can tweak with the intent of helping you correctly describe a digital circuit. As you would probably guess, the drawback is now that the digital designer has all these knobs to tweak, the probability the synthesizer generates a circuit that you intended goes down.

Having many knobs to tweak thus emphasizes the notion that you need to know how exactly to tweak those knobs to have the synthesizer generate the correct circuit. If you tweak the wrong knob, the synthesizer generates the wrong circuit. Once again, this text is about helping you always understand which knobs to tweak to keep the synthesizer under control, particularly when you're first learning the language.

3.5 Verilog Invariants

There are some characteristics about modeling digital circuits with Verilog that never change. We list these items before we discuss the modeling abilities of Verilog because these items are simple and quite applicable. The following items are relatively straightforward; the best approach is to learn them now so that they don't cause you problems later.

3.5.1 Coding Style & Documentation

Your primary mission using Verilog is to make that synthesizer and/or simulator happy, which you do by following the basic syntax rules associated with Verilog. The secondary mission is to create Verilog models that make the humans who may have to read your code happy. The Verilog language provides you with a great deal of flexibility in the way your write Verilog code; your mission is this to use this flexibility to create models that are easy to understand by humans. If you're successful in this pursuit, your code is easy to understand, debug, reuse, and modify.

3.5.1.1 Identifiers

We use the word *identifier* to refer to the names of variables, module names, and other items in Verilog. There are only a few rules regarding identifiers.

- Identifiers can use any letter or any decimal digit
- Identifiers can use underscores ("_") or dollar signs ("$")
- Identifiers cannot begin with a digital

The unstated Golden rule of identifiers is to choose them such that they expedite the understanding of the circuit they are modeling to humans reading the code. This means you must specify identifiers such that they are *self-commenting* in nature. For example, a module named "X" provides the reader with no information while a module named "ADDER" provides a meaningful written description of that module to the human reader. In most cases, the synthesizer doesn't care what you name it.

3.5.1.2 Case Sensitivity

Verilog is case sensitive, which means the identifier "x_signal" is different from the identifier "X_signal". Despite the case sensitivity of Verilog, you must keep the differences in identifier names significantly more different than the case of individual characters in the identifier.

You need to develop your own style as far as using case in the specification of identifiers. This means, for example, that you should name all your modules starting with a capital letter followed by all lower-case letters. Then choose a different case style for modules inputs and outputs. Choosing two different styles and using those styles consistently enhances the readability of your Verilog modules.

3.5.1.3 White Space

The notion of white space includes spaces, and blank lines. The Verilog language ignores all white space, which provides the person writing the code with an opportunity to enhance the readability of the Verilog code. For some reason, nubile Verilog coders have the tendency to attempt to make their code as compact as possible (possibly an artifact from computer programming), which always reduces the readability of the code, (so don't do it)[5]. Here are a few of the more obvious uses of white space in your code

- Your Verilog code must follow accepted indentation practices
- Use blank lines to separate different "ideas" such as modules in the vertical direct
- Use spaces to separate different ideas in the horizontal direction

You should never use actual tabs in your code. The problem is that printers and the text editors of people on your team interpret the tabs differently. It's always best to use the space bar rather than the tab key.

3.5.1.4 Comments

Comments are messages from the designer of the model to another human reading the model. The synthesizer and simulator ignore the Verilog comments. The general idea behind comment usage is that you should fully comment everything in your code that is not patently obvious. While it is possible to have too many comments in your code, it does not happen too often[6]. Verilog has two types of comment; you may notice these are the same comments associated with the C programming language.

[5] You can argue that there is a correlation between code length and efficiency of machine code generated for programming languages, but this is not true for HDLs. Once again, the synthesizer does not care how it looks so long as it follows syntax rules. The human attempting to debug your code is the one who cares.

[6] The bigger issue is to get people to put any comments in their code at all.

- Block Comments: These are multi-line comments, where the synthesizer ignores all text between the occurrence of a "**`/*`**" until the occurrence of a "**`*/`**". You can't nest this type of comment.
- Single-Line Comments: The synthesizer ignores any text following "//" until the end of the line.

3.5.1.5 Parenthesis

Like most languages, Verilog does have precedence rules. Like most digital designers, I know a few of the more obvious ones, but I don't know most of them. You should strive to make your code readable to humans, which means that not all humans know or have easy access to the precedence rules. The better option is thus to rely on the liberal use of parenthesis to define your coder rather than relying on the precedence rules of the language.

3.5.2 Statement Termination

All true statements in Verilog are terminated with a semicolon. This is easy to state, but harder to put into practice when you're first learning Verilog. The tendency is to put semicolons where they do not belong, which prevents your code from properly synthesizing. I suggest that when you see an example of a new Verilog construct, to take note of how and where the code uses those semicolons.

3.5.3 Reserved Words

The Verilog language contains many keywords, which means they have special meaning within the language. If you attempt to use these words as identifiers, you will generate an error. Because Verilog is case sensitive, it is possible to use these words with different case configurations, but doing so would represent bad coding practice. Specifically, the synthesizer would not care; but human readers of your code will make dolls that resemble you and stick pins in it. Table 3.1 shows the partial list of Verilog reserved words.

always	design	fork	library	reg	vectored
and	disable	function	medium	release	wait
assign	edge	generate	module	repeat	wand
automatic	else	genvar	nand	scalared	while
begin	end	if	negedge	signed	wire
buf	endcase	ifnone	nor	small	xnor
case	endgenerate	include	not	specify	xor
casex	endmodule	initial	or	table	
casez	endtable	inout	output	task	
cell	endtask	input	parameter	time	
config	event	instance	posedge	tran	
deassign	for	integer	primitive	tri	
default	force	join	real	unsigned	
defparam	forever	large	realtime	use	

Table 3.1: Partial list of Verilog reserved words.

3.6 Chapter Summary

- Verilog has a long and rich history. The language was created in the early 1980's as a simulation tool, but was recognized as so powerful, that it later became a tool for circuit synthesis. The first Verilog standard was established in the early 1990's, which was later modified and enhanced in 2001 with a new standard. The latest Verilog spinoff includes System Verilog, which adds many software-type features to aid in both synthesis and verification of circuits.
- The main theme behind HDLs is concurrency, which means the various HDL constructs are interpreted as concurrent, or operating in parallel. Parallel operation is the hallmark of hardware design, which is in stark contrast to the inherent sequential operation of software.
- HDL uses two approaches to modeling circuits: Direct Descriptions and Behavioral Descriptions. Direct descriptions primarily model circuits in terms of the underlying circuit elements. Behavioral description use various constructs associated with the HDL to describe the operation of the circuit without needing to concern designers with the lower-level implementation details.
- The Verilog language allows designers considerable freedom in the appearance of Verilog models. Digital designers must use this flexibility to enhance the readability and understandability of their models for the humans who read their models.
- Digital designers using Verilog should strive to make good circuit models by working with accepted conventions in the appearance and structure of their Verilog models. Characteristics such as proper use of white space and intelligent use of identifier naming are two primary area of concern that makes for good Verilog models.

4 Direct Modeling

4.1 Introduction

There are two basic approaches to using Verilog to model digital circuits. We're going somewhat outside accepted vernacular here, but we do so to help you understand how to quickly come up to speed modeling digital circuits. In rough terms, the two approaches are high-level and low-level, where the high-level approach refers to the behavioral modeling of circuits while the low-level refers to describing circuits on the gate-level.

We've create the term "Direct Modeling" as it better differentiates between the two main types of behavioral modeling. In a strict definition of behavioral modeling, this chapter presents a low-level form of behavioral modeling. Because this modeling is on the gate-level, we create the term "Direct Modeling" so as not confuse it with the raw power of behavioral modeling[7], which we cover in a later chapter. The Verilog language supports many constructs that don't relate to gate-level modeling; this text does not cover many of those constructs.

4.2 Basic Verilog Model Structure

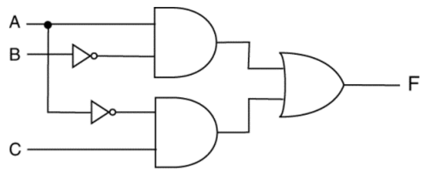
Verilog models follow a basic syntactic structure. We divide this basic model into three distinct sections, which we describe below. Verilog uses the notion of *modules* to define digital circuits, or what we like to refer to as "boxes". No worries, it will make more sense when we see the structure in actual examples.

> ***External Interface:*** The external interface comprises of the signals the outside world knows about. There are three types of signals for the external interface: **`input`**, **`output`**, and **`inout`**, where each of these types is a Verilog reserved word. The inout signal is a bi-directional signal, which we don't cover in this text; we only deal with signal of type **`input`** or **`output`**.
>
> ***Internal Interface***: Verilog uses internal signals as nodes, which means they are connections between the various modules (boxes) in a design. There are many types of internal signals in Verilog, this text uses only two: **`wire`** and **`reg`**. This chapter uses the **wire**-type of internal signals; later chapters cover the **reg**-type.
>
> ***Internal Circuitry:*** the internal circuitry of a module includes a bunch of boxes (modules) that connect together in such a manner as to create a meaningful digital circuit. Once again, there are many different types of internal circuitry in Verilog modules, this text only discusses a modest subset of the possibilities. Specifically, this text models circuits using with modules defined with direct modeling, behavioral model, and structural modeling. This chapter describes direct modeling.

[7] One good thing to say about VHDL is that it does a better job of separating between types of models. If you know VHDL, the Direct Modeling we're defining here is basically a dataflow model.

Example 4.1: Basic Circuit Modeling #1

Provide a Verilog model that you could use to synthesize the following circuit. Use only one statement in your model.



Solution: Figure 4.1 shows the solution for this example. This is our first example of a Verilog model, so we have many things to say about it:

- We use a bold-face font for the non-operator Verilog reserved words. This is for convenience; your text editor may not support such a feature.
- The model contains a few nicely placed comments. The identifiers are rather wimpy, but that's not a big deal for simple examples such as this one.
- When initially learning a new language, there is always an issue of when and where to place the semicolons. We don't describe the rules here; you learn them fast once you start generating actual Verilog models.
- The first part of any "box" definition in Verilog is the external interface. The vernacular Verilog uses to refer to that box is a **`module`**. There are several approaches to defining the Verilog module; the approach we present for this solution is the most simple and instructive.
- The circuit in the problem has three inputs and one output; we use the input and output keywords to describe them in the module definition[8]. The input and output specifications represent the interface to this circuit as the outside world sees it.
- The model uses one *continuous assignment* statement, which reflects one of the main tenets of Verilog. Recall that we're modeling a digital circuit. The notion of continuous assignment provides a means to model a box where the outputs are constantly updated when an input changes. What this means for this circuit is that *anytime a signal on the right side of the equations changes, the equation is re-evaluated*. This enables the output to change anytime there is a change in the inputs.
- The continuous assignment statement models the entirety of the circuit's internal circuitry.
- The continuous assignment statement uses several operators in the equation. The "&" operator represents a bitwise AND operation, the "|" operator represents a bitwise OR, and the "~" operator represent a bitwise complement operation[9]. A more complete listing of Verilog operators follows this example.
- The continuous assignment statement uses parenthesis. Verilog operators do have an associated precedence, but I only remember one: the "~" operator has the highest precedence of all the operators (or at least high enough that I don't need think about it in these equations). The equation includes parenthesis around the AND operators, just to be

[8] Ideally, we only want one identifier per **`input`** line, but we put A, B, & C on the same line to save space. Your code should use only one identifier put **`input`** or **`output`** line. If you do place multiple identifier declarations per line, the should have a similar purpose in the given circuit.

[9] This equation represents the first evidence that Verilog has many similarities to the C programming language.

sure. Yep, I probably should know the precedence rules[10]. But then again, using parenthesis makes equations such as this one easier for humans to read.

```
module example_01(A,B,C,F);
   // external interface
   input A, B, C;
   output F;

   // internal circuitry
   assign F = (A & ~B) | (~A & C);

endmodule
```

Figure 4.1: Solution for Example 4.1

4.3 Verilog Details

Now that you've seen an actual Verilog model that implements a simple circuit, it's time to toss in some more details. Verilog provides a rich set of operators; you saw three of them in the previous example. This section provides full disclosure of the bitwise logic operators you saw in the previous example.

4.3.1 Verilog Bitwise Operators

Verilog uses bitwise operators to implement basic logic operations. Table 4.1shows Verilog's bitwise logic operators with examples. The previous example used three of these operators on 1-bit wide signals.

Operator	Type	Example	Description
`~`	unary	`~X`	Invert all bits in `X`
`&`	binary	`X & Y`	AND each bit of `X` with each bit of `Y`
`|`	binary	`X | Y`	OR each bit of `X` with each bit of `Y`
`~^ or ^~`	binary	`X ~^ Y`	exclusive OR each bit of `X` with each bit of `Y`

Table 4.1: Bitwise logic operators in Verilog.

4.3.2 Verilog Nets and Variables

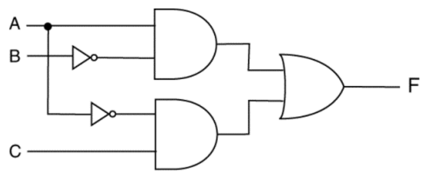
Verilog uses the notion of *nets* and *variables* to represent signals digital circuit descriptions. Although Verilog uses many different types of nets and variables, this text only discusses one type of net and one type of variable. This chapter only discusses the one type of net, the **`wire`**. A later chapter introduces the **`reg`** type of variable.

Digital designs are inherently hierarchical, which means they comprise of different modules exchanging control and data information. The Verilog **`module`** definition uses input and output specifications to define the module interface (signals in the design that the outside world knows about), but we need a way to define internal signals that are not part of the interface (signals the outside world does not know about). One way to do this is using a type of net Verilog refers to as a **`wire`**.

[10] When I program in C, I always have a copy of the precedence rules handy. If you're careful, you generally don't run into operator precedence issues in HDL modeling as often as you do in higher-level language programming. The general rule is still: ***if in doubt, use parenthesis***.

Example 4.2: Basic Circuit Modeling #2

Provide a Verilog model that you could use to synthesize the following circuit. Use one continuous assignment statement for each gate in the provided circuit. Assume an inverter is not a gate.



Solution: This solution is not necessarily the best way to model the circuit, but we present it to show the first use of an explicit **wire** declaration in a module. It's the same circuit as the previous problem, but we need to use a separate continuous assignment statement for each gate. From examining the circuit, you can see that the circuit has four internal signals that the outside world does not see. Only the two output signals of the AND gates are of interest to us. What we need to do is decompose the continuous assignment statement in the last examples to one assignment per gates in this example. When we do it this way, we need to utilize internal interface signal, which we do in the solution below. Here is the full smear:

- We use the net data type **`wire`** to declare the internal interface signal. We placed both net declarations on the same line; it is better to use two different lines for these declarations.
- The internal interface signals are not part of the module's interface (inputs & outputs).

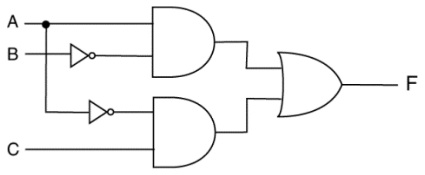
Although the amount of code in this example is greater than the previous example, it synthesizes the exact same hardware.

```
module example_02(A,B,C,F);
   // external interface
   input  A, B, C;
   output F;

   // internal interface
   wire X, Y;

   // internal circuitry
   assign X = (A & ~B);
   assign Y = (~A & C);
   assign F = X | Y;

endmodule
```

Figure 4.2: Solution for Example 4.2

Example 4.3: Basic Circuit Modeling #3

Provide a Verilog model that you could use to synthesize the following circuit. Use one continuous assignment statement for each gate in the provided circuit. Don't use a **`wire`** net in your design. Assume an inverter is not a gate.



Solution: The same circuit, yet again. The point behind this circuit is the notion that you can't use a **`wire`** declaration in your design. Figure 4.3 shows the solution to this example. Notice that the model is exactly the same as the solution to Example 4.2 except that it does not include the **wire** declaration. Despite this fact, the model properly synthesizes. The point of this example is that in many instances, you don't need to explicitly include the **wire** declaration. When the model does not include a **wire** declaration but does contain internal signals, the synthesizer automatically declares the required signals as **wire**s. The proper terminology in this problem is that the model implicitly declares **X** & **Y** as **wire**s.

The question you must ask yourself is whether this is a good coding practice or not. As you learn Verilog, it is the best idea to use explicit **wire** declarations, as it promotes a healthy understanding of the Verilog basics. Use implicit declarations once you are closer to master the language, or if you're trying confuse people.

```
module example_03(A,B,C,F);
   // external interface
   input  A, B, C;
   output F;

   // internal circuitry
   assign X = (A & ~B);
   assign Y = (~A & C);
   assign F = X | Y;

endmodule
```

Figure 4.3: Solution for Example 4.3 highlighting implicit wire declarations.

4.4 Scalar and Vector Data Types

The previous examples were relatively simple circuits; all the input and output signals of the circuits were one bit wide. Most digital circuits are not that simple and typically need to deal with bundled signals (busses). Verilog supports bundled signals with the notion of *vectors*; we refer to signals one bit wide as *scalars*.

Vectored signal declarations are identical to scalar declarations except that vector definitions must include the index range as part of the declaration. The *bit-select* operator is a single value in brackets, which allows access to individual bits in a vector. We use colon-separated numbers in brackets when we need access to multiple bits in the vector. The Verilog code fragment in Figure 4.4 provides a few generously commented examples of vector declarations and bit access to vectors.

```
input  A;         //- scalar definition for input

input [7:0] B;    //- vector definition for 8-bit input signal
                  //- B represents the entire vector
                  //- B[7] = MSB
                  //- B[0] = LSB

input [6:2] C;    //- vector definition for 5-bit input signal
                  //- C represents the entire vector
                  //- C[6] = MSB
                  //- C[2] = LSB

output [0:3] F;   //- vector definition for 4-bit output signal
                  //- F represents the entire vector
                  //- F[0] = MSB
                  //- F[3] = LSB
```

Figure 4.4: Example of vector declarations and access to bits within a vector.

Example 4.4: Basic Circuit Modeling #4

Provide a Verilog model of a circuit that outputs status of a 4-bit unsigned binary input. The circuit has three status outputs that indicate the following about the input: 1) when it is greater than or equal to 12, 2) when it is less than 8, and 3) when it is greater than 3.

Solution: This example does not provide a black box diagram (BBD), so our first step is to generate one based on the problem's description. This is a good first step because it helps us visualize the circuit's external interface. Figure 4.5 shows the BBD for this circuit.

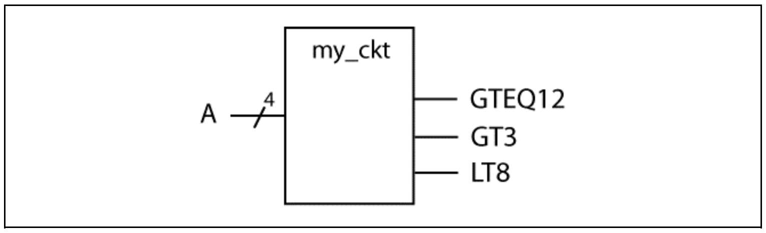


Figure 4.5: The black box diagram for circuit in this example.

Our next task is to generate the Verilog code that properly models the circuit solution. Figure 4.6 shows the solution to this example. Here are the fun and interesting things to note.

- The model name matches the name in the BBD, which is good practice.
- The **A** signal is a 4-bit vector, as we indicate in the external interface declaration with the use of the bracket notation.
- The model has three continuous assignment statements, which use an AND, OR, and inversion operators. The logic does in fact solve the problem, but what we're interested in here is how the model accesses the individual signals in the vector. This example only requires access to single bits. We can use the colon separator in cases where we need more bits, but the bits needs to be contiguous (such as **X**[3:2] = **Y**[3:2]).

```
//- definition of 3-bit comparators
module my_ckt(A, GTEQ12, GT3, LT8);
   //- external interface signals
   input [3:0] A;
   output GTEQ12, GT3, LT8;

   assign GTEQ12 = A[3] & A[2];

   assign GT3 = A[3] | A[2];

   assign LT8 = ~A[3];

endmodule
```

Figure 4.6: The solution for Example 4.4.

4.5 Chapter Summary

- We coined the term *Direct Modeling* to differentiate between official Verilog "behavioral modeling" done at a low level and on the true behavioral level (at a high level). This is a term we use in this text; you won't see it in other texts involving Verilog. The true power of HDL modeling lies with behavioral models, where circuits are modeled by describing their behavior rather than the low-level logic that implements the circuit.
- Verilog has many operators including bitwise logic operators.
- Verilog has many data types; with one of them being a net. The main type of net we use in the initial learning of digital design is the **wire-**type. **wire**s represent nodes in circuits; we use them to connect internal components of circuits, which we refer to as supporting the internal interface of the circuit.
- Verilog can use either scalar data types of vector data types. Verilog uses bracket notion when declaring vectors and accessing individual signals within vectors.

