FreeRange Verilog Foundation Modeling

With SystemVerilog Support

James Mealy © 2021

V4.02

Table of Contents

TABLE OF CONTENTS - 2 -
PRETENTIONS - 7 -
ACKNOWLEDGEMENTS - 8 -
THE PURPOSE OF THIS BOOK - 9 -
OVERVIEW OF CHAPTERS - 10 -
1 INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGES - 12 -
1.1 INTRODUCTION - 12 -
1.2 PURPOSES OF HDLS - 12 -
1.3 PROGRAMMING COMPUTERS VS. MODELING DIGITAL CIRCUITS - 12 -
1.3.1 Programming Computers - 13 -
1.3.2 Modeling Digital Circuits - 13 -
1.4 DIGITAL DESIGN AND MODELING DIGITAL CIRCUITS WITH AN HDL - 14 -
1.5 INTRODUCING VERILOG VS. SYSTEMVERILOG - 15 -
1.5.1 Verilog's Relation to SystemVerilog - 15 -
1.5.2 The Word on Verilog vs. SystemVerilog - 16 -
1.6 THE GOLDEN RULES OF MODELING DIGITAL CIRCUITS WITH HDL - 16 -
1.7 THE FINAL WORD - 17 -
1.8 CHAPTER SUMMARY - 18 -
1.9 CHAPTER EXERCISES - 19 -
2 DIGITAL DESIGN & DIGITAL DESIGN FOUNDATION MODELING - 20 -
2.1 INTRODUCTION - 20 -
2.2 DIGITAL DESIGN OVERVIEW - 20 -
2.2.1 Basic Tenets of Digital Design - 20 -
2.2.2 Digital Circuit Types - 20 -
2.2.3 Finite State Machines (FSMs) - 21 -
2.2.4 The Three Approaches to Digital Design - 22 -
2.3 PRINCIPLES OF DIGITAL DESIGN FOUNDATION MODELING - 23 -
2.3.1 DDFM Overview - 23 -
2.4 CHAPTER SUMMARY - 25 -
2.5 CHAPTER EXERCISES - 26 -
3 INTRODUCTION TO VERILOG - 27 -
3.1 INTRODUCTION - 27 -
3.2 HISTORY - 27 -
3.3 THE HDL GOVERNING CONCEPT OF CONCURRENCY - 27 -
3.4 MODELING DIGITAL CIRCUITS WITH VERILOG - 28 -
3.4.1 Dataflow Descriptions (Dataflow Models) - 28 -
3.4.2 Behavioral Descriptions (Behavioral Models) - 28 -
3.4.3 Using Dataflow or Behavioral Descriptions - 29 -
3.5 VERILOG INVARIANTS - 29 -
3.5.1 Coding Style & Documentation - 29 -
3.5.2 Statement Termination - 30 -
3.5.3 Reserved Words - 30 -
3.6 CHAPTER SUMMARY - 32 -

3.7 CHAPTER EXERCISES - 33 -

4 DATAFLOW MODELING - 34 -

4.1 INTRODUCTION - 34 -
4.2 BASIC VERILOG MODEL STRUCTURE - 34 -
4.3 SOME VERILOG DETAILS - 36 -
4.3.1 Verilog Bitwise Operators - 36 -
4.3.2 Verilog Nets and Variables - 37 -
4.3.3 Summary of Verilog `wire` Nets - 38 -
4.4 SCALAR AND VECTOR DATA TYPES - 39 -
4.5 SYSTEMVERILOG CONSIDERATIONS - 41 -
4.6 CHAPTER SUMMARY - 42 -
4.7 CHAPTER EXERCISES - 43 -

5 STRUCTURAL MODELING - 46 -

5.1 INTRODUCTION - 46 -
5.2 MODELING: BEFORE WE START - 46 -
5.3 STRUCTURAL MODELING SYNTAX - 46 -
5.4 UNARY REDUCTION OPERATORS - 51 -
5.5 THE FINAL WORD - 52 -
5.5.1 Structural Model's Relation to Higher-Level Computer Programming - 52 -
5.6 SYSTEMVERILOG CONSIDERATIONS - 53 -
5.7 CHAPTER SUMMARY - 54 -
5.8 CHAPTER EXERCISES - 55 -

6 BEHAVIORAL MODELING - 56 -

6.1 INTRODUCTION - 56 -
6.2 THE PATH OF BEHAVIORAL MODELING - 56 -
6.3 BEHAVIORAL MODELING WITH PROCEDURAL BLOCKS - 56 -
6.3.1 The `always` Block - 56 -
6.3.2 The Guts of the Always Block - 57 -
6.3.3 Combinatorial vs. Sequential Circuit Modeling - 58 -
6.4 SYSTEMVERILOG CONSIDERATIONS - 59 -
6.5 CHAPTER SUMMARY - 61 -
6.6 CHAPTER EXERCISES - 62 -