5 Structural Modeling

5.1 Introduction

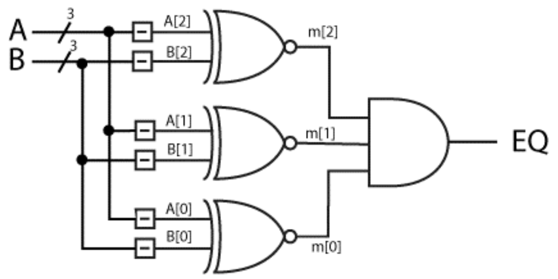
The two main tenets of modern digital design are that designs are both modular and hierarchical. While you could design every circuit on the gate-level, higher-level designs, where we abstract the design process to higher levels, are significantly more efficient. HDLs such as Verilog support modularity and hierarchy with the notion of *structural modeling*. The term structural modeling is a mechanism that supports the notion of module-based design, which includes placing boxes within boxes to create new boxes.

5.2 Structural Modeling Syntax

If your design is not some type of structural model, chances are that you will later use that design as a component in one of your later structural models. Structural modeling is one of the items that provides great modeling power to HDLs such as Verilog. The good news is that there is no new modeling concepts attached to structural models in terms of the languages ability to model circuits. The bad news is that you must learn a bunch of new syntax to create a structural model. Once you create a few models, the new syntax becomes second nature. Because structural models are so powerful, HDLs such as Verilog use them often, so you'll be an expert in no time.

Example 5.1: Structural Model of 3-Bit Comparator

Model the following circuit using structural modeling. This circuit is a 3-bit comparator, which you don't have to know to complete the problem.



Solution: First, when you look at this problem, you only need to see the gates; there is some extra information in there that's going to make the problem easier to model. We'll explain on the details later.

The circuit is a 3-bit comparator, which we typically model using bundle notion on the two 3-bit inputs. Although our main goal for this example is to use structural modeling to describe the comparator, we use vector notion to represent the input signals. Because we use the vector notation, we need a way to show the changing from the 3-bit bundle to the individual signals input to the XNOR gates. The small squares enclosing a "-" is the notation we use to indicate that we reduce the width of the signal before continuing in the diagram. For this diagram, this notion indicates the signal went from three bits to one bit. We then name the signal to support our VHD model, but we generally do not include that level of detail in a BBD. The signal naming notation in the example purposely uses Verilog syntax.

Figure 5.1 shows the final solution to Example 5.1. Here are many more items of interest regarding the solution in.

- We did not write the code in an optimal manner; we wrote it so save space in hopes that it would fit on a single page of text. Well-written Verilog code only has one identifier definition per line.
- The model first defines both an XNOR gate and a 3-input AND gate. Although these are simple gates and would be easy to define in one line of code, we define them as separate modules because we want to use them as modules in the higher-level model.
- The top-level module (comp_3b) uses Verilog bundle notation to declare the **A** & **B** inputs. We are modeling a 3-bit comparator, so both A & B are 3-bits wide; the code indicates this using "[2:0]" after the input modifier. Note that [2:0] includes three signals: 2, 1, & 0.
- The circuit has internal interfaces for the output of the XNOR gates (or inputs of the AND gate). We declare these three signals as a bundle. We could have declared them as three individual **wire**s, but using bundle notation is clearer.
- This model uses two external modules: the XNOR gates and the AND gates. We have declared and defined these modules elsewhere. We now need to *instantiate* three XNOR gates and one AND gates. In other words, our circuits requires three instances of XNOR gates and one instance of an AND gate.
- There are many approaches to instantiating modules; this circuit uses the clearest approach. This approach is clearer because it shows the direct relation between the external signals of the module we're instantiating and how those signals map to signals in the current level of the circuit model. Another way to look at this is the signal names preceded by the period are on a lower level than the signals in the parenthesis, which they are mapping to on the current level of the model. Thus, the approach in this example means the signals are explicitly mapped; the other approaches are implicitly mapped with creates a model that is horrifically hard to debug[1].
- There is never a name clash between the signal names in the module being instantiated and the signal names in the model that instantiating them. A common error for newbie digital designers is to make these signal names different. Don't ever do that, as it makes you models harder to understand, and calls out that you're a beginner.
- We align just about everything in the file to make it more readable. This is particularly true of the instantiations, where we align all the instantiations, as well as everything within the instantiations. This file is properly indented.
- Proper use of white space other than indentation makes the file more readable, including an intelligent use of blank lines.
- The commented code makes the model more understandable.

[1] And when you're first working with Verilog and its somewhat strange approach to instantiating modules, you're going to make mistakes. Use explicit mapping to make your models easier to debug.

```
module my_xnor(A, B, F);    //- defintion of XNOR gate
   input  A,B;
   output F;

   assign F = ~(A ^ B);
endmodule

module my_and(A, B, C, F);  //- definition of 3-input AND gate
   input  A, B, C;
   output F;

   assign F = A & B & C;
endmodule

//- definition of 3-bit comparators
module comp_3b(A, B, EQ);
    //- external interface signals
    input [2:0] A ,B;
    output EQ;

    //- internal interface signals
    wire [2:0] m;

   //- internal circuitry --------------
   //- XNOR instantiation
   my_xnor XNOR2 (
      .A (A[2]),
      .B (B[2]),
      .F (m[2])   );

   //- XNOR instantiation
   my_xnor XNOR1 (
      .A (A[1]),
      .B (B[1]),
      .F (m[1])   );

   //- XNOR instantiation
   my_xnor XNOR0 (
      .A (A[0]),
      .B (B[0]),
      .F (m[0])   );

   //- AND instantiation
   my_and  AND0 (
      .A (m[2]),
      .B (m[1]),
      .C (m[0]),
      .F (EQ)   );

endmodule
```

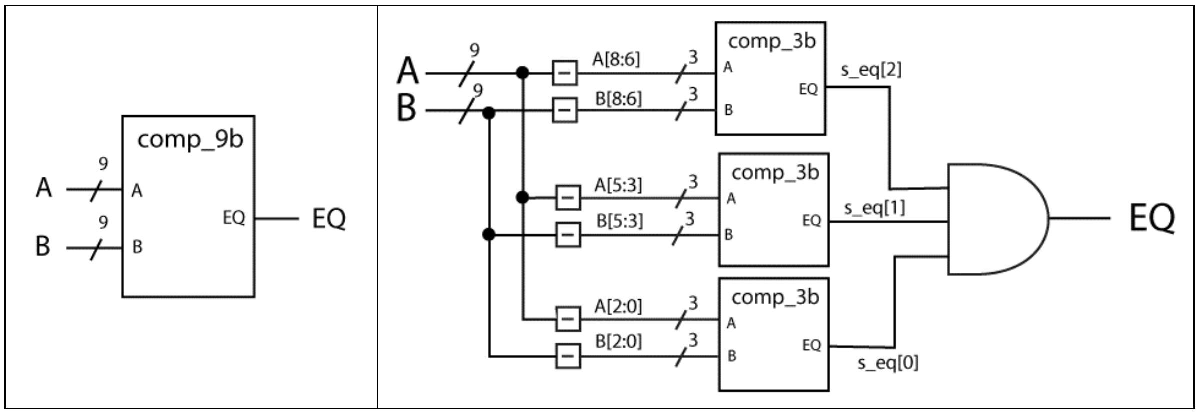
Figure 5.1: Solution to Example 5.1.

Example 5.2: Component-Based 9-Bit Comparator

Use structural modeling and three 3-bit comparators to model a 9-bit comparator. Use the 3-bit comparator from Example 5.1, so provide only the Verilog model of the highest level.

Solution: This is another comparator problem that we model using structural modeling. The problem did not provide any BBD, so our first step in this solution is to generate the BBD in Figure 5.2(a). A 9-bit comparator compares two 9-bit numbers; the output indicates whether they are equal or not. As you'll soon see, modeling a comparator of any bit-width is trivial when you use the correct technique; this problem provides practice in structural modeling, but is far from the optimal comparator module.

The problem states that we need to make a 9-bit comparator from three 3-bit comparators. The previous problem modeled a 3-bit comparator; this problem will thus reuse that module with three instantiations in the problem. Each of the 3-bit comparators has one EQ output; the 9-bit values are equal when the EQ output of each 3-bit comparator instance is asserted. This is a verbal description of the circuit in Figure 5.2(b). Note that the diagram in Figure 5.2(b) once again uses the bundle-width reduction square thingy; this time the signals reduce from nine to three bits.



(a) (b)

Figure 5.2: Top two levels for component-based 9-bit comparator.

Figure 5.3 shows the final solution to Example 5.2. Be sure to note that this solution shares many similarities to the previous example. Here are a few more items of extreme interest for the solution:

- Each of the 3-bit comparator instances inputs 3-bits from the 9-bit input bundle for the 9-bit comparator. The notation we use to indicate this in Figure 5.2(b) is purposely Verilog syntax; you can see the same syntax for each comp_3b instance.
- We arbitrarily use a continuous assignment statement in the model; we could have used an instance of our previous 3-input AND gate.
- The model also contains a second continuous assignment statement that is commented out. We include this because it provides an example of a reduction operator, which can be useful in some cases. Section 5.3 provides a full explanation of Verilog's operators.

```
//- Structural based 9-bit Comparator
module comp_9b(A, B, EQ);
   input   [8:0] A, B;
   output  EQ;

   wire [2:0] s_eq;

   //- 3-bit comparator instantiation
   comp_3b comp_02 (
      .A (A[8:6]),
      .B (B[8:6]),
      .EQ (s_eq[2]) );

   //- 3-bit comparator instantiation
   comp_3b comp_01 (
      .A (A[5:3]),
      .B (B[5:3]),
      .EQ (s_eq[1]) );

   //- 3-bit comparator instantiation
   comp_3b comp_00 (
      .A (A[2:0]),
      .B (B[2:0]),
      .EQ (s_eq[0]) );

   assign EQ = s_eq[2] &  s_eq[1] &  s_eq[0];

   //- The following line is equivalent to the previous
   //- line using a unary reduction operator
   //- assign EQ = &s_eq;

endmodule
```

Figure 5.3: Solution to Example 5.2.

5.3 Unary Reduction Operators

Verilog has a set of operators that you may never use, but you need to know about them in case you need to use them. The unary reduction operators provide a shortcut notation for certain circuit situations. The two situations where this come ups is with implementing parity generators and supporting generic Verilog models. We discuss generic models in a later chapter.

Table 5.1 shows a list of Verilog's the unary reduction operators. Figure 5.4 shows a few examples of these operators using a Verilog code fragment.

Operator	Example	Description
`&`	`&X`	AND all bits in **X** together (1-bit result)
`~&`	`~&X`	NAND all bits in **X** together (1-bit result)
`|`	`|X`	OR all bits in **X** together (1-bit result)
`~|`	`~|X`	NOR all bits in **X** together (1-bit result)
`^`	`^X`	XOR all bits in **X** together (1-bit result)
`~^ or ^~`	`~^X`	XNOR all bits in **X** together (1-bit result)

Table 5.1: Unary reduction operators in Verilog.

```
    input [7:0] A;
    output F;

    assign F = &A;    //- F = '1' when all bits in A are set;
                      //- otherwise, F = '0';

    assign F = ^A;    //- F = '1' when bits in A have odd parity;
                      //- otherwise, F = '0' (even parity);

    assign F = ~|A;   //- F = '1' when all bits in A are cleared;
                      //- otherwise, F = '0';
```

Figure 5.4: Verilog code fragment showing examples of unary operators.

5.4 The Final Word

Using a structural design is a choice the digital designer makes; and it certainly is a good choice. Structural design exists for one reason: to help humans model digital circuits. If your design does not instantiate any modules, then we refer to that design as a *flat design*. Flat designs are fine for simple circuits, but digital design is boring if all you find yourself doing is modeling simple circuits. Structural design supports modularity and abstraction of designs, both factors in helping humans understand your design and possibly reuse some of the modules in your design.

Part of the design process is to layout your design on the module level. The division of tasks between the modules should make sense; you should never attempt to make any one module overly complicated, as such modules are hard to verify (because you give the synthesizer too much freedom).

The final word is that the synthesizer does not care how you model your design, as it flattens your design as part of the synthesis process. A properly executed and formatted structural model will never generate more hardware than a functionally equivalent model using a flat design.

5.5 Chapter Summary

- Structural modeling is the HDL approach to supporting the two main tenets of modern digital design: modularity and hierarchical design.
- Structural designs generally do not increase the amount of hardware generated by synthesis; their primary purpose is to support the understanding of the circuits by humans.
- Structural designs enhance the readability and understandability of digital models. Models with these attributes are easier to modify and debug.

6 Behavioral Modeling

6.1 Introduction

The circuits we've modeled up to this point have been very relatively simple combinatorial circuits. The two approaches we used were direct modeling and structural modeling. As we move towards generating more complex circuits, we need to employ a more powerful modeling technique, which we refer to as behavioral modeling.

The main idea behind behavioral modeling is that we can model circuits by describing how the circuits "behaves" rather than describing the gate-level operation of the circuit. Describing a circuit by its behavior is a higher-level approach than describing the underlying logic that implements the circuit's functionality. In truth, the continuous assignment statement is a form of behavioral modeling; we choose to label it as direct modeling because of its ability to describe circuits at the gate level.

6.2 The Path of Behavioral Modeling

Our introduction to behavioral modeling follows our approach to learning Verilog in general: we limit the amount of information we provide and we provide that information in the context of actual design examples. Additionally, there are many concepts that we must introduce to provide a solid foundation of Verilog skills, so presenting this information in the correct order is imperative.

One of the main issues with using behavioral models the notion behavioral modeling adds more "knobs" to the design process. Recall that the more knobs a design has, the more likely the synthesizer is going to use one of those knobs in a way you did not expect, and subsequently generate a design that does not work. The fact is that behavioral designs have many knobs; the approach this text uses is to place constraints on how you can use those knobs. As you gather more Verilog skills and become more familiar with the synthesis and verification process, you can model circuits any way you deem appropriate.

Our approach is to first dedicate a few chapters to the behavioral modeling of combinatorial circuits. We then switch to the modeling of sequential circuits, which include finite state machines (FSMs). Additionally, all the sequential circuits we consider in this text are synchronous, where most of the module's actions are synchronized to an active clock edge in the device.

6.3 Behavioral Modeling with Procedural Blocks

The heart of Verilog modeling is the notion of procedural blocks. Verilog uses two types of procedural blocks: **`always`** blocks and **`initial`** blocks. The notion of an **`initial`** block is somewhat special so we save that for the chapter on model verification using testbenches. The majority of this text uses **`always`** blocks.

Now's a good time to remind you that your mission is to model digital circuits. A digital circuit is a smattering of digital modules all working in parallel. You must design this massively parallel circuit using text in a file, which is something you must read in a sequential manner. As you read through these verbage and examples that follow, don't allow the vernacular and syntax to make you to think that what you're doing has a relation to software. If creating digital models feels like writing software, you're probably doing something wrong (and your circuit probably will not work).

6.3.1 The `always` Block

The **always** block is officially an infinite loop which processes statements within the loop repeatedly. The body of the always block contains statements that describe the operation of the module. But then again, because the **always** block is modeling digital hardware, the **always** block is essentially one way to generate a "box" (or module).

The synthesizer interprets the statements within an **always** block in a sequential manner, yet the **always** block is a type of concurrent statement. This sounds strange, but I feel it is analogous to how a contractor would look at a set of plans to build a house. Houses have floors, roofs, ceilings, and other items; the associated set of blueprints lists everything the contractor needs to know in order to build a house. There is some semblance of order associated with building a house though: generally speaking, you don't start out building the roof, then the walls, then finally the foundation. I suppose you could, but if you do things in the correct order, you end up with a house that was built in an efficient manner. When you build a house, you start out with the foundation and work upwards.

Once again, synthesizers must read special code in a sequential[1] manner and magically creates the circuit you're hoping for. The moment we stop designing circuits using meaningful hardware such as gates, we have to start working more closely with the quirks of the synthesizer. The synthesizer interprets the code that goes inside an **always** block in a sequential manner, thus the order matters. The order that the sequential blocks appear in a Verilog module does not matter, as the synthesizer interprets all **always** blocks as modules that operate concurrently with other blocks in the circuit.

Modern digital design is about plopping down blocks and connecting them in such a way as to solve the given problem. It should thus be no surprise that HDLs such as Verilog have significant support for creating and plopping down boxes. The **always** block represents the third option for modeling blocks within a digital design. Here are all the options up to this point:

1) Modules (and their use in structural models)

2) Continuous Assignment statements

3) **`always`** blocks

6.3.2 The Guts of the Always Block

The interior of the **always** block contains the code we use to describe the operation of the circuit. There are two main types of statements that we can place inside **always** blocks, which we refer to as procedural programming statements and procedural assignment statements. The synthesizer interprets these states in the order they appear in the **always** block. A slightly more complete description of these items appear in the following sections, but you need to see them in the context of a circuit in order to get a good feel for them.

6.3.2.1 Procedural Programming Statements

Verilog has several types of procedural programming[2] statements; this text only discusses two of them. The two we discuss in this text gives you a clear understanding of how the synthesizer interprets procedural programming statements; as you gain more experience using Verilog, the other types of statements will be easier to understand and work with. The two types of statements we use are the **`if-else`** and **`case`** procedural programming statements. Similar to higher-level computer programming languages, the **case** statement is a special case of the **if-else** statement.

The synthesizer interprets the **if-else** and **case** statements similar to the compiler in higher-level languages. In particular, the synthesizer interprets the statements in order until a clause in the statement evaluates as true. Once one clause evaluates as true, the synthesizer makes the associated assignment and then does not evaluate any other clauses in the procedural programming statement. We mention this now to highlight the similarities; we wait until the next chapter to use these statements in actual models, because the true power of procedural programming statements make more sense in the context of actual circuit models. We wait to present the details until later so we can present them in the context of the relational operators associated with the **`if-else`** and **`case`** procedural programming statements

6.3.2.2 Procedural Assignment Statements

[1] This context of sequential is separate from the notion of a sequential circuit, one of two types of digital circuits.

[2] This is only a name… it's not really programming.

Verilog uses two main types of procedural assignment statements, which we refer to as *blocking procedural assignment* and *non-blocking procedural assignment*. These statements types represent two significant "knobs" that we've been referring to. The manner in which the synthesizer interprets these statements is significantly different. ***If you use these statements without knowing how the synthesizer interprets them, your circuit has no hope of working correctly***. Here is brief description of these assignments for you to compare and contrast. Once again, you need to see these statements in the context of an actual model in order to obtain an intuitive feel for them.

Blocking Procedural Statements: Figure 6.1 shows an example of two blocking procedural statements. The expression associated with a blocking statement is evaluated and assigned when the statement is encountered. This means that the synthesizer makes the assignment and the result is ready to use before the next line in the **always** block. The official vernacular is that that evaluation of other expressions is not done, or *blocked*, until the evaluation of the current blocking statement is complete. Once again, the first assignment complete before and the results of the evaluation are available before the evaluation of the next statement. This has significant ramifications for the synthesized hardware[3].

```
my_variable_1 = my_expression_1;
my_variable_2 = my_expression_2;
```

Figure 6.1: An example of blocking procedural statements.

Non-Blocking Procedural Statements: Figure 6.2 shows an example of two non-blocking procedural assignment statements. The assignment associated with the first procedural statement does not happen until the **always** statement that it is in terminates. This means that evaluation of statements in the **always** block continues (not *blocked*) onto the statement without actually changing the anything; thus the assignments are not-blocked, as they are in blocking procedural statements. Another way to look at this is that assignments are ***scheduled*** to occur, but don't actually occur until the evaluation of the statements in the **always** block reaches the end of the block[4].

```
my_variable_3 <= my_expression_3;
my_variable_4 <= my_expression_4;
```

Figure 6.2: An example of non-blocking procedural statements.

6.3.3 Combinatorial vs. Sequential

There are two main classifications of digital circuits: combinatorial and sequential. Verilog supports the modeling of both types of circuits, but the modeling approach is significantly different. The manner in which the synthesizer handles these circuit types is not complicated, but not overly intuitive either. Once again, the synthesizer follows a set of rules when generating circuits.

In this text, ***we only model sequential circuits using the always procedural block***. More specifically, we use the **`if-else`** and **`case`** procedural programming statements in a special manner to generate either sequential or combinatorial circuits. Here are the two important guidelines regarding the generation of combinatorial and sequential circuits.

[3] The notion here is that the synthesizer does its best to synthesize hardware that has the same blocking-type characteristic. This means that the hardware must generate that result; the problem is that nothing happens in hardware until that result is generated. If you know what you're doing, you can use this to your advantage. But in reality, this gives the synthesizer way too much control. This is something you want to avoid until you get a feel for how the synthesizer interprets the model.

[4] This is a working definition; there is actually more specific details involved that we save for another chapter.

Circuit Type	Guidelines
Combinatorial	1) Model using blocking assignments only 2) Completely specify all input conditions (include a catch-all)
Sequential	1) Model using non-blocking assignments only 2) Do not completely specify all input combinations (don’t include a catch-all)

Table 6.1: Guidelines for modeling combinatorial and sequential circuits.

6.4 Chapter Summary

- Behavioral modeling involves describing how circuits operate, or *behave*, rather than modeling the circuits with gate-level descriptions. Behavioral modeling represents a higher level of abstraction than gate-level circuit descriptions.
- Verilog's notion of procedural blocks in behavioral modeling provides more "knobs" to the synthesizer, which increases the chances that the synthesizer produces a circuit that does not work as intended. Designers can control this level of synthesis uncertainty by proper use of Verilog's behavioral modeling constructs.
- The heart of Verilog behavioral models intended for synthesis is the **`always`** block. This block can contain various types of procedural assignment and procedural programming statements.
- The synthesizer evaluates the procedural statements within an **always** block in sequential order; the **always** block itself is a form of a concurrent statement.
- Blocking and non-blocking procedural assignment statements are two types of statements that can appear in **always** blocks. Blocking assignment statements "block" until the associated expression completes evaluation and is assigned. The actual assignment of non-blocking assignment statements is "scheduled" to occur when the procedural block terminates.
- Always model combinatorial circuit in Verilog using blocking assignment; make sure all input condition possibilities are specified.
- Always model sequential circuits in Verilog using non-blocking statements; make sure that these statements do not include some type of catch-all clause.

7 Decoders

7.1 Introduction

We use Digital Design Foundation Modeling to teach the basic principles of digital design. In DDFM, we describe two types of decoders, and provide them with unique definitions. The examples we present in this chapter use these definitions, so we start this chapter with a brief descriptions of how DDFM models decoders.

7.2 Types of Decoders

DDFM models two types of decoders, which we refer to as *generic decoders* and *standard decoders*. Figure 7.1 shows a Venn diagram indicating that standard decoders are a special case of generic decoders. Standard decoders have specific uses in digital design, which is why we gave them their own designation.

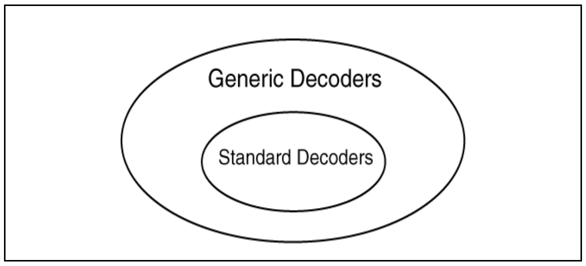


Figure 7.1: Venn diagram showing the hierarchy of decoders.

7.2.1 Generic Decoders

Anytime you can define a digital circuit using a tabular format, then you have effectively defined a generic decoder. A generic decoder is analogous to a *look-up-table* (LUT) in computer programming. If you can describe a circuit in tabular format, you've officially modeled a decoder.

Figure 7.2 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. You can define two general types of tables: 1) complete tables, and, 2) incomplete tables. 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 (meaning some input combinations are not explicitly defined). We make this distinction so you realize that you don't need to completely specify every possible input combination for generic decoders.

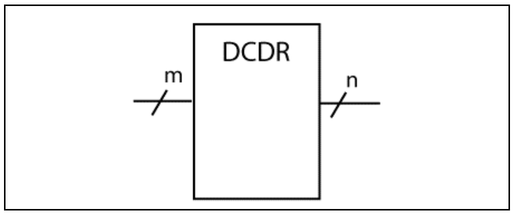


Figure 7.2: A black box diagram of a generic decoder.

7.2.1.1 Generic Decoders Foundation Module

We consider the generic decoder to be one of our Digital Design Foundation circuits and a controlled circuit; Figure 7.3 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 7.1 provides a description of all the inputs and outputs to the generic decoder.

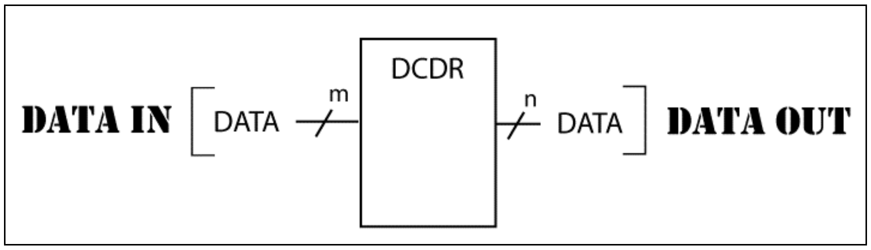


Figure 7.3: Data signals for a generic decoder.

	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 7.1: The foundation matrix for a generic decoder.

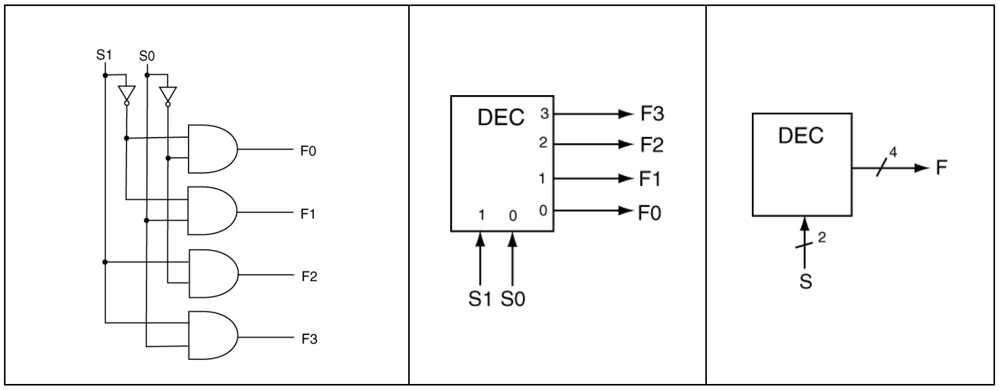
7.2.2 Standard Decoders

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 7.4(a) shows a gate-level model of a 2:4 standard decoder. Due to the configurations of the inputs **S1** and **S0** in Figure 7.4(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., which 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 7.4(b) is adequate to describe a standard decoder, the schematic diagram of Figure 7.4(c) is more common. The small numbers associated the circuit inputs and outputs in Figure 7.4(b) indicate a weighting on those inputs and outputs.
- 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 7.4: A standard 2:4 decoder in schematic and circuit forms.

7.2.2.1 Standard Decoder Foundation Module

We consider the generic decoder to be one of our Digital Design Foundation circuits and a controlled circuit; Figure 7.3 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 **STATUS** signals. By definition, the **STATUS** signals form a one-hot code. Table 7.2 provides a description of all the inputs and outputs to the standard decoder.

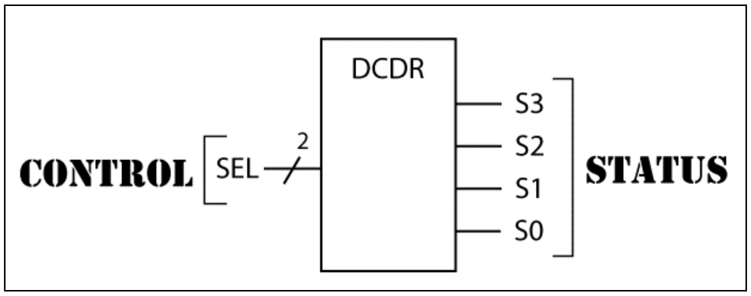


Figure 7.5: Control and status signals for a 2:4 standard decoder.

	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 7.2: The foundation matrix for a standard decoder.

7.3 Verilog Support Issues

This chapter presents the two types of decoders in the context of example problems. The Verilog models for these problems provide an opportunity to present some standard Verilog coding information in the context of actual problems. The representation and manipulation of signals in Verilog is a common part of any non-trivial circuit model.

7.3.1 Concatenating Signals

One of the goals of modeling digital circuits is to make the models as clear as possible. One tool to help in this process is Verilog's concatenation operator. We often concatenate signals in Verilog in order to simplify the modeling process as well as the overall understanding of the resulting code. Concatenating is similar to structural modeling in that it is primarily a tool for humans; liberal use of concatenation is not going to magically make the synthesizer generate a smaller circuit (or larger circuit).

The concatenation operator allows you to create a new signal of larger width from smaller signals. This operator consists of an opening and closing curly brace. The inside of the curly braces includes a comma-separated list of signals that require concatenation; the result of the operation is assigned to another signal. The comma-separated list of signals must be at least two, but you can concatenate as many signals as necessary.

Figure 7.6 shows a code fragment demonstrating the use of the concatenation operator; the comments in the code fragment provide a brief description of the code. Note that the concatenation operator uses the position of the smaller signals in the operator to orientate the new vector, thus the left-most signal in the operator becomes the left-most set of bits in the new signal.

```
input [3:0] A;  //- 4-bit vector
input [4:0] B;  //- 5-bit vector
input [5:0] C;  //- 6-bit vector

wire [6:0]  new_sig_1;
wire [7:0]  new_sig_2;
wire [8:0]  new_sig_3;
wire [11:0] new_sig_4;

new_sig_1 = {A,B};    //-  9-bit signal; A[3:0] are MSBs
new_sig_2 = {A,C};    //- 10-bit signal; A[3:0] are MSBs
new_sig_3 = {C,B};    //- 11-bit signal; C[5:0] are MSBs
new_sig_4 = {B,C,A};  //- 15-bit signal; B[4:0] are MSBs
```

Figure 7.6: A code fragment showing examples of concatenating signals.

7.3.2 Representing Integer Numbers

When modeling digital circuits, you often need to represent numbers. While all numbers in a digital circuit are inherently binary, we make clearer models by representing numbers in other formats. Once again, the synthesizer does not care; different number presentations are for the human digital circuit designers.

Figure 7.7 shows the format for specifying numbers in Verilog. Here is a description of the various fields in that format. Table 7.3 shows a few key examples.

size_in_bits: This is an optional field. You should always include this field in number specifications unless you have a good reason not to, as this value provides information to the entity reading your model.

- If you do not specify a size, the synthesizer represents the value using 32 bits.
- If the significant digits are smaller than the size, the value is zero-extended for unsigned values
- If the significant digits are larger than the size_in_bits, the value is truncated.

radix value: Verilog allows you to assign one of four different radix values: 1) ‘b=binary, 2) ‘o=octal, 3) ‘d = decimal, and 4) ‘h=hexadecimal[16]. If you don’t assign a radix value, the synthesizer interprets the number as a decimal value.

significant digits: The digits representing the numeric value. Values can include underscores to enhance readability; the synthesizer discards the underscore.

[16] There are actually signed versions of these values; this text does not include them.

```
<size_in_bits> ' <radix value> <significant digits>
```

Figure 7.7: A code fragment showing examples of concatenating signals.

Example	Size (bits)	Radix	Binary equivalent
`12`	unsized	10	`0… 0 1 1 0 0 (32 bits)`
`'hD`	unsized	16	`0… 0 1 1 0 1 (32 bits)`
`1'b1`	1	2	`1`
`6'o34`	6	8	`0 1 1 1 0 0`
`12'hFA3`	12	16	`1 1 1 1 1 0 1 0 0 0 1 1`
`5'b10110`	5	2	`1 0 1 1 0`
`5'b1_0110`	5	2	`1 0 1 1 0`
`8'b1111_0011`	8	2	`1 1 1 1 0 0 1 1`

Table 7.3: Example of Verilog number representations.

7.4 The `always` Procedural Block

The **always** procedural block is the primary approach this text uses to create "boxes" (or modules). The **always** block in Verilog has a significant amount of functionality; this text will only use a small portion of that functionality. You can read all about this functionality on various online sources, but the best approach to properly working with these blocks is to generate some actual models. Figure 7.8 shows a general form of the **always** block followed by a description of the various fields in the construct.

sensitivity list: the list of signals that initiate the evaluation of the **always** block

begin: the start of the block

statements: the body of the block, which contains various statements

end: the end of the block.

Also worthy to note is that the **always** statements don't require the begin-end pair if there is only one statement in the body of the block. And like in C programming, this can really mess you up if you don't know what you're doing. It is thus a good practice to *never not use* (trying not to say "always") the begin-end pair in an **always** block.

Lastly, we list a bunch of information regarding **always** blocks in the next few sections. I suggest reading it fast, and then referring back to it as needed. All this stuff makes more sense when you see it in code that is modeling a digital device that you're familiar with. The remainder of this chapter contains meaningful example problems.

```
always @(sensitivity_list)
   begin : block_name
      statements
   end
```

Figure 7.8: The generic form of an `always` block.

7.4.1 The Sensitivity List

The sensitivity list is a term borrowed from VHDL. For beginning digital designers, it is best to consider this a sensitivity list, though it actually a form of time control[17]. There is more to it than we cover in this text.

Following an underlying theme of this text, the sensitivity list provides "knobs" to the synthesizer, so we need to be careful with what we include in the list. We will initially be working with combinatorial circuits, so we deal with the sensitivity list in one manner: ***every signal on the right side of the blocking or non-blocking assignment operators needs to appear in the sensitivity list.*** Not that following rules is great, but this approach constrains the sensitivity list knob, which helps prevent the synthesizer from providing us with surprises.

7.4.2 The Block Body

The body of the **always** block contains two types of statements: *procedural programming statements* or *procedural assignment statements*. In this text, we will use only **if-else** and **case** types of procedural programming statements. Similarly, we will use only blocking or non-blocking types of procedural assignment statements. Additionally, we will always use blocking assignment statement for combinatorial logic and non-blocking statements for sequential logic.

7.4.3 Variable Assignment

Assignment within **always** blocks must be to **variables** types rather than **net** types. The previous models we worked with used continuous assignment statements to the **`wire`** net types. *You can't assign a value to a **wire**-type from a procedural block*; ***all assignments from procedural blocks must be to variables***. Verilog defines several variable types; we only use the **`reg`**-type in this text.

Figure 7.9 show the two flavors of **reg**-type declarations that we use in circuit models. Note that one type supporting the variable assignment when the signal is part of the module's interface Figure 7.9(a); the other type of assignment is when the model uses the signal as an internal signal.

[17] Don't worry about this, but keep it in the back of your mind.

(a)	(b)

`//----------------------------------------`
`//- reg declaration when variable is`
`//-  part of modules interface`
`//----------------------------------------`
`module var_example_1(A,B,C,F);`
`   //- external interface`
`   input A, B, C;`
`   output reg F;`
	`//----------------------------------------`
	`//- reg declaration when variable is an`
	`//- internal signal (not part of module`
	`//- interface)`
	`//----------------------------------------`
	`module var_example_2(A,B,C,F);`
	`   //- external interface`
	`   input A, B, C;`
	`   output F;`
	
	`   reg   my_var;`

Figure 7.9: Code fragments showing reg-type variable useage.

7.4.4 Statement Types

As we described in a previous chapter, there are two types of statements that we use in an **always** block: 1) procedural programming statements, and 2) procedural assignment statements. The procedural programming statements we use in this text are **if-else** statements and **case** statements.

7.4.4.1 Procedural Programming Statements

Verilog has several types of procedural programming statements; this text only covers the **if**, **if-else**, and **case** types. These three types are more than adequate to model a wide range of digital circuits.

The first type of procedural programming statement we examine is the **if** statement. Figure 7.10 shows examples of the two forms of if statements. Here are the things you should take away from these two forms:

- When the parenthetical expression evaluates are true, the statement (or statements) associated with the **if** clause are executed.
- The forms in Figure 7.10(a) and Figure 7.10(b) are essentially equivalent, but differ based on the number of statements associated with the if statement. Specifically, if there is only one statement associated with the **if** statement, as in Figure 7.10(a), we don't need to include the begin-end pair (although it's always a good idea to include them). When there is more than one statement associated with the **if** statement, you must use the **begin-end** pair.
- The comment in the code states that the **if** statement should not be used for combinatorial circuits. We'll address this issue more completely when we describe sequential circuits.

(a)	(b)

`//- not for combinatorial circuits`
`if (expression)`
`   statement`
	`//- not for combinatorial circuits`
	`if (expression)`
	`begin`
	`   statement(s)`
	`end`

Figure 7.10: Two forms of the if statement

The next type of procedural programming statement is the **if-else** statement. This is similar to the **if** statement, but most importantly, the associated else part of the **if-else** statement provides what we refer to as a *catch-all* to the **if-else** statement. This relates directly to modeling a sequential vs. combinatorial circuit. We cover this in the chapter on sequential circuits. What we boldly state here is that *your **always** statements always need some sort of catch-all to ensure the synthesizer generates a combinatorial circuit*.

```
if (expression)
   statement
else   //- catch-all
   statement
```

(a)

```
if (expression)
begin
   statement(s)
end
else   //- catch-all
begin
   statement(s)
end
```

(b)

Figure 7.11: Two forms of the if-else statement

Figure 7.12 show the generic form of a case statement. Here are a few interesting items regarding the **case** statement:

- The **case** statement compares the value of the evaluated expression with each case_item. If a case_item matches the evaluation, the associated statement executes.
- If no case_item matches the evaluation of the expression, the statement associated with the default statement executes. The default statement is thus a catchall, which guarantees that at least one statement in the **case** is executed.

```
case (expression)
   case_item : statement;
   case_item : statement;
   default : statement;  //- catchall
endcase
```

Figure 7.12: The general form of the case statment.

7.4.4.2 Procedural Assignment Statements

This text uses two types of procedural assignment statements, which are the blocking and non-blocking statements.

```
my_variable_1 = my_expression_1;    //- blocking assignment

my_variable_2 <= my_expression_2;   //- non-blocking assignment
```

Figure 7.13: An example of blocking procedural statements.

7.4.5 Equality Operators

The expressions associated with if clauses are often associated with equality relationships between two signals. The evaluation of the associated expression returns a single bit, where ‘1’ represents true and ‘0’ represents false. In other words, true means the statements associated with the particular if statements are executed; false means the statements are not executed. Table 7.4 shows Verilog’s two equality operators.

Operator	Example	Description
`==`	`(X == Y)`	Is X equal to Y? (true or false; 1-bit result)
`!=`	`(X != Y)`	Is X not equal to Y? (true or false; 1-bit result)

Table 7.4: Verilog's equality operators.

Example 7.1: Decoder Example #1

Write Verilog model that describes the function on the right.

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

Solution: This is a three-input, one output circuit. To make the problem more interesting, we assume the three inputs are scalar signals. Because the problem provides the function in a tabular format, we know we can implement this function using a generic decoder. Figure 7.14 shows the solution to this problem. The code comments and the following description highlight the important issues as this is our first problem using a procedural block.

- Everything in the model is nicely aligned; the code looks great standing a few feet back. Another reason the file looks good is because of proper use of white space.
- This is a generic decoder, so we use an **always** block rather than attempting to implement the function using continuous assignment statements (gate-level logic).
- Because we are using a procedural block, all assignments must be to **`reg`**-types. The **always** block assigns a value to **F** from the **always** block, so we declare the output port F as a **`reg`**.
- Working with bundled signals is generally the best option, so we concatenate the three input signals together to create one 3-bit bundled signal. If we did not do this, the model would be significantly less readable. The approach to doing this is to declare a bundled signal, which can be a **wire**-type, then use a continuous assignment statement and the concatenation operator to create the bundled signal.
- The **always** block's sensitivity list includes all inputs accessed in the block; in this case, it is only the ABC bundle. We do this because the decoder is a combinatorial device.
- The **always** block uses blocking-type assignment statements because a decoder is a combinatorial device.
- The **if** statements use equality operators ("==") to test input conditions.
- The **if-else** clause contains an **else**, which acts as a catch-all. We necessarily include a catch-all for every combinatorial device we model.
- The **if-else** statement essentially looks for the three cases when the input values generate a '1' on the output; if no **if** statement condition evaluates as true, the **if-else** statement executes the assignment associated with **else** (the catchall statement).

- The **case** statement includes a default clause, which is the catchall we seek.
- The code is nicely aligned, which supports readability by humans.

```
module dcdr_generic_ex1(A, B, C, F);
    input  A, B, C;
    output reg F;

    //- declare bundle
    wire [2:0] ABC;

    //- create bundled signal with concatenation
    assign ABC = {A,B,C};

    always @(ABC)
    begin
       if      (ABC == 3'b001)  F = 1'b1;
       else if (ABC == 3'b110)  F = 1'b1;
       else if (ABC == 3'b111)  F = 1'b1;
       else                     F = 1'b0;    //- catch-all
    end

endmodule
```

Figure 7.14: Solution to Example 7.1.

There are many different approaches to implementing this model. Verilog syntax allows for many different coding styles; it's your job to pick the ones that generate the clearest model for what you're doing. The model in Figure 7.14 uses an **if-else** statement, but this is probably not the best approach. Figure 7.15 provides an alternate solution using a **case** statement. The general rule in using **if-else** vs. **case** statements is that if you have more than 2-3 conditions to test for using **if-else**, you probably want to switch to a **case** statement. Here are few other things to note about Figure 7.15.