7 DECODERS - 63 -

7.1 INTRODUCTION - 63 -
7.2 TYPES OF DECODERS - 63 -
7.2.1 Generic Decoders - 63 -
7.2.2 Standard Decoders - 64 -
7.3 VERILOG SUPPORT ISSUES - 66 -
7.3.1 Variable Type for Behavioral Modeling: the `reg` - 66 -
7.3.2 Concatenating Signals - 67 -
7.3.3 Representing Integer Numbers - 68 -
7.4 THE `ALWAYS` PROCEDURAL BLOCK - 69 -
7.4.1 The Sensitivity List - 70 -
7.4.2 The Block Body - 70 -
7.4.3 Variable Assignment - 70 -
7.4.4 Statement Types - 71 -
7.4.5 Important Statement Considerations - 72 -
7.4.6 Equality Operators - 73 -
7.5 SOLVED EXAMPLE PROBLEMS - 73 -

7.6 SYSTEMVERILOG CONSIDERATIONS - 81 -
7.7 CHAPTER SUMMARY - 83 -
7.8 CHAPTER EXERCISES - 84 -

8 MULTIPLEXORS - 85 -

8.1 INTRODUCTION - 85 -
8.2 DIGITAL DESIGN FOUNDATION NOTATION: MUX - 85 -
8.3 GENERIC MODELING IN VERILOG - 88 -
8.3.1 Using Verilog parameters To Create Generic Models - 89 -
8.3.2 Module Parameters - 89 -
8.4 UNUSED INPUTS AND OUTPUTS IN MODULE INSTANTIATIONS - 92 -
8.5 CHAPTER SUMMARY - 94 -
8.6 CHAPTER EXERCISES - 95 -

9 COMPARATORS - 96 -

9.1 INTRODUCTION - 96 -
9.2 DIGITAL DESIGN FOUNDATION NOTATION: COMPARATOR - 96 -
9.3 VERILOG RELATIONAL OPERATORS - 96 -
9.4 GENERIC COMPARATOR MODELS - 99 -
9.5 WORKING WITH SIGNED NUMBERS IN VERILOG - 102 -
9.5.1 The Big Overview - 102 -
9.5.2 Operator Expansion Rules - 103 -
9.5.3 Verilog Signed and Unsigned Values - 103 -
9.5.4 Operations and the Signedness of Operands - 104 -
9.6 CHAPTER SUMMARY - 107 -
9.7 CHAPTER EXERCISES - 108 -

10 RIPPLE CARRY ADDERS - 109 -

10.1 INTRODUCTION - 109 -
10.2 THE RCA: UNDERLYING DETAILS - 109 -
10.3 DIGITAL DESIGN FOUNDATION MODEL - 109 -
10.4 VERILOG ARITHMETIC OPERATORS - 110 -
10.4.1 Using Mathematical Operators - 110 -
10.4.2 The Magic of Using Addition and Subtraction Operators - 111 -
10.4.3 Arithmetic Operator Issues - 111 -
10.5 CHAPTER SUMMARY - 117 -
10.6 CHAPTER EXERCISES - 118 -

11 D FLIP-FLOPS AND LATCHES - 119 -

11.1 INTRODUCTION - 119 -
11.2 MODELING SEQUENTIAL CIRCUITS - 119 -
11.2.1 Modeling Latches: An Asynchronous Circuit - 119 -
11.2.2 Modeling D Flip-flops: A Synchronous Circuits - 120 -
11.3 BLOCKING VS. NON-BLOCKING ASSIGNMENT STATEMENTS - 123 -
11.4 ALWAYS BLOCKS FOR SEQUENTIAL/COMBINATORIAL CIRCUITS - 124 -
11.5 SYSTEMS VERILOG CONSIDERATIONS - 124 -
11.5.1 SystemVerilog D Flip-Flop Model - 125 -
11.6 CHAPTER SUMMARY - 126 -
11.7 CHAPTER EXERCISES - 127 -

12 REGISTERS - 128 -

12.1 INTRODUCTION - 128 -
12.2 DIGITAL DESIGN FOUNDATION NOTATION: REGISTERS - 128 -

12.3 GENERIC REGISTER MODEL .... - 129 -
12.4 REGISTERS VS. REG-TYPE VARIABLES .... - 130 -
12.5 INLINE REGISTERS .... - 130 -
12.6 SYSTEMVERILOG CONSIDERATIONS .... - 132 -
12.7 CHAPTER SUMMARY .... - 133 -
12.8 CHAPTER EXERCISES .... - 134 -

13 FINITE STATE MACHINE MODELING .... - 135 -

13.1 INTRODUCTION .... - 135 -
13.2 FSM OVERVIEW .... - 135 -
13.2.1 State Diagrams .... - 135 -
13.3 FSM USING VERILOG BEHAVIORAL MODELING .... - 136 -
13.4 COMMON MISTAKES IN VERILOG FSM MODELING .... - 146 -
13.5 SYSTEMVERILOG CONSIDERATIONS .... - 146 -
13.6 CHAPTER SUMMARY .... - 149 -
13.7 CHAPTER EXERCISES .... - 150 -