- Note that the **case** statement contains a default clause, which we always need to do when we model combinatorial circuits such as decoders.
- The **case** statement uses blocking assignments, which we use for combinatorial circuit models.

```
module dcdr_generic_ex1b(A, B, C, F);
    input  A, B, C;
    output reg F;

    //- declare bundle
    wire [2:0] ABC;

    //- create bundled signal with concatenation
    assign ABC = {A,B,C};

    always @(ABC)
    begin
       case (ABC)
          3'b001  : F = 1'b1;
          3'b110  : F = 1'b1;
          3'b111  : F = 1'b1;
          default : F = 1'b0;
       endcase
    end

endmodule
```

Figure 7.15: Solution to Example 7.1.

Example 7.2: Decoder Example #2

Provide a Verilog model that implements the functionality described in the following truth table.

A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Solution: The problem asks us to model three functions given in tabular format; this means we use a generic decoder in the solution. Figure 7.16 shows a BBD for the solution.

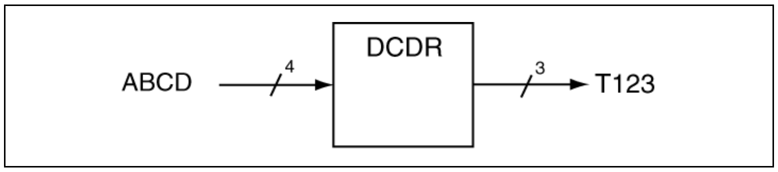


Figure 7.16: Dark box diagram for Example 7.2.

Figure 7.17 shows the solution to this problem using an **if-else** statement while Figure 7.18 shows an alternate solution using a **case** statement. A few things of merit in this solution:

- The code in both models is nicely aligned with liberal use of white space to enhance readability.
- We model the inputs and outputs as bundles, so we did not need to concatenate the individual input and output signals.
- We must declare the output as a **reg**-type because we are assignment the output value from a procedural block (the **always** block).
- For the **if-else** version, we use decimal format in the **if** expressions, and use binary in the assignment. Both of these choices are arbitrary.
- Both versions of this solution contain catch-all statements. The **else** clause is the catch-all for the **if-else** while the **default** clause is the catch-all for the **case** statement. We always provide a catch-all when modeling combinatorial circuits.

```
module dcdr_ex2a(ABCD, T123);
    input  [3:0] ABCD;
    output reg [2:0] T123;

    always @(ABCD)
    begin
       if       (ABCD == 0)  T123 = 3'b110;
       else if (ABCD == 1)  T123 = 3'b000;
       else if (ABCD == 2)  T123 = 3'b100;
       else if (ABCD == 3)  T123 = 3'b010;
       else if (ABCD == 4)  T123 = 3'b000;
       else if (ABCD == 5)  T123 = 3'b101;
       else if (ABCD == 6)  T123 = 3'b000;
       else if (ABCD == 7)  T123 = 3'b101;
       else if (ABCD == 8)  T123 = 3'b010;
       else                 T123 = 3'b000;
    end

endmodule
```

Figure 7.17: Solution to Example 7.1.

```
module dcdr_ex2b(ABCD, T123);
    input  [3:0] ABCD;
    output reg [2:0] T123;

    always @(ABCD)
    begin
       case (ABCD)
          0 : T123 = 3'b110;
          1 : T123 = 3'b000;
          2 : T123 = 3'b100;
          3 : T123 = 3'b010;
          4 : T123 = 3'b000;
          5 : T123 = 3'b101;
          6 : T123 = 3'b000;
          7 : T123 = 3'b101;
          8 : T123 = 3'b010;
          default :  T123 = 3'b000;
       endcase
    end

endmodule
```

Figure 7.18: Solution to Example 7.1.

Example 7.3: Modeling Multiple Functions with a Generic Decoder

Provide a Verilog model that models the following truth table. Not listed in the following table is a CE input; when the CE input is a '0', the outputs of the circuit are all '0's; otherwise, the circuit implements the following table.

A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Solution: This problem is similar to the previous problem, but now we include an enable-type input. When the CE input is asserted, the circuit acts like a normal decoder; when CE is not asserted, the circuit outputs zeros. Figure 7.19 shows the BBD for our solution; Figure 7.20 shows the associated Verilog model.

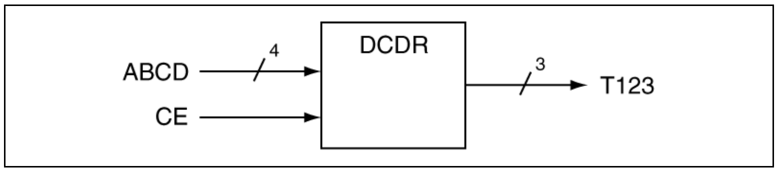


Figure 7.19: BBD for Example 7.3.

There are many different approaches to modeling this circuit; the model in Figure 7.20 is one of those solutions. Here are a few things to note about the Verilog model.

- The solution modifies the model of the previous solution to include a **CE** input. We stole (reused) most of the code from the previous problem.
- The model uses a **case** statement for the decoder; this is arbitrary, but the best solution for a model that has this many input combinations.
- Both the **if** and **else** clauses do not use **begin-end** pairs; we could do this because there is only one statement associated with each the **if** and **else** clause. We did not include begin-end in order to save space; it's arguably a better idea to use **begin-end** pairs in your actual models.

```
module dcdr_ex3(ABCD, T123, CE);
    input  [3:0] ABCD;
    input  CE;
    output reg [2:0] T123;

    always @(ABCD)
    begin
       if (CE == 1)
       begin
          case (ABCD)
             0 : T123 = 3'b110;
             1 : T123 = 3'b000;
             2 : T123 = 3'b100;
             3 : T123 = 3'b010;
             4 : T123 = 3'b000;
             5 : T123 = 3'b101;
             6 : T123 = 3'b000;
             7 : T123 = 3'b101;
             8 : T123 = 3'b010;
             default :  T123 = 3'b000;
          endcase
       end
       else
          T123 = 3'b000;
    end  //- end of always

endmodule
```

Figure 7.20: The final solution for Example 7.3.

Example 7.4: Standard Decoder

Provide a Verilog model for a standard 2:4 decoder.

Solution: The problem has a short description because we all know that a 2:4 decoder has two inputs and four outputs. Figure 7.21 and Figure 7.22 show the **if-else** and **case** versions of the solution, respectively. These models are similar to the models in previous example solutions, so there is little new to point out. But here are a few items of note:

- The code utilizes comments and white space in a meaningful manner.
- We once again specify the numbers using different radii.

Additionally, Figure 7.23 shows a model of a standard 2:4 decoder that uses a non-bundled control signal input. In this case, we use the concatenation operator inside the argument of the **case** statement in order to simply the code and to avoid declaring a separate signal for the concatenated values.

```
module dcdr_standard_1a(sel, data_out);
    input  [1:0] sel;
    output reg [3:0] data_out;

    always @(sel)
    begin
       if (sel == 0)       data_out = 4'b0001;  //- one-hot output
       else if (sel == 1)  data_out = 4'b0010;
       else if (sel == 2)  data_out = 4'b0100;
       else if (sel == 3)  data_out = 4'b1000;
       else data_out = 4'b0000;
    end

endmodule
```

Figure 7.21: Solution to Example 7.4 using an if-else statement.

```
module dcdr_standard_1b(sel, data_out);
    input  [1:0] sel;
    output reg [3:0] data_out;

    always @(sel)
    begin
       case (sel)
          0: data_out = 4'b0001;  //- one hot output
          1: data_out = 4'b0010;
          2: data_out = 4'b0100;
          3: data_out = 4'b1000;
          default data_out = 4'b0000;
       endcase
    end

endmodule
```

Figure 7.22: Solution to Example 7.4 using a case statement.

```
module dcdr_standard_3(s1, s0, data_out);
    input  s1,s0;
    output reg [3:0] data_out;

    always @(s1,s0)
    begin
       case ({s1,s0})  //- concatenation
          0: data_out = 4'b0001;
          1: data_out = 4'b0010;
          2: data_out = 4'b0100;
          3: data_out = 4'b1000;
          default data_out = 0;
       endcase
    end

endmodule
```

Figure 7.23: Solution to Example 7.4 using a case statement.

7.5 Chapter Summary

- If we can model the operation of a digital circuit using a truth table, we have essentially defined a decoder. Decoders are basically the look-up-tables (LUTs) of the digital design world.
- We define two types of decoder in digital design: generic decoders and standard decoders. Standard decoders are a subset of generic decoders; they have an n:2^n relationship regarding the number of inputs:outputs.
- Verilog has special operators for concatenating individual signals to create a signal or larger width.
- Verilog has special syntax to represent binary, octal, and hexadecimal numbers.
- Verilog has two main types of statements that can appear in **always** blocks: 1) *procedural programming statements*, and 2) *procedural assignment statements*. Three main types of procedural programming statements include if, if-else, and case statements. Two main types of procedural assignment statements include *blocking* and *non-blocking* statements.
- Verilog procedural assignment statements utilize different types of operators including equality operators.

8 Multiplexors

8.1 Introduction

When you hear the word multiplexor, or MUX, you probably 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 input and output data can be of any width, but the data widths of the inputs and output always match. Additionally, the width of the select control inputs must be sufficient to choose between the given number of data inputs to the MUX.

8.2 Digital Design Foundation Notation: MUX

We consider the MUX to be one of our Digital Design Foundation circuits and a controlled circuit; Figure 8.1 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 8.1 provides a description of the MUX’s inputs and outputs.

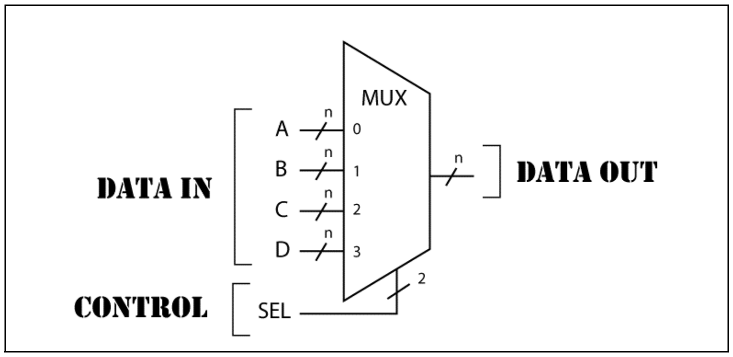


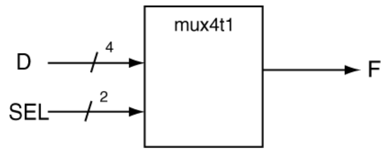
Figure 8.1: Data and control signals for a 4:1 MUX.

	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 8.1: The foundation matrix for a MUX.

Example 8.1: 4:1 MUX Model

Provide a Verilog model describing the following 4:1 MUX.



Solution: Because 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 8.2 and Figure 8.3 show two versions of the solution, one using an **if-else** statements and the other using a **case** statement. Here are a few worthy things to note about these solutions.

- The Verilog models for MUXes are similar in structure to decoder models. Both MUXes and decoders are combinatorial circuits.
- The **if-else** version of the solution uses a bus for the data input (D); we then must access the signals in the bundle individually for assignment to the output.
- We place all input values into the sensitivity list of the **always** block; we do this for all combinatorial circuits.
- Because the MUX is a combinatorial circuit, we make sure to include a catch-all clause in for the procedural programming assignment statements (the **else** for the **if-else** statement and the **default** for the **case** statement).

```
module mux_4t1_a(SEL, D, F);
   input  [1:0] SEL;
   input  [3:0] D;
   output reg F;

   always @(SEL, D)
   begin
      if      (SEL == 0)  F = D[0];
      else if (SEL == 1)  F = D[1];
      else if (SEL == 2)  F = D[2];
      else if (SEL == 3)  F = D[3];
      else                F = 0;
   end

endmodule
```

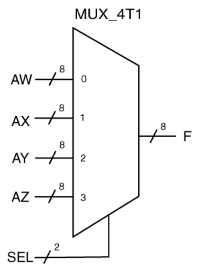
Figure 8.2: The solution to Example 8.1 using an if-else statement.

```
module mux_4t1_b(SEL, D, F);
   input  [1:0] SEL;
   input  [3:0] D;
   output reg F;

   always @(SEL, D)
      case (SEL)
         0: F = D[0];
         1: F = D[1];
         2: F = D[2];
         3: F = D[3];
         default  F = 0;
       endcase

endmodule
```

Figure 8.3: The solution to Example 8.1 using a case statement.

Example 8.2: 4:1 Bundle-Based MUX Model

Provide a Verilog model of the following MUX.



Solution: This MUX is similar to the MUX in the previous problem but this problem chooses between bundled inputs. The resulting models for the bundled and non-bundled data are surprisingly similar, which should not be surprising. Figure 8.4 and Figure 8.5 show two solutions to this example; the model in Figure 8.4 uses an **if-else** statement while the model in Figure 8.5 uses a **case** statement.

```
module mux_4t1_c(SEL, AW, AX, AY, AZ, F);
   input  [1:0] SEL;
   input  [7:0] AW, AX, AY, AZ;
   output reg [7:0] F;

   always @(SEL, AW, AX, AY, AZ)
   begin
      if      (SEL == 0)  F = AW;
      else if (SEL == 1)  F = AX;
      else if (SEL == 2)  F = AY;
      else if (SEL == 3)  F = AZ;
      else                F = 0;
   end

endmodule
```

Figure 8.4: The solution to Example 8.2 using an if-else statement.

```
module mux_4t1_d(SEL, AW, AX, AY, AZ, F);
   input  [1:0] SEL;
   input  [7:0] AW, AX, AY, AZ;
   output reg [7:0] F;

   always @(SEL, AW, AX, AY, AZ)
      case (SEL)
         0: F = AW;
         1: F = AX;
         2: F = AY;
         3: F = AZ;
         default  F = 0;
      endcase

endmodule
```

Figure 8.5: The solution to Example 8.2 using a case statement.

8.3 Chapter Summary

- MUXes are data selection circuits; they choose one of many inputs to pass to the output under control of the data selection inputs.
- MUXes are combinatorial circuits that we model using procedural blocks. Because of the combinatoriality of the circuit, we use procedural programming statements that include catch-all clauses in the modeling. We also use blocking procedural programming statements in their models.

9 Comparators

9.1 Introduction

A comparator is a combinatorial digital device that compares two numbers and provides relational information regarding the results of the comparison. The comparator is a basic digital circuit and one of our Digital Design Foundation Modules. One of interesting aspects of a comparator is that they are somewhat involved to design on a gate-level, but at the same time straightforward to model using an HDL.

9.2 Digital Design Foundation Notation: Comparator

The comparator is a controlled circuit and one of our Digital Design Foundation modules. Figure 9.1 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**.

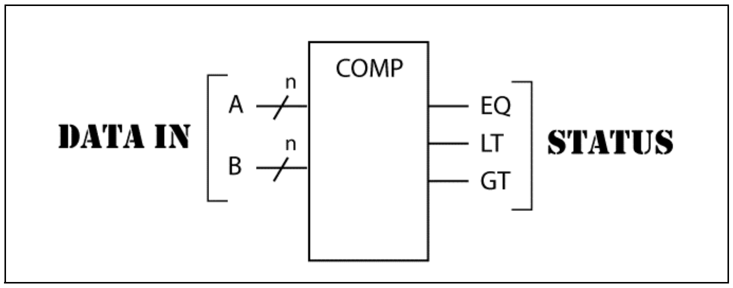


Figure 9.1: Typical data, and status signals for a comparator.

	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 9.1: The foundation matrix for a comparator.

9.3 Verilog Relational Operators

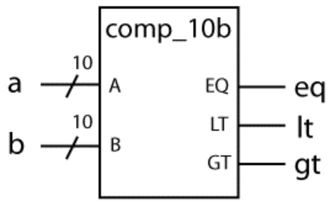
Verilog contains of set of relational operators that we typically use as expressions in procedural assignment statements. The most common use for relational operators is in **if** clauses in **if** & **if-else** statements. Table 9.2 shows the set of Verilog's relational operators.

Operator	Example	Description
`>`	`(X > Y)`	Is X greater than Y? (true or false; 1-bit result)
`<`	`(X < Y)`	Is X less than Y? (true or false; 1-bit result)
`>=`	`(X >= Y)`	Is X greater than or equal to Y? (true or false; 1-bit result)
`<=`	`(X <= Y)`	Is X less than or equal to Y? (true or false; 1-bit result)

Table 9.2: Verilog's relational operators.

Example 9.1: 4:1 10-Bit Comparator

Provide a Verilog model for a 10-bit comparator. This comparator has three outputs: EQ, LT, and GT.



Solution: Modeling comparators is the first circuit that underscores the power of behavioral modeling using HDLs. Figure 8.4 shows the solution to this problem. The solution contains some new items which we describe below.

- The model uses three relational operators to determine the **EQ** (==), **GT**(>), **LT**(<) outputs.
- The comparator is a combinatorial circuit so we include all inputs in the **always** block's sensitivity list. We also include an **else** clause with the **if** statements.
- Each **if** clause assigns all three outputs; if we assigned less than three output per **if** clause, the model would necessarily induce memory, and it would thus not be a proper comparator (comparators are combinatorial circuits).
- Because there are three assignments per **if** clause, we need to include the **begin-end** clause in each **if** clause.
- We placed three blocking assignments per line, which we did primarily to save space on the page. We generally only place multiple assignments on a line when the assignments are relatively closely related, which they are in this case.

```
module comp_10(a, b, eq, lt, gt);
    input  [9:0] a,b;
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0;
       end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end //- always

endmodule
```

Figure 9.2: The solution to Example 9.1 using an if-else statement.

9.4 Generic Models

The comparator is a common digital module that finds its way into many digital designs. These designs basically use the same comparator, but the comparator has different data widths for the inputs. In these cases, we don't want to generate a new module for each data width. The better option is to create a model that has the flexibility to be instantiated using any data width; in this way, the particular instantiation is responsible for declaring the desired data with. This is the generic approach to digital design; it help designers be efficient in that it strongly supports code reuse and reduces the overall text-based size of models (which makes any model less daunting to understand).

Figure 9.3 shows a model for an "n-bit" comparator. We refer to it as an n-bit comparator because the model can use any integer value for n. The given instantiation of this comparator provides the value for "n". Here are some more details regarding the generic model in Figure 9.3.

- The model uses a "variable" to specify the data-width of the two input values. The model officially states the width using the bundle operator that includes the variable "n" ([n-1:0]).
- Because we use a variable in the data-width specification, we also must provide that variable with a value, which the model uses as a default value. We use the *parameter* keyword to specify what we now refer to as the default data-width. We refer to this as a default data-width because if your instantiation does not provide a data width, that particular instance of the comparator defaults of an 8-bit comparator. We use this feature in the next example.
- Other than using a variable in the data-width declaration and the **parameter** definition, the model of this n-bit comparator is the same as the 10-bit comparator from a previous example.

```
module comp_nb(a, b, eq, lt, gt);
    input  [n-1:0] a, b;
    output reg eq, lt, gt;

    //- default input data width
    parameter n = 8;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0;
       end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end

endmodule
```

Figure 9.3: The solution to Example 8.2 using two instances of a generic comparator.

Example 9.2: Module-Based 12-Bit Comparator

Provide a Verilog structural model for a 12-bit comparator by instantiating an 8-bit and 4-bit comparators. This comparator only has an EQ output.

Solution: The solution we provide to this problem is not the optimal solution; if we need a 12-bit comparator, we use the generic model of Figure 9.3. The point of this problem is to use a generic model in a circuit in two different ways. We need to use the generic comparator model and instantiate a 4-bit and 8-bit comparators. Our generic comparator model defaults to an 8-bit comparator, which we deal with in this example's solution. Figure 9.4 shows a BBD for the solution while Figure 9.5 shows the associated Verilog model. Here are a few things of merit to notice about the Verilog model.

- The Verilog code models the BBD in Figure 9.4. We included the signal names to make the diagram better match the model. The choice of how to divide up the 12 signals between eight and four signals is arbitrary.
- We need two instantiations for this example: a 4-bit and 8-bit comparator. We do these instantiations slightly different to show how to use parameterized circuits in Verilog models. The first instantiation is for a 4-bit comparator, so we need the instantiation to override the default data-width of 8 in the generic comparator model. The code overrides the parameter with the "**`#(.n(4))`**" notation appearing between the module name and the instance name in the instantiation code. Note that if we had a circuit with more than one parameterized values, we would provide a commented list after the pound symbol.
- This circuit also requires an 8-bit comparator, so we instantiate an 8-bit version of the generic comparator module. Because we do not need to override the default data-width in the generic comparator, the instantiation code appears like a non-parameterized instantiation. It's always

better to not rely on the default parameter as a way to make your model more clear to human readers.

- The model uses bundle access operators to route the correct signal ranges to the various comparator instances. We arbitrarily place the 4-bit comparator as the low end of the 12 bits.
- The model does not use the **gt** & **lt** signals, but we leave them in the instantiation. We could have not included them, but it would have made human readers wonder whether we forgot to include them or did not need them. Including them and leaving them blank is the best approach.