14 COUNTERS .... - 152 -

14.1 INTRODUCTION .... - 152 -
14.2 DIGITAL DESIGN FOUNDATION NOTATION: COUNTERS .... - 152 -
14.3 GENERIC N-BIT UP/DOWN COUNTER MODEL .... - 153 -
14.4 CHAPTER SUMMARY .... - 157 -
14.5 CHAPTER EXERCISES .... - 158 -

15 SHIFT REGISTERS .... - 159 -

15.1 INTRODUCTION .... - 159 -
15.2 DIGITAL DESIGN FOUNDATION NOTATION: SHIFT REGISTER .... - 159 -
15.3 FINAL NOTES ON VERILOG SHIFT OPERATORS .... - 163 -
15.3.1 Shift Operators and Signedness of Numbers .... - 163 -
15.3.2 Additional Verilog Shift Operator Support. .... - 164 -
15.4 CHAPTER SUMMARY .... - 165 -
15.5 CHAPTER EXERCISES .... - 166 -

16 TESTBENCHES .... - 167 -

16.1 INTRODUCTION .... - 167 -
16.2 HARDWARE MODELING VS. HARDWARE VERIFICATION .... - 167 -
16.2.1 Types of Verification .... - 168 -
16.3 TESTBENCHES .... - 168 -
16.3.1 Testbench Support Constructs .... - 169 -
16.4 TESTBENCHES FOR CLOCKED CIRCUITS .... - 176 -
16.4.1 The `forever` Keyword .... - 176 -
16.5 SYSTEMVERILOG CONSIDERATIONS .... - 180 -
16.6 CHAPTER SUMMARY .... - 182 -
16.7 CHAPTER EXERCISES .... - 183 -

APPENDIX .... - 184 -

VERILOG CODING GUIDELINES .... - 185 -

Best Practices for Coding Verilog .... *- 185 -*
Naming Convention .... *- 185 -*
Code Format .... *- 185 -*
Commenting Your Models .... *- 186 -*
Model Coding Specifics .... *- 186 -*

Structural Modeling Specifics .... *- 186 -*
Simulation & Debugging .... *- 186 -*

VERILOG STYLE FILE .... - 187 -

CHEATSHEETS .... - 189 -

STRUCTURAL MODELING CHEATSHEET .... - 189 -
VERILOG BEHAVIORAL MODEL FINITE STATE MACHINE CHEATSHEET .... - 191 -
TESTBENCH CHEATSHEET .... - 192 -

DIGITAL DESIGN FOUNDATION MODULE TEMPLATES .... - 195 -

GENERIC DECODER .... - 195 -
2: 4 STANDARD DECODER .... - 195 -
MULTIPLEXOR (MUX) .... - 196 -
N-BIT COMPARATOR .... - 197 -
N-BIT RIPPLE CARRY ADDER (RCA) .... - 198 -
N-BIT REGISTER .... - 199 -
N-BIT UP/DOWN COUNTER WITH ASYNCHRONOUS CLEAR .... - 200 -
N-BIT UNIVERSAL SHIFT REGISTER .... - 201 -
FINITE STATE MACHINES (FSMS) .... - 201 -

INDEX .... - 202 -

Pretentions

Legal Stuff

FreeRange Verilog Foundation Modeling with SystemVerilog Support

Copyright © 2021 James Mealy.

Release: 4.00

Date: 08-12-2021

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

www.unconditionallearning.com

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

This book is licensed under the Creative Commons Attribution-ShareAlike Un-ported License, which permits unrestricted use, distribution, adaptation and re-production in any medium, provided the original work is properly cited. If you build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. To view a copy of this license, visit:

http://creativecommons.org/licenses/by-sa/3.0/

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

bmealy@calpoly.edu

Acknowledgements

Thanks to everyone that helped with this text. Here are their names:

Yeah, it's an empty list.

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.
- To introduce a few useful SystemVerilog concepts and language constructs.

The main purpose of this text is to give you the understanding and confidence to model digital circuits using the Verilog HDL. In truth, verifying the models you design is actually the larger and more important part of the process of digital circuit design. Despite circuit verification being “more important”, we choose to present only a small amount of time to the notion of circuit verification in this text. Digital circuit verification is simply a different subject, and is also a subject that grows as your circuit models become more complex. Additionally, circuit verification is an art form of sorts, because your circuit will become complex enough that you can’t possible test every input combination, which requires you to at some point in the testing declare your circuits as “working”. Digital circuit verification is an art form because you must be familiar enough with the Verilog HDL and the circuit you’re testing to declare with confidence that your circuit is “working” when you have not tested 100% of the circuit.

As many people know, when you read through a book on Verilog or digital design, the information is rarely presented in an easily usable format. Books on Verilog provide you with tons of information starting on page one, which is never 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 and/or advanced features of the language.

This book also serves as a companion manual to another text: ***FreeRange Digital Design Foundation Modeling***, or ***FRDDFM***, which is available free of charge from several places. This book presents digital design using a unique approach, which is comfortably different from approaches found in other digital design texts. I’ve include as much details as reasonably possible regarding FRDDFM, but you may want to obtain a copy of that text for the full story. Currently, you can find a version of FRDDFM at the following website: www.unconditionallearning.com.