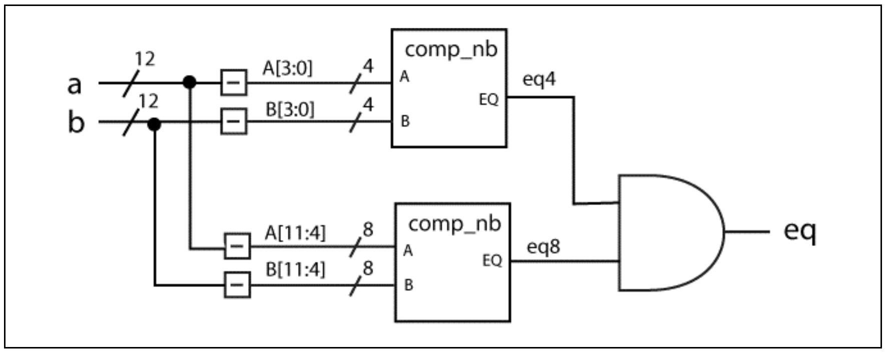


Figure 9.4: BBD for the solution to Example 9.2.

```
module comp_12b(a, b, eq);
   input  [11:0] a,b;
   output eq;

   wire eq4,eq8;

   assign eq = eq4 & eq8;

   //- instantiate 4-bit comparator
   comp_nb  #(.n(4)) MY_COMP4 (
      .a  (a[3:0]),
      .b  (b[3:0]),
      .eq (eq4),
      .gt () ,
      .lt ()    );

   //- instantiate 8-bit comparator
   comp_nb  MY_COMP8 (
      .a  (a[11:4]),
      .b  (b[11:4]),
      .eq (eq8),
      .gt (),
      .lt ()   );

endmodule
```

Figure 9.5: The solution to Example 9.2 using two instances of a generic comparator.

9.5 Chapter Summary

- A comparator is a digital device that compares two data inputs and outputs information regarding the result of the comparison. Typical comparator outputs include: equal, less than, and greater than.
- Comparators are notoriously tedious to model on the gate-level but straightforward to model using HDLs such as Verilog. Verilog models of comparators utilize various relational operators.
- Comparators are common digital circuits, but these circuits have different input data widths. Verilog utilizes language constructs such as parameters to allow digital designers to generate generic models, which enhance the readability of the models.

10 Ripple Carry Adders

10.1 Introduction

Digital circuits typically perform many different arithmetic operations. Our focus in this text is with a simple adder circuit, the ripple carry adder (RCA). Additionally, though Verilog supports many data-types, this text only describes circuits that use unsigned types.

10.2 The RCA: Underlying Details

The RCA is usually one of the first arithmetic modules introduced to beginning digital designers, as it is a relatively simple circuit. Figure 10.1 shows a lower-level BBD for a 4-bit RCA. As you can see, a 4-bit RCA comprises of four 1-bit adders, where we often model the LSB of the adder as a full adder rather than the half adder in Figure 10.1.

Typically we use the RCA as a tool to describe structural modeling, and provide Verilog models for the low-level circuitry. Because we require students to implement an RCA in the lab portion of a typical digital design course, we opt not to provide the models in this text (that's what web searches are for…).

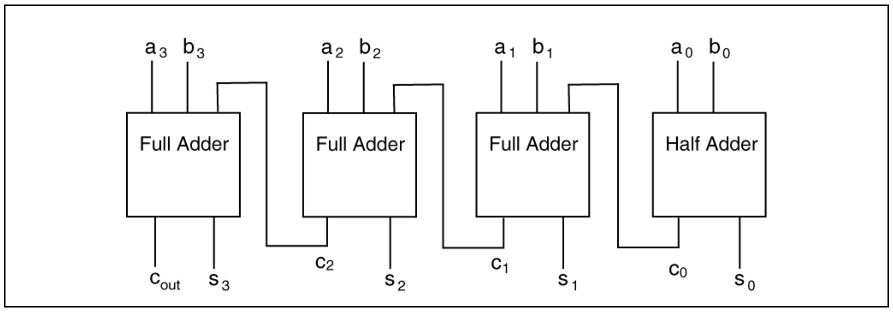


Figure 10.1: Lower-level BBD for a 4-bit Ripple Carry Adder.

10.3 Digital Design Foundation Model

The RCA is a controlled circuit and a combinatorial circuit; it is also a Digital Design Foundation module. Figure 10.2 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 10.1 provides a description of all the inputs and outputs to the RCA.

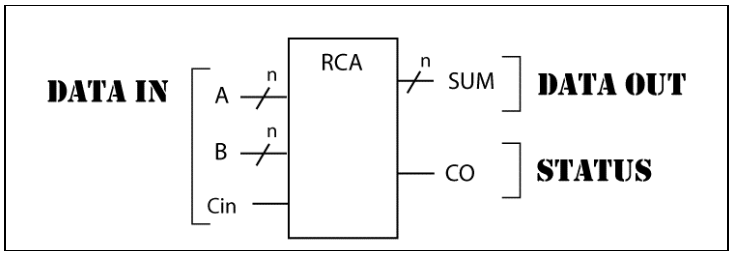


Figure 10.2: Data, control and status signals for a RCA.

	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 10.1: The foundation matrix for a RCA.

10.4 Verilog Arithmetic Operators

Table 10.2 shows a partial list of the arithmetic operators available in Verilog. These operators seem familiar to anyone who has programmed a computer. But being that Verilog is an HDL, there are some other issues involved that are less prominent when programming a computer. We cover those issues in the following subsection.

Operator	Example	Description
`+`	`(X + Y)`	Add X to Y
`-`	`(X - Y)`	Subtract Y from X
`*`	`(X * Y)`	Multiply X by Y
`/`	`(X / Y)`	Divide X by Y
`%`	`(X % Y)`	Modulus of X / Y
`<<<`	`(X <<< Y)`	Shift X left Y-times (zero-fill from right)
`>>>`	`(X >>> Y)`	Shift X right Y times (zero-fill for unsigned numbers)

Table 10.2: Verilog's relational operators.

10.4.1 Arithmetic Operator Issues

There are a few issues that digital designers need to be aware of when using arithmetic operators. The main idea is that we generally use HDL models to synthesize hardware. The digital designer must know a few things about the synthesis process in order to ensure the synthesized circuit works properly. The number of issues is somewhat attenuated because we only work with unsigned types in this text. Here are the items to know to make you into a Verilog super-hero:

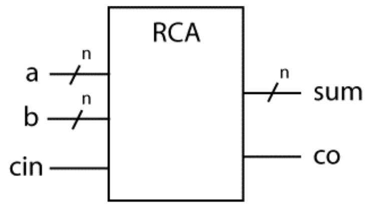
The general rule of statements that contain arithmetic operators is that the operator on the right side of the expression are first expanded to the largest vector size of any vector in the expression, including the vector that the result is assigned to. ***It is thus the digital designer's responsibility to ensure the data-width of the result vector is sufficient to contain the result in the context of the particular arithmetic operator(s) appearing in the expression***. Two conditions can arise as a result of data-width incompatibilities associated with the values appearing in the right-side and left side of the assignment statement.

Result Vector Too Large: In this case, the result is zero-extended (the synthesizer fills the extra unused bits in the result with zeros). This is not generally a problem.

Result Vector Too Small: In this case, the data-width of the result vector is not sufficient to hold the result of the operations on the right-side of the assignment operator. In this case, the synthesizer truncates the higher-order bits of the calculation before assigning it to the result vector. This could be an issue as the result will no longer represent the operations specified on the right side of the assignment statement. MSB of the

Example 10.1: n-Bit Ripple Carry Adder (RCA)

Provide a Verilog model for an n-bit RCA.



Solution: This is a generic RCA; designers have the option of declaring a value for "n" when they instantiate the module. If designers opt to not override the given parameter value, the data-width of the RCA defaults to 8. Figure 10.3 shows the final solution to this example; here are a few interesting items to note about this solution.

- There are many different ways to model an RCA in Verilog; this solution shows one way. As you become more familiar with Verilog, you'll for sure be exploring other options.
- Both output values are **reg**-types because the model assigns these values in the **always** block.
- The RCA is a combinatorial device, which is why we use a blocking assignment in the **always** block.
- The assignment itself is somewhat clever. We assign both the **sum** and **co** in the same assignment statement with the use of the concatenation operator. This highlights a clever but not overly intuitive use of the assignment operator in that the given line uses an assignment to a concatenation operator. The ordering in the concatenation operator is important; the **co** appears on the left side of the comma, which provides position information to the synthesizer regarding the result. Because the **co** appears left of the **sum**, the **co** becomes the MSB of the result.

```
module rca_nb(a, b, cin, sum, co);
    input   [n-1:0] a,b;
    input   cin;
    output  reg [n-1:0] sum;
    output  reg co;

    parameter n = 8;

    always @(a, b, cin)
    begin
       {co, sum} = a + b + cin;
    end

endmodule
```

Figure 10.3: The solution to Example 10.1.

10.5 Chapter Summary

- The ripple carry adder (RCA) is a device that adds two operands (and often a carry-in) and generates a sum and carry-out. The data-width of the sum and to input operands are equivalent. We can use the RCA for subtraction if we change the sign of one of the operand.
- The RCA is a combinatorial device, so we model it include all input operands in the **always** block sensitivity list and use blocking-type assignments.
- Verilog has a modest set of arithmetic operators, including addition and subtraction.
- Use of Verilog operators require the digital designer to be aware of the data-width of the value being assigned to as it relates to the values the arithmetic operators in the expression, which are the left and right side of the assignment operator, respectively.

11 D Flip-Flops

11.1 Introduction

The notion of flip-flops is slowly disappearing from the digital design landscape. The various forms of registers are more common, and the flip-flop is essentially a 1-bit D flip-flop. We dedicate a chapter to the implementation of D flip-flops because there are some Verilog-based modeling issues that are easier to describe using D flip-flops. These issues primarily deal with the notion of memory as the D flip-flop is the first sequential circuit we examine. The good news is that Verilog being what it is, all of the issues associated with D flip-flops transfer to multi-bit registers.

11.2 Modeling Sequential Circuits

There are two distinct approaches to modeling sequential circuits using Verilog. The primary difference between these two approaches is the notion of an asynchronous vs. a synchronous sequential circuit. We consider asynchronous circuits in this text to be some type of latch.

11.2.1 Modeling Asynchronous Circuits

When modeling combinatorial circuits using the **always** procedural block, we made sure the procedural programming statement (the **if-else** or the **case** statement) had a catch-all statement. Providing a catch-all was a message to the synthesizer that the circuit should be combinatorial. When we want the synthesizer to generate an asynchronous sequential circuit, we don't provide catch-all statements to the procedural programming statements. In other words, if we incompletely specify the procedural programming statement, the synthesizer generates a latch.

Figure 11.1 shows an example of a model for an SR latch; here are a few things to note about this model.

- The **always** statement's sensitivity list contains all the inputs to the device (both **S** & **R**).
- The **if-else** statements contains no **else** clause; this means there is no catch-all, which instructs the synthesizer to generate a sequential circuit.

```
module sr_latch(S, R, Q);
    input S, R;
    output reg Q;

    always @ (S, R)
    begin
       if (S == 0 && R == 1)
          Q <= 1'b0;
       else if (S == 1 && R == 0)
          Q <= 1'b1;
    end

endmodule
```

Figure 11.1: An example of latch generation using an incompletely specified if-else statement.

11.2.2 Modeling Synchronous Circuits

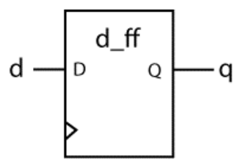
All the sequential circuits in this text are synchronous in nature. This means that most of the changes in the circuit's state are synchronized with the edge of another input signal, which is generally a clock signal. Another way to look at this is that all changes in the circuit are associated with the edge of some signal, whether it be a clocking signal (a true synchronous circuit as the clock is generally periodic) or some other signal non-periodic input signal (such as the edge of a clear signal).

The notion here is that this form of a sequential circuit is sensitive to edges of signals. Verilog uses two keywords to specify that the **always** block is sensitive to a given clock edge. These keywords are **negedge** (for negative edge sensitivity) and **posedge** (for positive edge sensitivity).

11.2.2.1 The Basic D Flip-Flop

Figure 11.2(a) shows the BBD for a basic D flip-flop; Figure 11.2(b) shows the associated Verilog model. Here are few items of interest regarding the Verilog model.

- The sensitivity list shows that the **always** block is sensitive to the positive edge of the **CLK** signal, which means when the module detects a positive edge, the body of the **always** block executes.
- The body of the **always** block that has one non-blocking assignment statement. When we intend on synthesizing sequential circuits, we make sure we use non-blocking assignment statements.



```
module d_ff(D, CLK, Q);
    input D, CLK;
    output reg Q;

    always @ (posedge CLK)
    begin
       Q <= D;
    end

endmodule
```

(a) (b)

Figure 11.2: The basic D flip-flop BBD (a) and associated model (b).

11.2.2.2 The D Flip-flop with an Asynchronous Clear

Figure 11.3(a) shows the BBD for a D flip-flop with active-low asynchronous clear signal; Figure 11.3(b) shows the associated Verilog model. Here are few items of interest regarding the Verilog model.

- The sensitivity list now includes a reference to the falling edge of one signal and the rising edge of another signal.
- The **nCLR** input is interpreted as asynchronous because it is not the **CLK** signal. The **always** block is thus sensitive to the falling edge of the **nCLR** signal. The body of the **always** block first checks to see if the **nCLR** signal is low, and resets the flip-flop when it is low. The way we structure the code in the **always** block indicates that the asynchronous **nCLR** signal has precedence over the **D** input.
- Similar to the basic D flip-flop, this flip-flop latches the D input when there is not a negative edge on **nCLR** signal. This issue here is that the first **if** statement tests as true when a negative edge on the **nCLR** signal caused the **always** block to be evaluated. If the positive edge on the CLK signal caused the **always** block to evaluate, the **always** block ignores the first **if** clause and then executes the second **if** because it was the positive clock edge that caused the evaluation of **always** block.

(a)	(b)

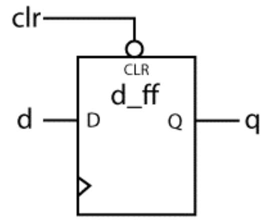
	`module d_ff_nclr_async(D, CLK, Q, nCLR);`
	`    input D, CLK, nCLR;`
	`    output reg Q;`
	
	`    always @ (negedge nCLR, posedge CLK)`
	`    begin`
	`       if (nCLR == 0)`
	`          Q <= 1'b0;`
	`       else`
	`          Q <= D;`
	`    end`
	
	`endmodule`

Figure 11.3: The basic D flip-flop with asynchronous active low clear BBD (a) and associated model (b).

11.2.2.3 The D Flip-flop with a Synchronous Clear

Figure 11.4(a) shows the BBD for a D flip-flop with active-low synchronous clear signal; Figure 11.4 (b) shows the associated Verilog model. Here are few items of interest regarding the Verilog model.

- The model only differs from the D flip-flop with an asynchronous clear input in the **always** block's sensitivity list: this sensitivity list only includes a reference to the **CLK** signal.
- The synthesizer interprets the **nCLR** as synchronous because the **always** block is only sensitive to the **CLK** signal, which means changes in the nCLR signal don't cause an evaluation of the block. The body of the **always** block first checks to see if the **nCLR** signal is low, and resets the flip-flop when it is low. The way we structure the code in the **always** block indicates that the asynchronous **nCLR** signal has precedence over the **D** input.
- This flip-flop latches the D input on an active clock edge and when the **nCLR** signal is not asserted. The code in the **always** block is sequential, which means the **nCLR** signal has precedence over the latching of the **D** input based because of how the code is ordered.
- From looking at the BBD, you can't discern the synchronicity of the **nCLR** input or the precedence of the **D** & **nCLR** inputs; someone must provide this information for you.

(a)	(b)

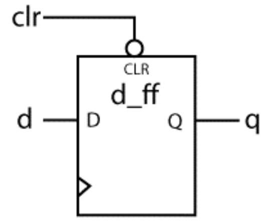
	`module d_ff_nclr_async(D, CLK, Q, nCLR);`
	`    input D, CLK, nCLR;`
	`    output reg Q;`
	
	`    always @ (posedge CLK)`
	`    begin`
	`       if (nCLR == 0)`
	`          Q <= 1'b0;`
	`       else`
	`          Q <= D;`
	`    end`
	
	`endmodule`

Figure 11.4: The basic D flip-flop with asynchronous active low clear BBD (a) and associated model (b).

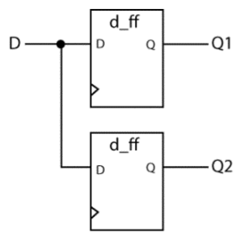
Blocking vs. Non-Blocking Models

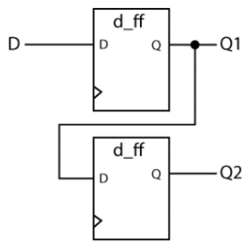
We previously stated directly that we need to always use blocking statements for combinatorial circuits and non-blocking statements for combinatorial circuits. Our claim was that you should use this rule until you get a good feel for how the synthesizer interprets your code to generate circuits. Figure 11.5 provides an example of how the synthesize interprets what looks like the same code, but differs only in the use of blocking vs. non-blocking assignment statements. Here are a few things of interest regarding the code:

- The BBDs do not connect the **CLK** signal in order to keep the BBDs neat.
- The key to understanding the relation between the Verilog models and the associated BBDs is recalling how the synthesizer interprets the blocking and non-blocking statements. Use of the blocking statement means the interpretation of the body of the **always** block halts, or *blocks*, until the evaluation of the blocking statement completes. The code using non-blocking statements never blocks, which means if a statement modifies a variable, other statements in the **always** block can't use the result of the statement until evaluation of the **always** block terminates. Using this information and staring at the resultant circuit diagrams should give you a good idea as to how the synthesizer interprets blocking and non-blocking assignment statements.

```
module d_ff_blocking(D, CLK, Q1, Q2);
    input D, CLK;
    output reg Q1, Q2;

    //- blocking assignment
    always @ (posedge CLK)
    begin
       Q1 = D;
       Q2 = Q1;
    end
endmodule
```

```
module d_ff_non_blocking(D, CLK, Q1,Q2);
    input D, CLK;
    output reg Q1, Q2;

    //- non-blocking assignment
    always @ (posedge CLK)
    begin
       Q1 <= D;
       Q2 <= Q1;
    end
endmodule
```





(a) **(b)**

Figure 11.5: Examples of blocking (a) vs. non-blocking (b) assignment statements and the circuits the synthesizer generatoes.

11.3 Chapter Summary

- Verilog uses two different approaches to synthesizing sequential circuits. These two approaches differ between synchronous and asynchronous circuits. Verilog uses incompletely specified procedural programming statements to generate asynchronous circuits. Verilog uses the **posedge** and **negedge** keywords to generate synchronous circuits.
- The D flip-flop is the basic 1-bit synchronous storage element in digital design.

12 Registers

12.1 Introduction

Registers are one of the most common digital circuits. The family of registers include D flip-flops (1-bit registers), counters, and shift registers. The latter two items are topics in upcoming chapters. A register is a synchronous sequential device that is able to store data. Counters and shift registers are essentially registers with features.

There not much to say about registers because we've already covered most of the important information in our descriptions and associated Verilog models of D flip-flops. As you'll see from our generic register model, the models for D flip-flops and registers only differ in the associated data widths.

12.2 Digital Design Foundation Notation: Registers

The register is a synchronous sequential circuit; it is a controlled circuit and is a Digital Design Foundation modules. Figure 12.1 shows the 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 (**CLR**). Table 12.1 show a complete description of the registers input and output signals.

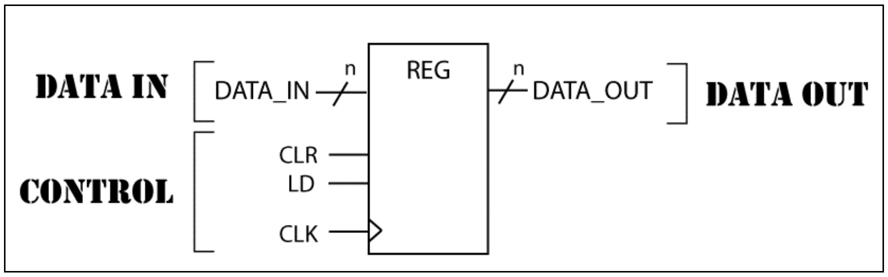


Figure 12.1: Typical data and control signals for a register.

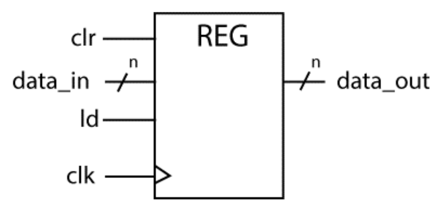
	Signal Name	Description
INPUT DATA	**DATA_IN**	The data that can be latched into the register's storage elements.
OUTPUT DATA	**DATA_OUT**	The DATA_OUT signal is the data currently being stored in the counter's storage elements.
CONTROL	**CLK**	Registers are synchronous circuits, in that the loading 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 12.1: The foundation description for a simple register.

12.3 Generic Register Model

Figure 12.2(a) shows the BBD for a generic register; Figure 12.2(b) shows the associated Verilog model. Here are a few items to note about the Verilog model of Figure 12.2(b):

- The model is very similar to the D flip-flop with an asynchronous clear; the two models primarily differ in data widths.
- The model is parameterized, and thus defaults to a data-width of 8 if an instantiation of this device does not override the default parameter.
- The register is a synchronous sequential device, so we use **posedge** keywords in the sensitivity list and use non-blocking assignment statements in the body of the **always** statement.



```
module reg_nb(data_in, clk, clr, ld., data_out);
    input  [n-1:0] data_in;
    input  clk, clr, ld;
    output reg [n-1:0] data_out;

    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
       if (clr == 1)      // asynch clr
          data_out <= 0;
       else if (ld == 1)
          data_out <= data_in;
    end

endmodule
```

(a) **(b)**

Figure 12.2: The basic register BBD (a) and associated model (b).

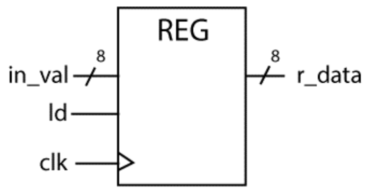
12.4 Registers vs. `reg`-Type Variables

Digital design has registers and Verilog has **reg**-types variables. We all know what registers are and what they do, but we're just learning about Verilog **reg**-type variables. The truth is that referring to the variable as a **reg** is highly misleading. A register is a sequential device, which means it stores data. In Verilog, the values being assigned in **always** blocks must be reg-types, but being a **reg**-type does not magically give the variable memory capability. Recall that we can use **always** blocks to generate either combinatorial or sequential circuits. Whether there is storage associated with the **reg**-type or not is a matter of how the **always** blocks handles the given variable. Don't be fooled. A good practice is to prefix an "r_" in front of variable names that you're going to use as storage to let the human reader know that the variable has memory associated with it.

12.5 Inline Registers

This chapter primarily presented registers as separate modules that you can instantiate into your design. While this approach is useful, it is not the only possible approach, as you can also model "inline" registers. In this case, we use the work *inline* to indicate a register that we're not instantiating. Recall that an **always** block is essentially a module in a BBD. Figure 12.3(b) shows a fragment of Verilog code that induces a register associated with the BBD of Figure 12.3(a); here are some notable features of the given model.

- The code that uses this model must define **ld** & **clk** as single-bit signals somewhere else in the design as either inputs, **wire**s, or reg-types. Similarly, the code must define **in_val** as an 8-bit vector as either an input of a reg-type.
- By the way we structure the code, **r_data** is the storage element as it is assigned in such a way as to ensure the synthesizer induces a memory element associated with the **r_data** variable.
- The design must define **r_data** as reg-type because the code assigns a value to **r_data** in the body of the **always** block.
- We use an "r_" prefix on the label associated with the memory element to indicate to human readers of the code that the code is using "**r_data**" as a register. This is good coding practice as it clarifies the usage of **r_data** with little coding effort.

	
	```
	    //- ld, clk: single signals in design

	    //- in_val: 8-bit signal in design

	    reg [7:0] r_data; //- the storage element

	    always @(posedge clk)
	    begin
	       if (ld == 1)
	          r_data <= in_val;
	    end

	endmodule
	```

(a)	(b)

Figure 12.3: A generic register BBD (a) and a fragment of it's inline Verilog model (b).

12.6 Chapter Summary

- A register is a synchronous sequential circuit. We use basic registers to store multi-bit data. We typically model registers (in our heads, at least) as a set of D flip-flops connected in parallel.
- Modeling registers in Verilog is similar to modeling D flip-flops, as the D flip-flop is essentially a 1-bit register.

13 Finite State Machine Modeling

13.1 Introduction

Finite State Machines (FSMs) is a topic that spans many different technical fields. Digital electronics, however, typically use FSMs as a means to control digital circuits. We can design FSMs at many different levels; typical digital design texts design them at low-level using various flip-flops and gates. We are most interested in modeling FSMs at a higher-level, as this approach lends itself nicely to modern digital design-based computer aided design (CAD) tools.

Based on the power of Verilog, there are many different approaches to modeling FSMs; this text examines only one approach. We feel this is the simplest approach for those new to Verilog. Additionally, this approach gives you a feel for the number of states in the FSM and how they relate to the width of the state registers. Once you gain more Verilog modeling skills you may consider using another approach.

13.2 FSM Overview

Figure 13.1 shows the standard medium-level model for an FSM. There are three basic parts to the FSM, which include the *next state decoder*, the *output decoder*, and the *state registers*. We mention these here for historical purposes; we'll soon be abstracting our FSMs to a higher level. The general idea behind a FSM acting as a controller is that the FSM reacts to the external inputs in such a way as to send out the appropriate control outputs. We understand the external inputs to be status signals from the external circuitry that the FSM controls; the FSMs output are control signals that make the external circuitry operate appropriately.

One of the main highlights of Figure 13.1 is the notion that FSMs can have two different types of outputs. Moore-type outputs are a function of the state of the FSM only, while Mealy-type outputs are a function of both the FSM's state and the external inputs. We could say more about these outputs, but that is a topic better covered in a basic digital design textbook (consider looking at Digital Design Foundation Modeling for a cool approach at a good price point).

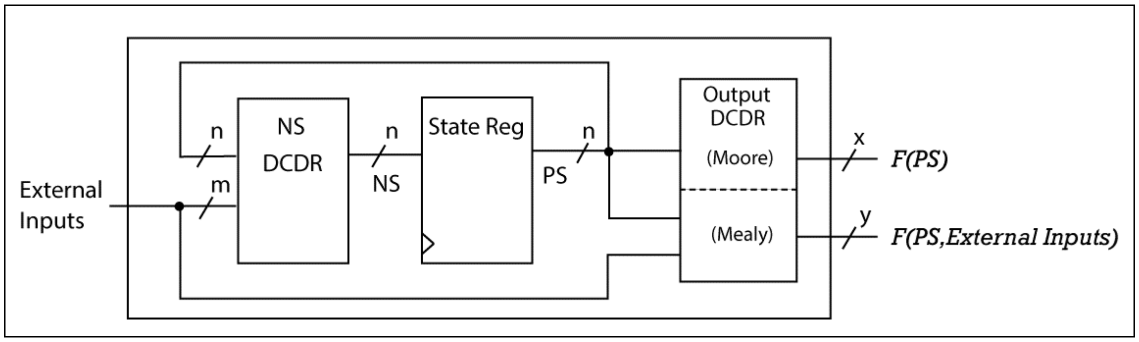


Figure 13.1: The lower-level BBD for a generic FSM.

13.2.1 State Diagrams

There are many ways to describe the operation of FSMs, but the most efficient approach for humans is to use a state diagram. Using state diagrams to represent FSMs is part science and part art, where the main goal is to create state diagrams that quickly transfer as much information as possible to the human viewer. Once again, the topic of state diagrams is a topic best covered in a digital design textbook; this text assumes you have a working knowledge of state diagrams.

This chapter is about modeling FSMs using Verilog behavioral modeling. The stating point of modeling FSM using Verilog is the state diagram. As you will see, generating the state diagram is the "engineering step" in the process; translating a state diagram to a Verilog behavioral model is somewhat cookbook, and turns out to be gruntwork once you pick up some working knowledge of the process.

13.3 FSM Using Verilog Behavioral Modeling

Figure 13.1 shows the standard approach to modeling at a fairly low level. Because modeling FSMs at a low level is not efficient, we first want to use a BBD to describe how will model FSMs at a higher level of abstraction. Figure 13.2 shows a BBD modeling our Verilog behavior descriptions of FSM; here are a few items of interest to note about this model.

- There are two blocks: a sequential block and a combinatorial block. This means that we now divide the functionality of the three boxes in Figure 13.1 into the two boxes in Figure 13.2.
- The SEQ_CKT box is the sequential circuit that handles the state registers. This box is synchronous but it often has asynchronous control inputs such as reset signals. The main purpose of the SEQ_CKT box is to implement the state changes in the circuit; thus the COMB_CKT box determines what the next state should be based on the present state and external inputs, the SEQ_CKT latches that new state into the state registers thus making it the present state.
- The COMB_CKT box is a combinatorial circuit that essentially implements both the next state and output decoder. The next state is a function of the present state and the external inputs. The COMB_CKT box also determines the values for the Mealy and Moore-type outputs.

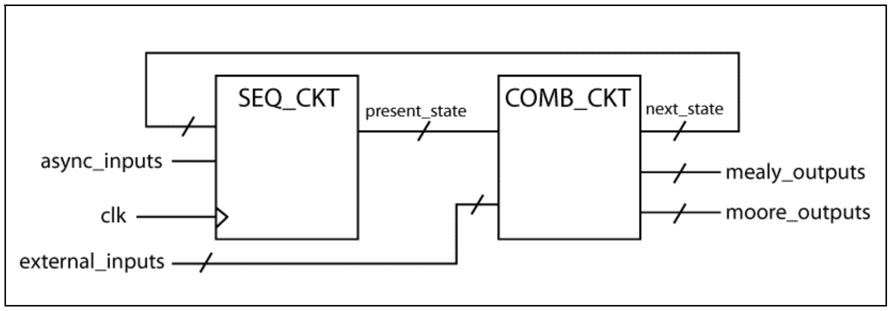
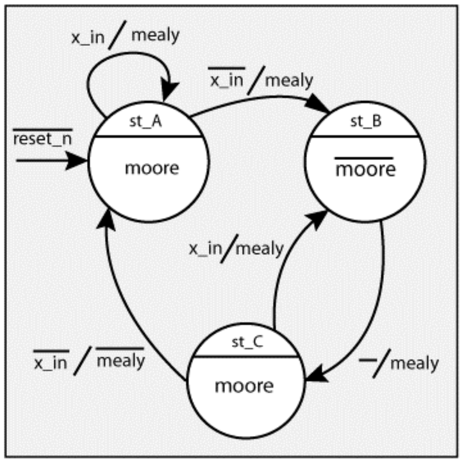


Figure 13.2: A BBD showing the Verilog behavioral modeling approach to implementing FSMs.

Example 13.1: Verilog FSM Model Implemenation #1

Use a Verilog behavioral model to describe the follow state diagram.



Solution: This is a contrived state diagram that shows how to implement every possible characteristic in a FSM. Note that from the state diagram we have both a Mealy and Moore-type output, an asynchronous reset input, and one external input. It does not get any more complicated than this. Figure 13.3 show the results of the first step in this example, which is to generate a BBD. Figure 13.4 shows the final solution to this example. Here are all the features to note about his example.

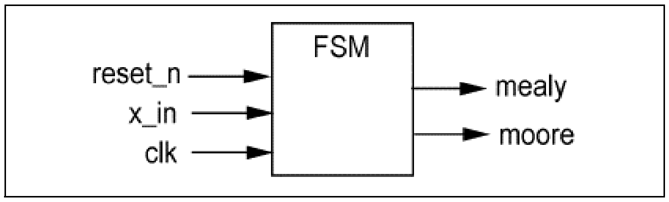


Figure 13.3: The BBD for this problem's FSM.

- The modeling approach uses two **always** blocks, one block is for the combinatorial processes that handles issues specific to the output and next state decoder; the other **always** block handles issues associated with the state registers, including asynchronous inputs.
- The model must first create variables that handle the state mechanism, as well as how exactly the FSM encodes those states. The model gives the appropriate names of **NS** and **PS** for the state information, which not surprisingly stands for next state and present state. The model then uses the parameter data-type to assigned specific binary values to each of the states in the state diagram. The **parameter** keyword assigns constant values to the state names.
- The state diagram has three states, so we need at least two bits as state variables. We arbitrarily chose the different bit combinations, which leave ones bit combination used, which is potentially a hang state.
- We prefix the state names with a "st_" to give the human reader that the particular variable is the name of a state, which represents good self-commenting Verilog code.
- The **always** block associated with the sequential process is sensitive to the edges: the negative edge of the **reset_n** signal and the positive edge of the **clk** signal. The **reset_n** signal is asynchronous because its evaluation is independent of the **clk** signal. The **reset_n** signal also

has a higher precedence than the **clk** signal based on the sequential[18] nature of the code in the body of the **always** block. All signal assignments in the **always** block are blocking.

- The second **always** block (the sequential **always** block) is sensitive to changes in the external input signal (**x_in**) and changes in the signal representing the present state (**PS**). Thus, a change in either of these signals causes the evaluation of the **always** block.
- The first thing we do in the second **always** block is to assign all external outputs a value, which we do to ensure the synthesizer never generates a latch for any output. The alternative to this is to assign every output in every state, which becomes increasingly hard to read as the number of control outputs increase for a given FSM.
- The sequential **always** block contains one procedural programming statement: a case statement. This **case** statement has one clause for each state in the state diagram and one **default** clause to make the FSM self-correcting.
- Each clause in the **case** statement has three responsibilities: 1) assign the Moore-type outputs, 2) assign the Mealy-type outputs, and, 3) assign the next state. The Moore-type output assignments are a function of the state only, so that assignment always appears first in the associated state. Both the next state (**NS**) and Mealy-type outputs are a function of the external inputs, so we must use a procedural programming statement (an if-else) to correctly assign both **NS** and the Mealy-type output.
- The "st_B" state transitions unconditionally to the "st_C" state. Because the external input does not control this transition, the Mealy-type output is constant. We thus have no **if-else** clause in this section of the case statement.
- All the **case** clauses use **if-else** procedural programming statements, meaning that using the **else** clause provides a catch-all statement to partially ensure the synthesizer generates no latches and to ensure the FSM is deterministic.
- The case statement contains a default statement. The FSM should never find itself in this condition, but it if does, the FSM uses the default external output assignments and directs the FSM to state "st_A". A good debug strategy is to have the FSM output something that is very noticeable in simulation if it by chance finds itself in a state it should not be in.

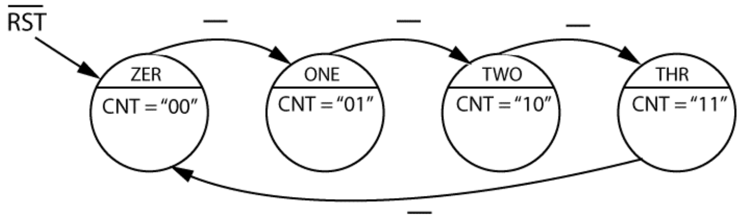
[18] This means sequential as the always block reads the statements in the order they appear; this does not mean sequential in relation to a type of digital circuit.

```
module fsm_template(reset_n, x_in, clk, mealy, moore);
    input  reset_n, x_in, clk;
    output reg mealy, moore;

    //- next state & present state variables
    reg [1:0] NS, PS;
    //- bit-level state representations
    parameter [1:0] st_A=2'b00, st_B=2'b01, st_C=2'b11;

    //- model the state registers
    always @ (negedge reset_n, posedge clk)
       if (reset_n == 0)
          PS <= st_A;
       else
          PS <= NS;

    //- model the next-state and output decoders
    always @ (x_in,PS)
    begin
       mealy = 0; moore = 0; // assign all outputs
       case(PS)
          st_A:  //-------------------------------
          begin
             moore = 1;
             if (x_in == 1)
             begin
                mealy = 0;
                NS = st_A;
             end
             else
             begin
                mealy = 1;
                NS = st_B;
             end
          end

          st_B: //---------------------------------
             begin
                moore = 0;
                mealy = 1;
                NS = st_C;
             end

          st_C: //---------------------------------
             begin
                 moore = 1;
                 if (x_in == 1)
                 begin
                    mealy = 1;
                    NS = st_B;
                 end
                 else
                 begin
                    mealy = 0;
                    NS = st_A;
                 end
             end

          default: NS = st_A;
          endcase
      end
endmodule
```

Figure 13.4: The solution to Example 13.1.

Example 13.2: Verilog FSM Model Implemenation #1

Use a Verilog behavioral model to describe the follow state diagram.



Solution: The state diagram provides all the information we need to complete this problem. Figure 13.5 shows the result of the best first step to take, which is to draw the BBD. You certainly don't have to draw the BBD, but doing so helps prevent dumbtarted mistakes in the Verilog modeling process. We can note from the state diagram that this FSM has one output: **CNT**, which is a Moore-type output. All state transitions are unconditional, and the circuit has an active low reset input (**RST**), which places the FSM into the ZER state.

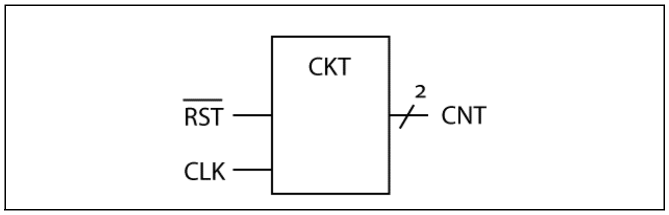


Figure 13.5: The BBD for this problem's FSM.

Figure 13.6 shows the final Verilog model for this example; here are a few non-boring things to note about this solution.

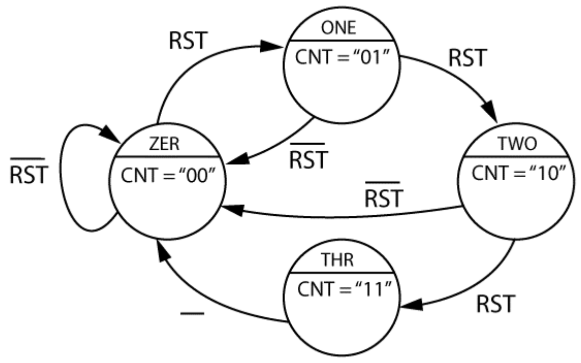
- The code in the model uses white space (blank line & indentation) to enhance readability. The code is adequately commented as well.
- All state transitions are unconditional, so the individual clauses in the case statement do not include **if-else** constructs.
- The state names in the model closely match the state labels in the state diagram, which enhances the understandability of the model to human viewers.

```
module fsm_example_01(rst, clk, cnt);
    input rst, clk;
    output reg [1:0] cnt;

    //- next state & present state variables
    reg [1:0] NS, PS;
    parameter [1:0] st_ZER=2'b00, st_ONE=2'b01, st_TWO=2'b10, st_THR=2'b11;

    //- model the state registers
    always @ (negedge rst, posedge clk)
       if (rst == 0)
          PS <= st_ZER;
       else
          PS <= NS;

    //- model the next-state and output decoders
    always @ (PS)
    begin
       case(PS)
          st_ZER: //------------------------------
          begin
             cnt = 2'b00;
             NS = st_ONE;
          end

          st_ONE: //------------------------------
          begin
             cnt = 2'b01;
             NS = st_TWO;
          end

          st_TWO: //------------------------------
          begin
             cnt = 2'b10;
             NS = st_THR;
          end

          st_THR: //------------------------------
          begin
             cnt = 2'b11;
             NS = st_ZER;
          end

          default: NS = st_ONE;

        endcase
    end    //- always

endmodule
```

Figure 13.6: The solution to Example 13.2.

Example 13.3: Verilog FSM Model Implemenation #3

Use a Verilog behavioral model to describe the follow state diagram.



Solution: This problem is the similar to the previous problem, but now the **RST** input is synchronous. The first thing to note is that the **RST** is always associated with state-to-state transitions, rather than the nowhere-to-state transitions associated with the asynchronous flavor of **RST**. Figure 13.7 shows the BBD associated with this FSM while Figure 13.8 shows the final solution.

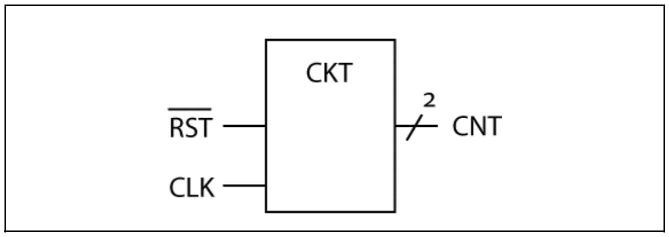


Figure 13.7: The BBD for this problem's FSM.