The Verilog HDL is a subset of the SystemVerilog HDL. SystemVerilog is essentially a more modern version of Verilog, which has many added constructs that can be helpful when your intent is to focus on verification. You can only use most of the new features in SystemVerilog to verify your circuits are working properly, which means you can’t use these features for basic circuit design. Because our primary focus in this text is circuit design, we limit our dealings with most of the new features in SystemVerilog. We do mention a few SystemVerilog constructs in this text when pertinent, but we primarily discuss Verilog.

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

Verilog Coding Guidelines

Verilog Style File

Cheatsheets

- **Structural Modeling CheatSheet:** Contains one example of structural model for simple circuit
- **Finite State Machine Modeling CheatSheet:** Contains an example of Verilog model for simple FSM
- **Verilog Testbench CheatSheet:** Contains two example of simple Verilog Testbenches

Digital Design Foundation Module Templates: This part of the appendix provides templates for each of the various Digital Design Foundation Modules (we omit parity circuits). 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*

1 Introduction to Hardware Description Languages

1.1 Introduction

This chapter provides an overview of Hardware Description Languages (HDLs). Although this text primarily introduces 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. Recall here that a model is simply a description of a circuit. 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 is not a trivial process 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 that your original model synthesized correctly. We typically use HDLs to create models that verify our other models are working properly. We generally don’t intend to synthesize our models used for verification, a fact that underscores the difference between synthesis and verification.

This is the funny part. Most HDLs started out as a means to verify circuits. Not surprisingly, the HDLs have a significant amount of constructs that are not synthesizeable; the verification process makes good use of these constructs. 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 their non-synthesizability of the HDL code.

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 significantly different ways based on the notion that the given models have different intended purposes (namely, synthesis vs. non-synthesis). Many of the constructs in the HDL exist for verification only; those constructs are thus meaningless for the purpose of synthesis; in other words, it’s simply not possible to synthesize the typical verification model.

1.3 Programming Computers vs. Modeling Digital Circuits

If you’re reading this, you more likely 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. Keep in mind that your experience with writing computer programs is generally quite useful when your intent is to verify digital circuits.

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; this software typically includes an assembler and/or a compiler. The following is a brief overview of various levels of programming and issues associated with them. Yes, there is a point, so keep reading.

1.3.1.1 Assembly Code

This is the step above programming using machine code. Assembly languages are device-dependent, which means every computer probably has a different assembly language (a different instruction set architecture, or ISA). 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 much more syntactically structured compared to higher-level languages. As a result, the software that implements the assembler is relatively simple compared to the software associated with a compiler. If you're familiar with assembly code, you know that the assembly programs all rather 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 compared to assembly languages, those requirements are relatively loose.

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, and then generates a meaningful result from that structure. 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 the process magic? No… but it sure seems like it. Is the process 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.

1.3.2 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, your HDL code may look similar to higher-level language code, but appearance is its only similarity.

---

[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.

[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.

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 compilers have a 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.

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-based 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.

The important point to realize here is that when you hand off part of your design to some piece of software, the ultimate quality of your final product is constrained by the quality of software you're using. The more complex the software is (the more requirements of the software), the more likely the final product will have problems. This fact is important when using HDLs for design and verification because various software packages implement these processes. Note that the process of taking a page full of text (such as an HDL model) and converting that text to a working digital circuit is highly dependent upon various software packages. The unstated and always present goal of modeling digital circuits using an HDL is to do all you can to ensure the various software packages involved will generate the circuit you intended. While this sounds like a complex undertaking, it's actually not at beginning levels of HDL modeling. The two basic approaches this book uses to get you writing simple and understandable HDL models that don't requires a lot of knob tweaking and are relatively easily tested are these:

1) **Understand the basic tenets of HDLs**: Be aware of some basic and often non-intuitive aspects of modeling a digital circuit starting from a syntactically structured text-based description of that circuit.
2) **Keep your circuits as simple as possible**: The tendency for noobs is to write giant models that do a many things primarily because the HDL has similarities with higher-level programming languages. The better approach is to write many simple models that interface with each other, an approach that supports ensuring the synthesizer generates the circuit you intended and supports the verification process.

1.4 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 design is thus about increasing the efficiency of the digital design process by keeping designs abstracted away from low-level design. HDLs such as 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

[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.

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 repeatedly. 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 of 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. However, 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 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 you or anyone using the code in other designs.

1.5 Introducing Verilog vs. SystemVerilog

This text opts to introduce HDL concepts using primarily Verilog. The question always comes up as to why a text such as this does not introduce SystemVerilog rather than Verilog. It is generally people who are not familiar with any HDL who ask this question, such as students in a digital design course, but I've have other instructors ask me this question as well. This section covers my justifications for primarily introducing Verilog rather than SystemVerilog.