The major differences between the solution to this problem and the solution to the previous problem involve the modeling of the **RST** input. Here are some specifics:

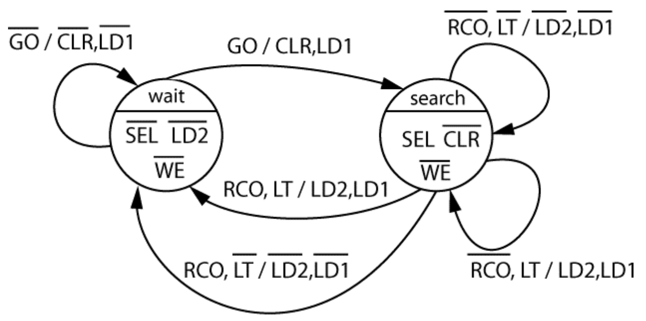
- The model using the asynchronous **RST** signal handles the signal in the sequential process while the asynchronous version handles **RST** in the combinatorial process. We could have modeled **RST** in the combinatorial process, but it avoids later confusion to place asynchronous signal implementation in the sequential process.
- Because we now model **RST** in the combinatorial process, each of the clauses of the **case** statement now contain **if-else** statements. The associated **always** block is combinatorial, so we must include an **else** statement.
- In state "st_THR", the RST signal does not matter as the FSM always transitions to "st_ZER". Thus, the state transitions are a function of **RST** and the state for the all states but "st_THR", where it is a function of state only.
- The **CNT** output is only a function of state so we model it as a Moore-type output.
- The model contains a default clause. This FSM uses all the 2-bit code space, so illegal state recovery is not an issue for this model.

```
module fsm_example_02(rst, clk, cnt);
    input rst, clk;
    output reg [1:0] cnt;

    //- next state & present state variables
    reg [1:0] NS, PS;
    parameter [1:0] st_ZER=2'b00, st_ONE=2'b01, st_TWO=2'b10, st_THR=2'b11;

    //- model the state registers
    always @ (posedge clk)
       if (clk == 1)
           PS <= NS;

    //- model the next-state and output decoders
    always @ (PS)
    begin
       case(PS)
          st_ZER: //-----------------------------------
          begin
             cnt = 2'b00;
             if (rst == 0)
                NS = st_ZER;
             else
                NS = st_ONE;
          end

          st_ONE: //-----------------------------------
          begin
             cnt = 2'b01;
             if (rst == 0)
                NS = st_ZER;
             else
                NS = st_TWO;
          end

          st_TWO: //-----------------------------------
          begin
             cnt = 2'b10;
             if (rst == 0)
                NS = st_ZER;
             else
                NS = st_THR;
          end

          st_THR: //-----------------------------------
          begin
             cnt = 2'b11;
             NS = st_ZER;
          end

          default: NS = st_ZER;

        endcase
     end

endmodule
```

Figure 13.8: The solution to Example 13.3.

Example 13.4: Verilog FSM Model Implemenation #4

Use a Verilog behavioral model to describe the follow state diagram.


$\overline{GO}$ / $\overline{CLR}$,$\overline{LD1}$
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

Solution: This state diagram has fewer states than the previous examples, but the state transitions and associated conditions are more complicated. The state diagram indicates that the FSM has four inputs: **RCO**, **GO**, **LT**, and **CLK** (implied). The state diagram indicates that the FSM will have five outputs: **LD1**, **LD2**, **SEL**, **WE**, and **CLR**. We know that **WE** and **SEL** are Moore-type outputs because they are only a function of the state; the other outputs are Mealy-type outputs because they are a function of both state and external inputs. Figure 13.9 shows the result of the first step in this solution, which is to draw the BBD.

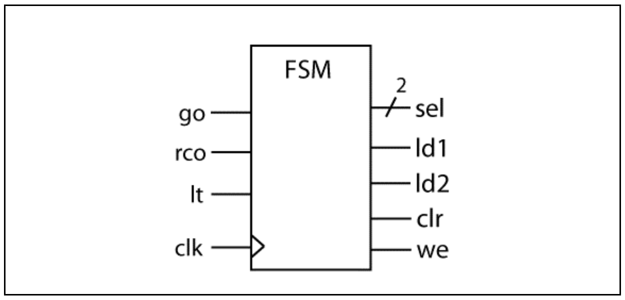


Figure 13.9: The top-level BBD for the FSM.

The next step in this solution is to generate the Verilog model; Figure 13.10 shows the solution. Here are a few things to note about this solution.

- A state diagram of this complexity lends itself of an infinite number of models. The one in Figure 13.10 is the one we feel is more appropriate based on space limitations of this text and complexity of the model in general. This is not necessarily the optimal solution, but it fits on a page without shrinking the font.
- The model uses a combination of **if-else** clauses both with and without **begin-end** pairs. Once again, we do this for space considerations and not for readability.
- The second clause in the **case** statement could have used an embedded **case** statement, but we opted to use **if-else** clauses. We could have been clever here and assigned everything in one statement, but we opted to keep the solution as clear and simple as possible.
- For both **if-else** clauses in the “st_search” state, we both list every possible input combination and also provide an **else** clause. We could have used one less **if-else** clause and replaced it with an **else**, but we always like to see all possible options listed as part of the **if-else** clause rather than relying on **else** clause to cover the final set of output options.

```
module fsm_example_03(go, rco, lt, clk, we, ld1, ld2, clr, sel);
    input go, rco, lt, clk;
    output reg clr, ld1, ld2, we, sel;

    //- next state & present state variables
    reg  NS, PS;
    parameter st_wait=1'b0, st_search=1'b1;

    //- model the state registers
    always @ (posedge clk)
       if (clk == 1)
           PS <= NS;

    //- model the next-state and output decoders
    always @ (PS)
    begin
       we = 0; ld1 = 0; ld2 = 0; clr = 0; sel = 0;
       case(PS)
          st_wait: //----------------------------------
          begin
             sel = 0;  we = 0;  ld2 = 0;
             if (go == 0)   begin
                clr = 0;  ld1 = 0;   end
             else
             begin
                clr = 1;  ld1 = 1;
             end
          end

          st_search: //--------------------------------
          begin
             //- handle outputs
             sel <= 0;     we <= 0;    clr <= 0;
             if (rco == 0 && lt == 0) begin
                ld2 <= 0; ld1 <= 0;  end
             else if (rco == 0 && lt == 1) begin
                ld2 <= 1; ld1 <= 1;  end
             else if (rco == 1 && lt == 0) begin
                ld2 <= 1; ld1 <= 1;  end
             else if (rco == 1 && lt == 1) begin
                ld2 <= 0; ld1 <= 1;  end
             else
             begin
                ld2 <= 0; ld1 <= 1;
             end

             //- handle state transitions
             if (rco == 1)
                NS <= st_wait;
             else if (rco ==0)
                NS <= st_search;
             else
                NS <= st_wait;
          end

          default: NS = st_wait;
        endcase
     end
endmodule
```

Figure 13.10: The solution to Example 13.4.

13.4 Chapter Summary

- Many different technical fields use FSMs to model the operations of processes; digital design primarily uses FSMs as digital circuits that control other digital circuits.
- The engineering step in FSM design is the creation of the state diagram; most everything else beyond that point is grunt-work based on how straightforward FSMs are to model in Verilog. The state diagram indicates how it controls a circuit; any type of model of the state diagram is much less impressive.
- The accepted FSM hardware model includes three basic parts: 1) next state decoder, 2) output decoder, and 3) state registers, where only the state registers are a sequential circuit. There are several approaches to modeling a FSM using Verilog, the most simple model uses two blocks: one to handle the combinatorial sections of the FSM (the next state and output decoders), and one to handle the sequential part of the FSM (the state registers).
- Modeling FSMs using Verilog is a cut & paste process based on the notion that the challenging step is generating the original state diagram. Working from a template is always the best approach rather than starting from scratch.

14 Counters

14.1 Introduction

A counter is a type of register; we generally consider it a register with “features”. The counter is a synchronous sequential circuit and one of our Digital Design Foundation modules. There are many different flavors of counters such as up counter, down counters, up/down counters, etc. You can design counters at many different levels, we only design counters at the behavioral level in this text.

14.2 Digital Design Foundation Notation: Counters

The counter is a controlled circuit and is both sequential and synchronous; Figure 14.1 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.

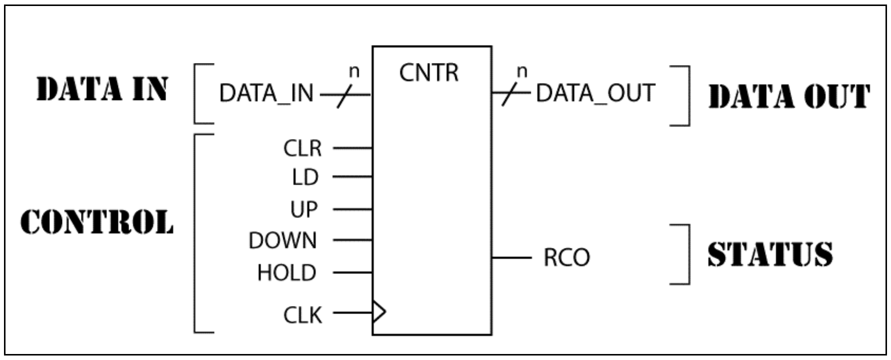


Figure 14.1: Typical data, control and status signals for a counter.

Table 14.1 shows all the inputs and outputs that we can typically associate with a counter. Table 14.1 lists a set of features that we can apply to a counter. The two things to note about this list are 1) that not every counter has every feature, and 2) actual counter implementations typically combine many of the required control features into fewer signals than listed.

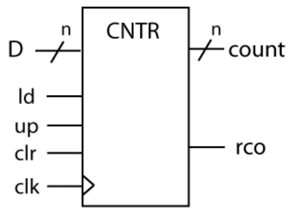
	Signal Name	Description
INPUT DATA	**DATA_IN**	A counter is a register, so it can typically load data in to the counter's storage elements. The DATA_IN input is the data that is loaded to the counter.
OUTPUT DATA	**DATA_OUT**	A counter is a register, so the DATA_OUT signal is the data currently being 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 (ripple carry out) 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 14.1: The foundation description for a full-featured counter.

14.3 Generic N-Bit Up/Down Counter Model

Figure 14.2(a) shows a BBD for a generic n-bit up/down counter with asynchronous clear while Figure 14.2(b) shows the associated Verilog model. Here are a few issues of merit about this model.

- The model is parameterized and thus defaults to an 8-bit counter. We override this default value when we instantiate the module.
- This counter is an up/down counter; it counts either up or down based on the value of the **up** input.
- The counter uses two different arithmetic operators; the model uses the addition operator (+) to increment the count value and the subtraction operator (-) to decrement the count value. The status of the **up** signal determines the count direction.
- The counter has an **rco** value, which indicates when the counter reaches its terminal count. Because this counter counts both up and down, we need to verify the count direction before ascertaining when the **rco** is asserted or not. The lower **always** block generates **rco**; note that this model is combinatorial.
- The **rco** calculation uses two applications of replication operators as part of the **if** statement to determine if the count is all 0's or all 1's.



```
module cntr_udclr_nb(clk, clr, up, ld, D, count, rco);
    input   clk, clr, up, ld;
    input   [n-1:0] D;
    output  reg [n-1:0] count;
    output  reg rco;

    //- default data-width
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)       // asynch reset
           count <= 0;
        else if (ld == 1)   // load new value
           count <= D;
        else if (up == 1)   // count up (increment)
           count <= count + 1;
        else if (up == 0)   // count down (decrement)
           count <= count - 1;
    end


    //- handles the RCO, which is direction dependent
    always @(count, up)
    begin
       if ( up == 1 && &count == 1'b1)
          rco = 1'b1;
       else if (up == 0 && |count == 1'b1)
          rco = 1'b1;
       else
          rco = 1'b0;
    end

endmodule
```

Figure 14.2: A BBD and a Verilog model for an N-bit up/down counter with asynchronous clear.

14.4 Chapter Summary

- Counters are synchronous sequential circuits; they are essentially registers with a special repeatable output sequence we refer to as the count.
- Counter generally have the feature set of register, include clear, and load inputs (depending on how you model the counter).
- Counters often include status outputs such as RCO (ripple carry out) that indicate when the counter has reached its terminal count.

15 Shift Registers

15.1 Introduction

The shift register is similar to the counter in that it is another "register with features". The main feature of a shift register is that it "shifts" the data in its internal storage elements. While these shifts don't sound too exciting, they are quite useful in many digital circuits. In particular, the single-bit left shift is a fast multiply by two operation, while a single-bit right shift is a fast divide by two operation. The shifting action of a shift register is interestingly useful for other purposes in digital circuits.

This chapter describes what we refer to as a universal shift register (USR). The notion of a USR means that the module does more than a single shift in one direction. Digital designers can model USRs to do other shifting type operations as arithmetic shifts, barrel shifts, and rotates.

15.2 Digital Design Foundation Notation: Shift Register

We consider the shift register a Digital Design Foundation module; the shift register is a sequential and 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 15.1 shows the foundation module for a shift register.

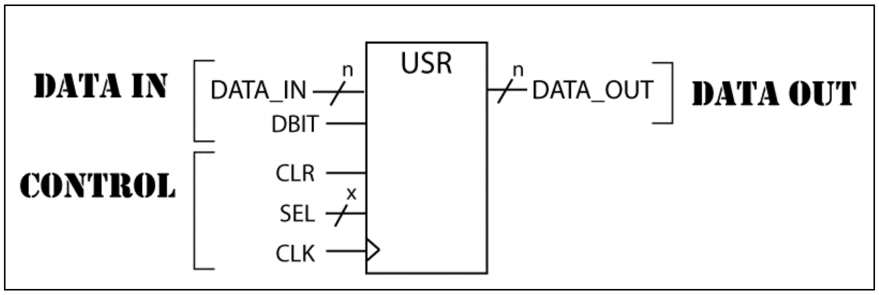


Figure 15.1: Typical data, control and status signals for a universal shift register.

The shift register model we provide is very similar to a simple register model, but includes some extra features. We only included a load, hold, shift left, and shift right in the model, but we organized the model such that it is straightforward to add other special features that our circuits may require. Thus, we often use this shift register model as a starting point for more feature-laden circuits.

	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 15.1: The foundation description for a universal shift register.

```
module usr_nb(data_in, dbit, sel, clk, clr, data_out);
    input  [n-1:0] data_in;
    input  dbit, clk, clr;
    input  [1:0] sel;
    output reg [n-1:0] data_out;

    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)     // asynch reset
           data_out <= 0;
        else
           case (sel)
              0: data_out <= data_out;                 // hold value
              1: data_out <= data_in;                  // load
              2: data_out <= {data_out[n-2:0],dbit};  // shift left
              3: data_out <= {dbit,data_out[n-1:1]};  // shift right
              default data_out <= 0;
           endcase
    end

endmodule
```

Figure 15.2: The solution to Example 10.1.

15.3 Chapter Summary

- A shift register is a sequential digital device that performs operations that we can describe as shift-like.
- A simple shift register shifts one bit location in one direction, but is not a useful device. We generally work with universal shift registers that perform more than one type of operation based on the devices selection input.
- Other types of shifting operations include arithmetic shifts, barrel shifts, and rotates.
- Shift-left and shift right operations implement multiplication and division by two, respectively. These are known to be fast operations because they are simple to implement in hardware. Each single shift left or right is a multiply or divide by two, respectively. We obtain higher powers of two operations by multiple shifts.

16 Testbenches

16.1 Introduction

Testbenches are an important part of HDL modeling. The main issue is that the synthesizer has the ability to interpret your HDL models in ways you would not anticipate. One of the main themes of this text is to keep your models intended for synthesis as simple as possible in order to give the synthesizer as few options as possible (as few knobs as possible). As circuits become larger, it becomes harder to keep the HDL models simple, and you must work with the synthesizer.

Working with the synthesizer is a two-step process. First you must have a basic understanding of how the synthesizer operates. Second, you must always test the circuit the synthesizer provides for you. In other words, designing a circuit is half the battle, testing the circuit is the other half[19].

Designers know that many of the constructs in Verilog language do not synthesize into digital hardware. This may sound strange, but it underscores the fact that circuit verification is a significant part of the digital design process. There are many interesting constructs in Verilog designed specifically for verification; this text uses only a few of them[20]. Circuit verification is part art and part science; it's a truly in-depth topic. The science portion of verification primarily involves writing models with as much coverage as possible for the circuit being tested; the other part involves creating generic and automatic test models that run to completion and state directly whether your circuit works as expected. In academia, the main focus of courses that use HDLs are the design and generation of circuits; the testing portion of circuit design is unfortunately highly attenuated due to time constraints.

16.2 Hardware Modeling vs. Hardware Verification

Any writing you find out there regarding HDLs always places a strong emphasis on the fact that using HDLs is inherently different from writing software. Moreover, such writing rightfully claims that if you use the HDL's syntax in a manner similar to writing software, your circuit has less chance of working. But here we are in the chapter presenting testbenches. It turns out that a significant amount of the constructs in any HDL are "not synthesizable". So what are they there for? They're there to help you verify the HDL code you write to model a circuit.

This text does not go deeply into verification due to time constraints associated with typical introductory digital design courses. Because of this, you won't get a good feel for the "art" of writing testbenches. Here's the funny issue… you tossed out your programming skills to become a person skilled at modeling digital circuit. If your job is to verify synthesized HDL models, you need to pick back up those computer programming skills, as writing HDL code for verification is more like writing software than it is modeling hardware. Once again, this text does not go there; but be prepared if your instructor of boss needs you to do some viable verification.

16.3 Testbenches

Test benches range from quite simple to massively complex depending on their intended purpose; the test bench can sometimes become more complicated than the actual circuit you are testing. Figure 16.1 shows a general model of a testbench. The test bench comprises of two main components: the *stimulus driver* and the *design under test*. The design under test, or *DUT*, is the HDL model you intend on testing. The stimulus driver is a HDL model that communicates with the DUT. In the general case, the stimulus driver provides test vectors

[19] It's probably more than half the battle in real life; it's usually less than half the battle when first learning to model circuits with HDLs.

[20] System Verilog has even more constructs that are designed specifically for verification, meaning the language constructs are not synthesizable.

to the DUT and also examines results. The stimulus driver can also interact with the external environment, which allows for reading test vectors from files and writing various data and status notes to files. In this text, our stimulus drives only provide information to the DUT; the DUT does not provide status back to the stimulus driver.

The stimulus driver is nothing special in terms of a HDL model. The main difference here is that instead of dealing with signals that interface with the outside world (such a switches and LEDs), we're now dealing with signals that are driving the unit we intend to test. Note that in Figure 16.1 there are no signals touching the dotted line, therefore the testbench itself has no inputs or outputs. The testbench generally has two sets of modules: one is the DUT, which you instantiate into your testbench; everything else in the testbench is part of the stimulus driver. This is clearer when you see it in an example.

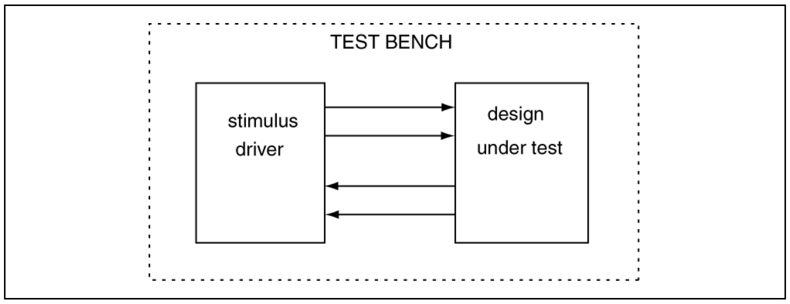


Figure 16.1: The general model of an HDL testbench.

Example 16.1: Testbench Example #1

Use the Verilog model of a 4-bit comparator in Figure 16.1 to generate a basic testbench model.

Solution: Figure 16.2 shows a model of a 4-bit comparator with eq, lt, & gt outputs. Figure 16.3 shows the final testbench for this example.

```
module comp_4b(a, b, eq, lt, gt);
    input  [3:0] a,b;
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0; end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end
endmodule
```

Figure 16.2: The Verilog model for a 4-bit comparator.

Here are the more pertinent features of the testbench model of Figure 16.3.

- The argument list for the testbench module declaration is empty. Note that in Figure 16.1 that there are no inputs or outputs to or from the testbench box.
- The testbench model then provides declarations that the stimulus driver and DUT output uses. The rule here is that we always declare outputs from the stimulus driver (inputs to the DUT) as **reg**-type and we always declare outputs from the DUT as **wire**-types. Both declarations use **wire**-types as needed.
- The model uses an **initial** procedural block to generate stimulus to the DUT as a function of time. This block is similar to an **always** block, but the block is only executed one time.
- We use the "#" to indicate relative time in the **initial** block. The **initial** block only specifies inputs to the DUT; the simulator automatically generates the outputs. The first data specifications do not contain time indicators, which mean the simulation start at time zero. The next time the code changes the test vectors is 20 time units later as indicated by the "#20". This code changes the values two more times. The second "#20" officially occurs 40 time units after the vectors are first assigned as this model uses relative time.
- The testbench provides all inputs values to the DUT with initial values; if we did not do this, the simulator would post unknown values on both the DUT's inputs and outputs.

```
module tb_comp_4b(   );
   //- stimulus connections to DUT
   reg [3:0] a, b;       //- stimulus outputs
   wire eq, lt, gt;      //- DUT outputs

   //- DUT instantiation
   comp_4b MY_COMP (
      .a  (a),
      .b  (b),
      .eq (eq),
      .lt (lt),
      .gt (gt)   );

   initial
      begin
         //- initial values of a & b
         a = 'hA;
         b = 'hB;

         //- a & b values 20 time units later
         #20 a = 'hB;
             b = 'hB;

         //- a & b values 20 time units later
         #20 a = 4'b1011;
             b = 4'b0001;

         //- a & b values 20 time units later
         #20 a = 4'b0001;
             b = 4'b0001;
      end
endmodule
```

Figure 16.3: The model of an HDL testbench for a 4-bit comparator

Example 16.2: Testbench Example #2

Use the Verilog model of an n-bit comparator in Figure 16.4(a) to generate a testbench model.