1.5.1 Verilog's Relation to SystemVerilog

SystemVerilog is essentially a newer and updated version of Verilog. Accordingly, Verilog is thus a subset of SystemVerilog, which means that everything you learn in Verilog transfers directly to SystemVerilog. It's not surprising that people panic when someone suddenly asks them to model circuits using SystemVerilog rather than Verilog, as these people don't understand that the transition is seamless. For example, if you write a file using Verilog and then save the model as a SystemVerilog file, it's going to work perfectly. Here are the official reasons why I introduce Verilog rather than SystemVerilog.

- If you're reading this book, then you probably are new to Verilog and/or modeling digital circuits using an HDL. This being the case, you need to learn the basics of both HDL modeling and the Verilog modeling. In this beginning stage, there are bunches of new features in SystemVerilog that you won't initially use. Recall that the goal of this text is to get you up and running fast; presenting details you won't use will slow down that process. A bad analogy would be leaning to fly an airplane: you can learn in a two-seater prop plane (Verilog) or you can learn in a Boeing 747 Jumbo jet (SystemVerilog); the choice seems simple to me.
- This text is primarily associated with using Verilog to model digital circuits. It's a given that once you model a circuit that you'll need to verify that your circuit models actually work, which you typically do using the same Verilog language as you did to model the circuit. Digital circuit design and digital circuit verification are two significantly different processes despite the fact you use the same language to do them. Note that all most of the features added to SystemVerilog involve verification of circuit, which makes it seemingly pointless to go with SystemVerilog. Once again, this text is not trying to tell you everything there is to know about Verilog, it's only trying to get you up and going quickly.
- If you tell students in an introductory course that they are using SystemVerilog, they will put that notion on their resume. When a potential interviewer sees that on their resume, the interviewer will ask them a technical question about SystemVerilog, which means the question the interviewer asks could be about parts of SystemVerilog that students probably did not use, namely constructs used exclusively for verification (as opposed to synthesis). You probably don't want to put yourself in this position.

1.5.2 The Word on Verilog vs. SystemVerilog

This text does introduce a few SystemVerilog concepts. Some SystemVerilog constructs can be quite handy when modeling digital circuits that you intend on synthesizing. Other SystemVerilog constructs are handy to assist you in making your code generic. Even other constructs can simplify the minor amount of verification we do in this text by enabling the tools to list signals as “labels” as opposed to numerical value. As you will soon find out, it’s a bummer to be examining a screen full of 1’s and 0’s; it’s much more visually pleasing to use SystemVerilog constructs such that the signal appear as alpha values rather than numbers. No need to worry about the details here; they will make more sense when we describe them in later chapters.

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 your models are relatively simple. Simple in HDL modeling means your models use the simplest possible set of knobs. Complex designs require more knobs; more knobs give the synthesizer more options, which lowers the probability the synthesizer will generate a circuit the works the way your model intended it to.
- We categorize good digital circuit models as boxes exchanging information with other boxes; these individual boxes make no attempt to do everything required by the circuit but instead work together to implement a circuit the complete system. HDL syntax is powerful enough to allow you to describe large complex circuits with a single box, but that design approach gives the synthesizer the freedom to generate a circuit different from the one you were intending to model. 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; instead know how the synthesizer does things and work with the strengths of the synthesizer and avoid its weaknesses.
- 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 designing appears on paper before you start modeling with HDL constructs. In other words, use HDLs to implement designs as opposed to using HDLs as the main design tool.

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

Justifications:

- The synthesizer follows a set of rules in order to do what it needs to do, which sometimes can lead to the synthesizer generating a circuit that does something you did not intend. Additionally, we're all human, and we're going to make unintended mistakes as we create our models. 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 your design will be to test.

Golden Rule #4: Neatness counts in HDL models.

Justifications:

- Part of testing 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 you poorly formatted your code. 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 The Final Word

This is only a figurative title for this section. You're about to embark on you Verilog-based HDL journey, with the goal of being able to model digital circuit. The strange issue that comes up is that we're using a text-based language to model digital circuit. Moreover, we most often abstract relatively far away from the logic that out models later generate resulting from the synthesis process. The Verilog HDL itself has many similarities to higher-level programming languages (namely C), so have a tendency to forget our original goal is do model digital circuits.

Your unstated goal as a digital circuit modeler is to never forget that when all is said in done, you're still modeling digital circuits. The text-based HDL may seem strange, but you'll be using it to generate a relatively small set of the digital modules associated with digital design. The more you rely on your digital design skills and the less you rely on the "magic tools" to do the work for you, the better off you'll be in terms of creating efficient and easily verifiable models.

1.8 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.
- SystemVerilog is a superset of Verilog. SystemVerilog is “more modern” with the addition of many software-like constructs that are not synthesizable and are used specifically for verification.
- Verilog is the preferred way to learn both digital circuit-modeling concepts because the language is essentially less feature-packed than SystemVerilog. Everything anyone learns in Verilog transfers directly to SystemVerilog.
- 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.