Solution: Figure 16.4 shows a model of a 4-bit comparator with eq, lt, & gt outputs. Figure 16.4 shows the final testbench for this example. This problem is similar to the previous problem; the difference being that this example uses a generic version of the comparator and instantiates two 4-bit comparators to form an 8 bit comparator.

```
module comp_nb(a, b, eq, lt, gt);
    input  [n-1:0] a,b;
    output reg eq, lt, gt;

    parameter n = 8;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0; end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end

endmodule
```

(a)

```
module tb_comp_nb(   );
   reg [3:0] a, b;
   wire eq, lt, gt;

   //- DUT instantiation
   comp_nb #(.n(4)) MY_COMP (
      .a  (a),
      .b  (b),
      .eq (eq),
      .lt (lt),
      .gt (gt)   );

   initial
   begin
      a = 'hA;
      b = 'hB;

      #20 a = 'hB;    //- time=20
          b = 'hB;

      #20 a = 4'b1011; //- time=40
          b = 4'b0001;

      #20 a = 4'b0001; //- time=60
          b = 4'b0001;
      end
endmodule
```

(b)

Figure 16.4: The Verilog model for an n-bit comparator (a) and an associated testbench (b).

Example 16.3: Testbench Example #3: Generic Up/Down Counter

Use the Verilog model of an n-bit counter in Figure 16.4 to generate a basic testbench model for an 8-bit counter.

Solution: Figure 16.5 shows a model of an n-bit counter; Figure 16.6 shows the final testbench for this solution. This solution has one major difference found in most circuits, which is a clock generator.

```
module cntr_udclr_nb(clk, clr, up, ld, D, count, rco);
    input  clk, clr, up, ld;
    input  [n-1:0] D;
    output   reg [n-1:0] count;
    output   reg rco;


    //- default data-width
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)       // asynch reset
           count <= 0;
        else if (ld == 1)   // load new value
           count <= D;
        else if (up == 1)   // count up (increment)
           count <= count + 1;
        else if (up == 0)   // count down (decrement)
           count <= count - 1;
    end


    //- handles the RCO, which is direction dependent
    always @(count, up)
    begin
       if ( up == 1 && &count == 1'b1)
          rco = 1'b1;
       else if (up == 0 && |count == 1'b0)
          rco <= 1'b1;
       else
          rco <= 1'b0;
    end

endmodule
```

Figure 16.5: The Verilog model of an n-bit up/down counter.

Figure 16.6 shows the testbench that supports the n-bit counter. Here are a few comments of interest regarding the counter this testbench model.

- The original model defaults to an 8-bit (see parameter in n-bit counter model), so the testbench utilizes this default value. The testbench knows it needs to work with an 8-bit counter, so the testbench declares the D input and count output as an 8-bit vector. The instantiation of the DUT does not override the default value.
- The counter is a synchronous circuit, so it requires the stimulus driver to provide a clock signal. We opt to generate a periodic signal for testing as well. There are many ways to generate a clock signal in a Verilog testbench, this solution shows one of those ways. The testbench model uses a **forever** statement inside of an initial block for the clock generator. This could have been done with an **always** block, but the clk signal would need to be provided with an initial value in some other process.
- The testbench start the counter at a relatively high 8-bit value, which causes the counter to roll over after a few clock cycles. After that, the testbench changes the clock direction to count down. The up count asserts the rco when the counter reaches its terminal count. The **rco** asserts once again in the down direction when the count reaches zero. Rather exciting stuff…

```
module tb_comp_nb(   );
   reg [7:0] D;
   reg clk, clr, up, ld;
   wire [7:0] count;
   wire rco;

   cntr_udclr_nb MY_CNTR (
       .clk   (clk),
       .clr   (clr),
       .up    (up),
       .ld    (ld),
       .D     (D),
       .count (count),
       .rco   (rco)   );

   //- Generate periodic clock signal
   initial
   begin
      clk = 0;   //- init signal
      forever  #10 clk = ~clk;
   end;

   initial
      begin
         clk = 0;
         up = 1;
         ld = 0;
         D =  'hFB;
         clr = 0;

         //- send out LD pulse
         #10 ld = 1;
         #30 ld = 0;

         //- change count direction
         #200 up = 0;

      end

endmodule
```

Figure 16.6: The testbench for the n-bit counter.

16.4 Chapter Summary

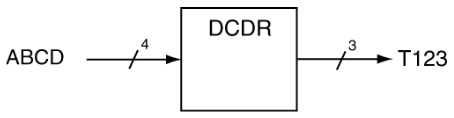
- Testbenches are HDL models designed to verify the correct operation of HDL models.
- The verification HDL models are significantly different than HDL models intended for synthesis. HDL models intended for verification are very similar to software; HDL models intended for synthesis are very different from software.
- A significant portion of Verilog’s various language constructs can’t be synthesized; they exist primarily for verification purposes.
- Writing HDL models for verification is both an art and science; this text only touches upon a few simple examples.

Appendix

Digital Design Foundation Module Templates

Generic Decoder

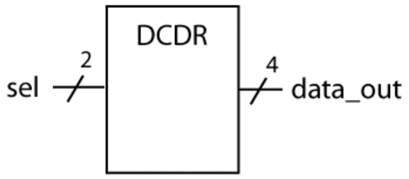
```
module dcdr_ex2a(ABCD, T123);
    input  [3:0] ABCD;
    output reg [2:0] T123;

    always @(ABCD)
    begin
       if       (ABCD == 0)  T123 = 3'b110;
       else if (ABCD == 1)  T123 = 3'b000;
       else if (ABCD == 2)  T123 = 3'b100;
       else if (ABCD == 3)  T123 = 3'b010;
       else if (ABCD == 4)  T123 = 3'b000;
       else if (ABCD == 5)  T123 = 3'b101;
       else if (ABCD == 6)  T123 = 3'b000;
       else if (ABCD == 7)  T123 = 3'b101;
       else if (ABCD == 8)  T123 = 3'b010;
       else                 T123 = 3'b000;
    end

endmodule
```



A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

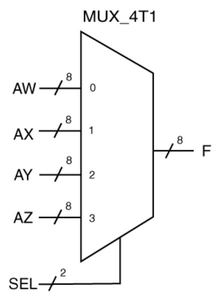
2: 4 Standard Decoder

```
module dcdr_standard_2t4(sel, data_out);
    input  [1:0] sel;
    output reg [3:0] data_out;

    always @(sel)
    begin
       case (sel)
          0: data_out = 4'b0001;  // 1-hot output
          1: data_out = 4'b0010;
          2: data_out = 4'b0100;
          3: data_out = 4'b1000;
          default data_out = 4'b0000;
       endcase
    end

endmodule
```

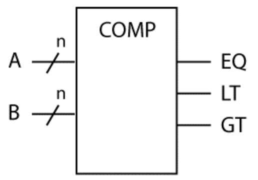


Multiplexor (MUX)

```
module mux_4t1_d(SEL, AW, AX, AY, AZ, F);
   input  [1:0] SEL;
   input  [7:0] AW, AX, AY, AZ;
   output reg [7:0] F;

   always @(SEL, AW, AX, AY, AZ)
      case (SEL)
         0: F = AW;
         1: F = AX;
         2: F = AY;
         3: F = AZ;
         default  F = 0;
      endcase

endmodule
```

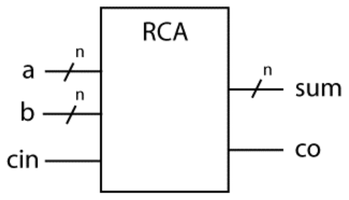


N-Bit Comparator



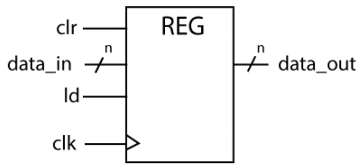
```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////
// Company: Ratner Engineering
// Engineer: James Ratner
//
// Create Date: 07/04/2018 02:13:56 PM
// Design Name:
// Module Name: comp_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: n-bit comparator model
//
//   Usage (instantiation) example for 16-bit comparator
//          (model defaults to 8-bit comparator)
//
//      comp_nb #(.n(16)) MY_COMP (
//          .a (my_a),
//          .b (my_b),
//          .eq (my_eq),
//          .gt (my_gt),
//          .lt (my_lt)
//          );
//
// Dependencies:
//
// Revision:
// Revision 1.00 - File Created: 07-06-2018
// Additional Comments:
/////////////////////////////////////////////////////////////

module comp_nb(a, b, eq, lt, gt);
    input  [n-1:0] a, b;
    output reg eq, lt, gt;

    parameter n = 8;

    always @ (a, b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0; end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end

endmodule
```

N-Bit Ripple Carry Adder (RCA)



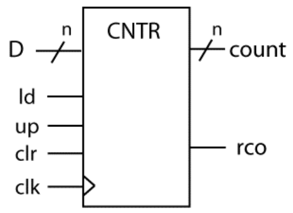
```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////
// Company: Ratner Engineering
// Engineer: James Ratner
//
// Create Date: 07/04/2018 02:13:56 PM
// Design Name:
// Module Name: rca_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: n-bit RCA model
//
//   Usage (instantiation) example for 16-bit RCA
//          (model defaults to 8-bit RCA)
//
//      rca_nb #(.n(16)) MY_RCA (
//          .a (my_a),
//          .b (my_b),
//          .cin (my_cin),
//          .sum (my_sum),
//          .co (my_co)
//          );
//
// Dependencies:
//
// Revision:
// Revision 1.00 - File Created: 07-04-2018
// Additional Comments:
//
///////////////////////////////////////////////////////


module rca_nb(a, b, cin, sum, co);
    input   [n-1:0] a, b;
    input   cin;
    output  reg [n-1:0] sum;
    output  reg co;

    //- default bit-width
    parameter n = 8;

    always @(a, b, cin)
    begin
       {co, sum} = a + b + cin;
    end

endmodule
```

N-Bit Register



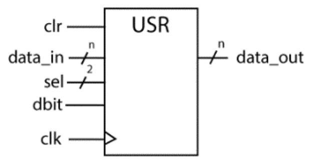
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Company:   Ratner Surf Designs
// Engineer:  James Ratner
//
// Create Date: 09/08/2018 07:17:37 PM
// Design Name:
// Module Name: reg_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Model for generic register (defaults to 8 bits)
//                with asynchronous clear
//      //- Usage example for instantiating 16-bit register
//      reg_nb #(.n(16)) MY_REG (
//          .data_in  (my_data_in),
//          .ld       (my_ld),
//          .clk      (my_clk),
//          .clr      (my_clr),
//          .data_out (my_data_out)
//          );
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////

module reg_nb(data_in, clk, clr, ld, data_out);
    input  [n-1:0] data_in;
    input  clk, clr, ld;
    output reg [n-1:0] data_out;

    //- default bit-width specification
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
       if (clr == 1)     // asynch clr
          data_out <= 0;
       else if (ld == 1)
          data_out <= data_in;
    end

endmodule
```

N-Bit UP/DOWN Counter with Asynchronous Clear



```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Company:   Ratner Surf Designs
// Engineer:  James Ratner
//
// Create Date: 07/04/2018 02:46:31 PM
// Design Name:
// Module Name: cntr_udclr_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Generic n-bit up/down counter with asynchronous
//     reset. This counter also has RCO that works for both
//     up & down counting.
//
//      cntr_udclr_nb #(.n(16)) MY_CNTR (
//          .clk   (my_clk),
//          .clr   (my_clr),
//          .up    (my_up),
//          .ld    (my_ld),
//          .D     (my_D),
//          .count (my_count),
//          .rco   (my_rco)   );
//
// Revision:
// Revision 1.00 - File Created (07-06-2018)
// Additional Comments:
//
/////////////////////////////////////////////////////////////////

module cntr_udclr_nb(clk, clr, up, ld, D, count, rco);
    input  clk, clr, up, ld;
    input  [n-1:0] D;
    output   reg [n-1:0] count;
    output   reg rco;

    //- default data-width
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)       // asynch reset
           count <= 0;
        else if (ld == 1)   // load new value
           count <= D;
        else if (up == 1)   // count up (increment)
           count <= count + 1;
        else if (up == 0)   // count down (decrement)
           count <= count - 1;
    end


    //- handles the RCO, which is direction dependent
    always @(count, up)
    begin
       if ( up == 1 && &count == 1'b1)
          rco = 1'b1;
       else if (up == 0 && |count == 1'b0)
          rco = 1'b1;
       else
          rco = 1'b0;
    end

endmodule
```

N-Bit Universal Shift Register

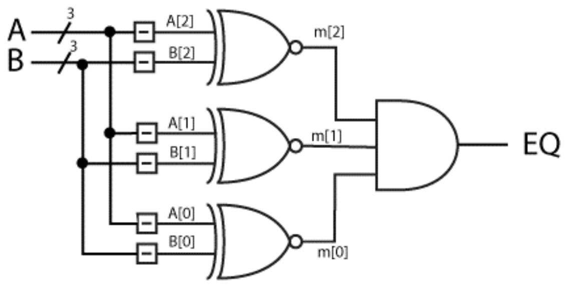


```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////
// Company:   Ratner Surf Designs
// Engineer:  James Ratner
//
// Create Date: 07/04/2018 02:46:31 PM
// Design Name:
// Module Name: usr_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Generic n-bit shift register
//      with an asynchronous reset.
//
//      usr_nb #(.n(16)) MY_USR (
//          .data_in (my_data_in),
//          .dbit (my_dbit),
//          .sel (my_sel),
//          .clk (my_clk),
//          .clr (my_clr),
//          .data_out (my_data_out)
//          );
//
// Dependencies:
//
// Revision:
// Revision 1.00 - File Created (07-06-2018)
// Additional Comments:
//////////////////////////////////////////////////////////

module usr_nb(data_in, dbit, sel, clk, clr, data_out);
   input  [n-1:0] data_in;
   input  dbit, clk, clr;
   input  [1:0] sel;
   output reg [n-1:0] data_out;

   parameter n = 8;

   always @(posedge clr, posedge clk)
   begin
       if (clr == 1)     // asynch reset
          data_out <= 0;
       else
          case (sel)
             0: data_out <= data_out;                // hold value
             1: data_out <= data_in;                 // load
             2: data_out <= {data_out[n-2:0],dbit}; // shift left
             3: data_out <= {dbit,data_out[n-1:1]}; // shift right
             default data_out <= 0;
          endcase
   end

endmodule
```

Finite State Machines (FSMs)

CheatSheets

Structural Modeling CheatSheet



```
//- definition of XOR gate
module my_xor(A, B, F);
   input  A,B;
   output F;

   assign F = A ^ B;
endmodule

//- definition of 3-input AND gate
module my_and(A, B, C, F);
   input  A, B, C;
   output F;

   assign F = A & B & C;
endmodule

//- definition of 3-bit comparator
module comp_3b(A,B,EQ);
    //- external interface signals
    input [2:0] A,B;
    output EQ;

    //- internal interface signals
    wire [2:0] m;


    //- XOR instantiations
    my_xor XOR2 (
      .A (A[2]),
      .B (B[2]),
      .F (m[2])    );

    //- XOR instantiation
    my_xor XOR1 (
      .A (A[1]),
      .B (B[1]),
      .F (m[1])    );

    //- XOR instantiation
   my_xor XOR0 (
      .A (A[0]),
      .B (B[0]),
      .F (m[0])   );

    //- AND instantiation
   my_and  AND0 (
      .A (m[2]),
      .B (m[1]),
      .C (m[0]),
      .F (EQ)   );

endmodule
```

Finite State Machine CheatSheet

```
module fsm_template(
    input logic reset_n, x_in, clk,
    output logic mealy, moore
    );
    reg [1:0] NS, PS;
    parameter [1:0] st_A=2b'00, st_B=2b'01, st_C=2b'10;
```

user-defined type for states

```
    always @ (negedge reset_n, posedge clk)
       if (reset_n == 0) PS <= st_A;
       else              PS <= NS;
```

state register definition

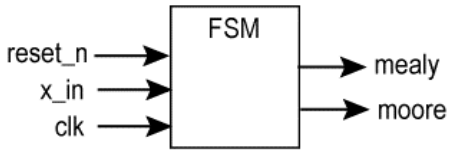
```
    always @ (x_in, PS)
    begin
       mealy = 0; moore = 0;
       case(PS)
```

assign all outputs to avoid latches

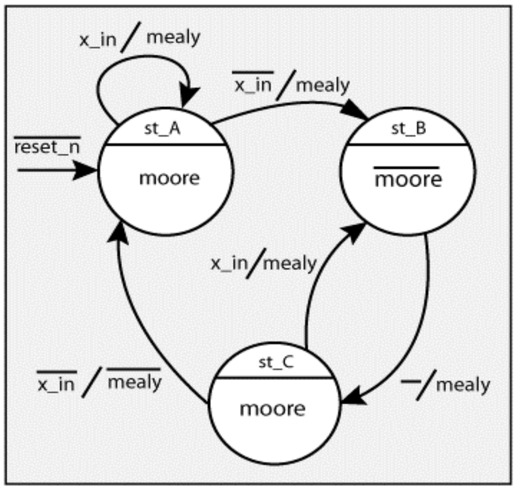
```
       st_A:
          begin
             moore = 1;
             if (x_in == 1)
             begin
                mealy = 0;
                NS = st_A;
             end
             else
             begin
                mealy = 1;
                NS = st_B;
             end
          end
```

Moore outputs are function of state only

Mealy outputs are function of state and external input



```
       st_B:
          begin
             moore = 0;
             mealy = 1;
             NS = st_C;
          end

       st_C:
          begin
              moore = 1;
              if (x_in == 1)
              begin
                 mealy = 1;
                 NS = st_B;
              end
              else
              begin
                 mealy = 0;
                 NS = st_A;
              end
          end
```

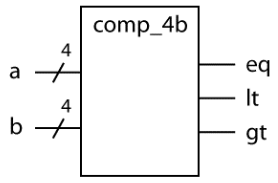


```
       default: NS = st_A;
       endcase
    end  // always

endmodule
```

illegal state recovery

Testbench CheatSheet

```
module comp_4b(a, b, eq, lt, gt);
    input  [3:0] a,b;
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0;
       end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end
endmodule
```

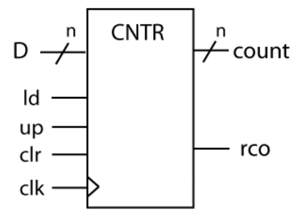


```
module tb_comp_4b(   );
   reg [3:0] a, b;       //- stimulus outputs
   wire eq, lt, gt;      //- DUT outputs

   //- DUT instantiation
   comp_4b MY_COMP (
      .a  (a),
      .b  (b),
      .eq (eq),
      .lt (lt),
      .gt (gt)   );

   initial
      begin
         //- initial values of a & b
         a = 'hA;
         b = 'hB;

         //- a & b values 20 time units later
         #20 a = 'hB;
             b = 'hB;

         //- a & b values 20 time units later
         #20 a = 4'b1011;
             b = 4'b0001;

         //- a & b values 20 time units later
         #20 a = 4'b0001;
             b = 4'b0001;
      end

endmodule
```

Testbench CheatSheet

```
module
cntr_udclr_nb(clk,clr,up,ld,D,count,rco);
    input  clk, clr, up, ld;
    input  [n-1:0] D;
    output   reg [n-1:0] count;
    output   reg rco;

    //- default data-width
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)        //- asynch reset
           count <= 0;
        else if (ld == 1)    //- load value
           count <= D;
        else if (up == 1)    //- increment
           count <= count + 1;
        else if (up == 0)    //- decrement
           count <= count - 1;
    end

    //- handles the direction dependent RCO
    always @(count, up)
    begin
       if ( up == 1 && &count == 1'b1)
          rco = 1'b1;
       else if (up == 0 && |count == 1'b0)
          rco <= 1'b1;
       else
          rco <= 1'b0;
    end

endmodule
```

```
module tb_comp_nb(   );
   reg [7:0] D;
   reg clk, clr, up, ld;
   wire [7:0] count;
   wire rco;

   cntr_udclr_nb MY_CNTR (
       .clk   (clk),
       .clr   (clr),
       .up    (up),
       .ld    (ld),
       .D     (D),
       .count (count),
       .rco   (rco)   );

   //- Generate periodic clock signal
   initial
   begin
      clk = 0;   //- init signal
      forever  #10 clk = ~clk;
   end;

   initial
      begin
         clk = 0;
         up = 1;
         ld = 0;
         D =  'hFB;
         clr = 0;

         //- send out LD pulse
         #10 ld = 1;
         #30 ld = 0;

         //- change count direction
         #200 up = 0;

      end

endmodule
```



Index