1.9 Chapter Exercises

1) List the two most commonly used HDLs.
2) List and briefly describe the two main purposes of HDLs.
3) List the three levels of computer programming.
4) What is the final output of all programming languages?
5) In your own words, describe the purpose of the HDL synthesizer.
6) In your own words, describe the meaning and purpose of "knobs" in HDL modeling.
7) In your own words, describe the two approaches this text uses to getting you to write solid HDL models.
8) List and briefly describe the two main tenets of digital design.
9) Briefly describe what we mean by the term *flat design.*
10) Briefly list the main difference between Verilog and SystemVerilog.
11) Briefly list why this text primarily introduces Verilog and not SystemVerilog.
12) List and briefly describe the four Golden Rules of using HDL to model circuits.
13) Briefly describe the purpose of a style file.

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 FreeRange Digital Design Foundation Modeling (FRDDFM), which is how we refer to our approach to teaching basic digital logic and 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 our modern approach to digital design.

This text is also serves as a supplement to learning the basic concepts of digital design. This being the case, you skip reading this chapter, particularly if you're using the FRDDRM text. Honestly, I lifted most of the information in this chapter from the FRDDRM text. I'm not sure if it's possible to plagiarize yourself, but that's what I'm doing. Send the writing police over to my place if this causes you any discomfort.

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 2.1 shows 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.

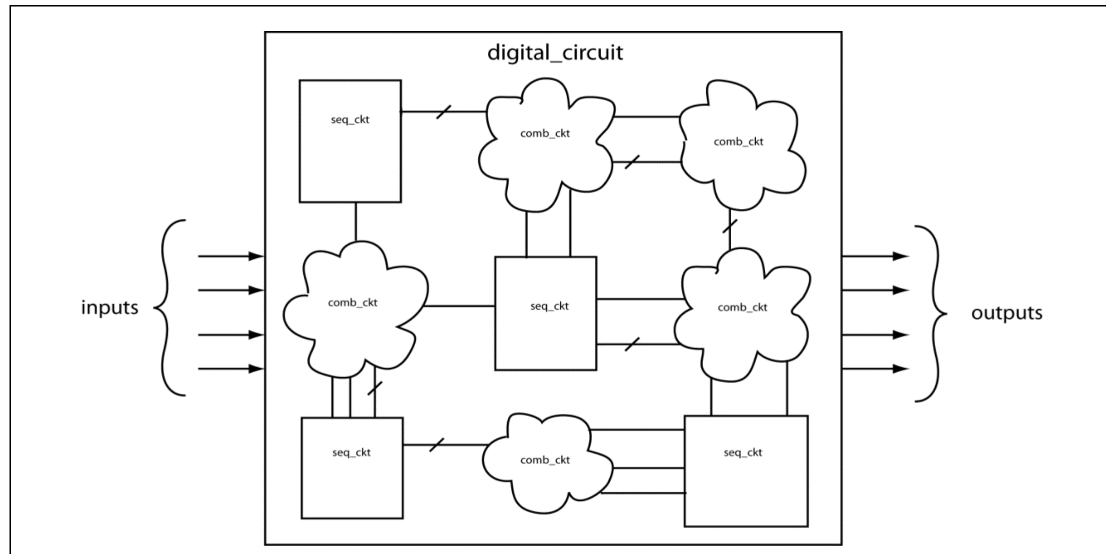
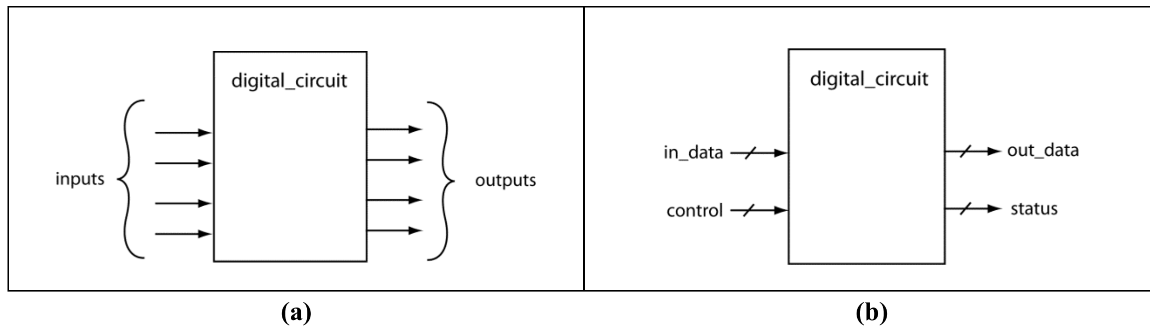


Figure 2.1: A basic logic circuit.

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.

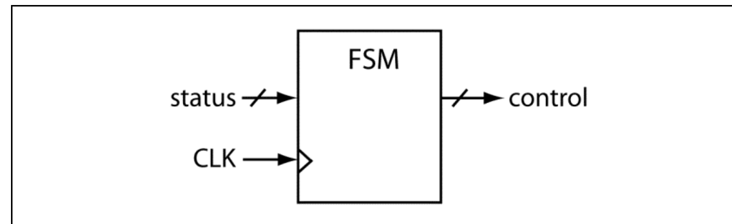


Figure 2.3: A black box model of an FSM

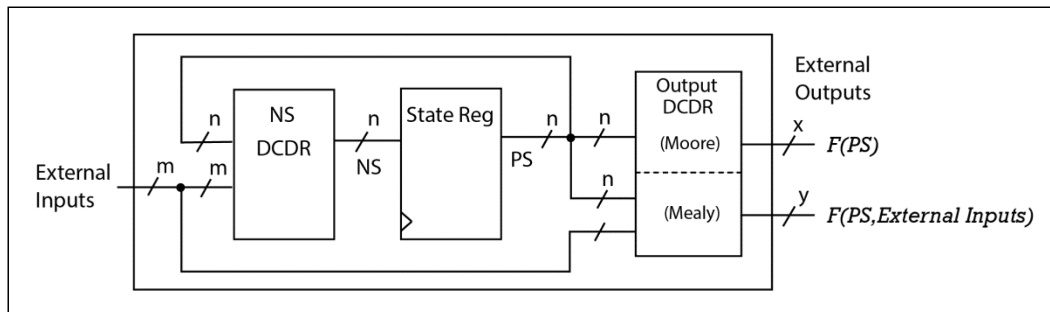


Figure 2.4: The FSM model showing the two types of outputs (Mealy and Moore).

Figure 2.5 shows a modified version of Figure 2.1 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.

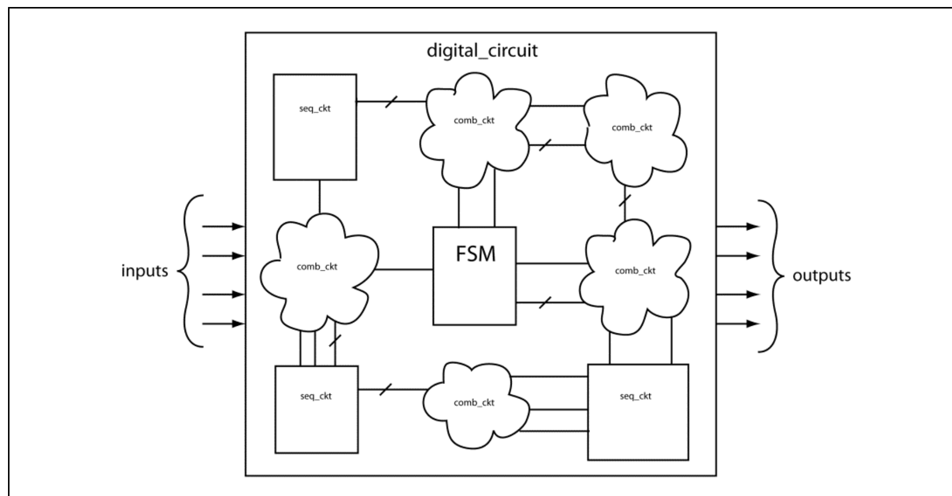


Figure 2.5: A basic logic circuit controlled by an FSM.

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***[1] 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 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

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

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.

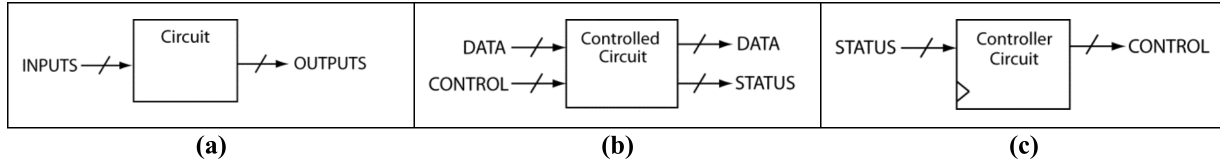


Figure 2.6: Old digital circuit model (a); models for controlled (b) and controller circuits (c).

The FRDDFM 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. 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.

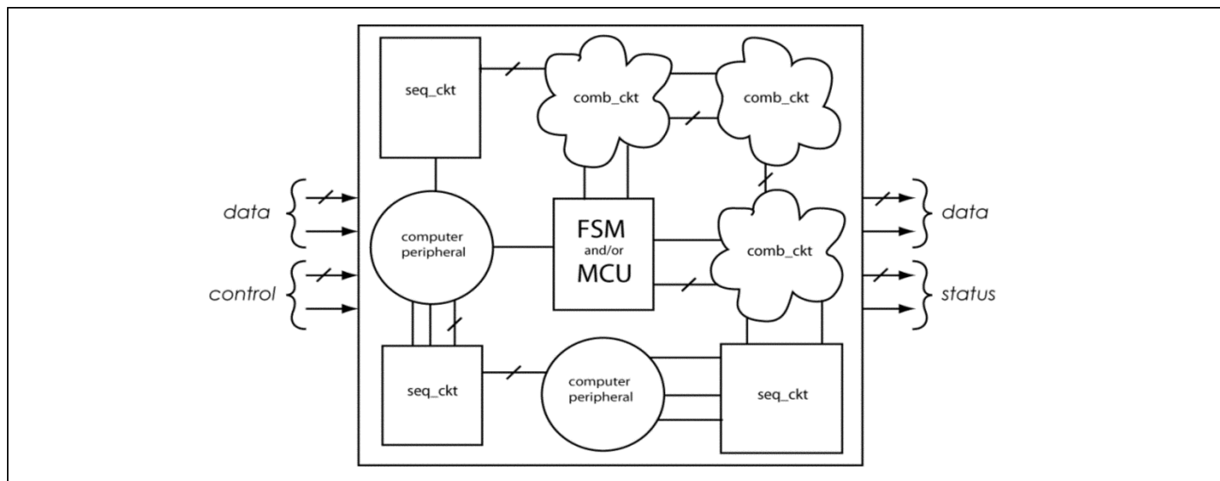


Figure 2.7: Our unifying digital circuit model.

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**

2.5 Chapter Exercises

1) Briefly describe whether it is possible to plagiarize yourself. If this is possible, what entity would be ethically and morally responsible for making such a determination?
2) Briefly describe what is meant by the term “digital design”.
3) List and briefly describe the two main tenets of digital design.
4) List and briefly describe the two main types of digital circuits.
5) List and briefly describe the four major classifications of inputs and outputs of a digital circuit.
6) Briefly describe the main purpose of an FSM in digital design.
7) List and briefly describe the three main components of an FSM.
8) List and briefly describe the two main classifications of FSM outputs.
9) List and briefly describe the three main types of approaches to digital design.
10) Briefly explain the concept of structured design as it relates to digital design.
11) List and briefly describe the four approaches to controlling a digital circuit.
12) Digital design foundation modeling divides circuits into two categories; list and briefly describe those categories.
13) Control signals are typically output from one circuit and input to another circuit. List what types of circuits they are output from and what type of circuits that are input to.
14) Status signals are typically output from one circuit and input to another circuit. List what types of circuits they are output from and what type of circuits that are input to.

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 modeling by presenting some of the higher-level HDL concepts related to both Verilog and basic digital design. The intent of this chapter is to provide you with a basic feel for modeling digital circuits using Verilog; we'll dive into actual modeling concepts in a later chapter.

3.2 History

Engineers created Verilog sometime in the early 1980s as a way to standardize the process of simulating digital circuits. Verilog started out as a method to simulate and verify digital circuits, but soon after that Verilog became a tool used to synthesize digital circuits. Although Verilog initially was a proprietary language, the language entered the public domain in 1990. The continued development of Verilog eventually led to its adoption 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. SystemVerilog is the latest and greatest thing out there; SystemVerilog is a superset of Verilog and was design to support an extensive list of software-like features primarily designed for circuit verification.

3.3 The HDL Governing 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. Recall that we use the term "model" in engineering to mean some type of representation of something. In the case of HDLs, the Verilog file contains the text-based description of the circuit we are modeling[1]. Note that we often refer to the Verilog model as the "Verilog code"[2]. 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 software package we refer to as the "synthesizer"; the final result somewhere down the line is 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 themselves digital circuits that are all operating in parallel (simultaneously). This means that the order that you list the boxes in your HDL model does not matter because the synthesizer will interpret them as separate modules operating concurrently. 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[3], which means you can place the code that models these boxes in any order in the Verilog code and the model will be functionally equivalent. 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 (processors acting in parallel) is a pain in the arse in comparison to design digital circuit modules that act in parallel.

---

[1] Note that the "D" in the HDL acronym stands for "description".

[2] This code is similar to the "code" of a computer program in that both sets of code are text files, but HDL code is inherently different from code for computer programs.

[3] It is helpful for human readers of your models to arrange the Verilog code in the file such that it the order is something other than random.

To be perfectly clear, the order of the appearance of the boxes in Verilog models 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. Recall that 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. While we'll wait until a later chapter to get into those details, you must keep in mind that knowing how to describe combinatorial and sequential circuits using HDLs such as Verilog is massively important.

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 Dataflow Descriptions (Dataflow Models)

The term "dataflow descriptions" is a common term used in HDLs that describes one of the two basic modeling approaches that HDLs use. We use the term "dataflow descriptions" to refer to Verilog models that are comprised of low-level descriptions of circuits, which generally refer to 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 at the gate-level.

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 (Behavioral Models)

The main point behind behavioral models of digital circuits is that you are no longer constrained to describing circuit operation at the gate-level. As you will see, the awesome power of HDLs lie in behavioral descriptions of circuits. The various constructs in HDLs provide you ways to describe how a circuit should "behave" rather than describing the gate-level circuit that will make that behavior actually happen. 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 dataflow model; in fact, I would not want to describe in any manner and much prefer leaving the details to the synthesis tools. .

Most 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 when you don't know how to work with these knobs. .

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.

3.4.3 Using Dataflow or Behavioral Descriptions

The fast summary: there are dataflow and behavioral descriptions, and there are sequential and combinatorial circuits: what are you going to use to model your circuit? The problem with the question is that it lacks a usable context. It would be nice to have a set of rules that you can use so you can mindlessly design circuits, but if there were such rules, no one would be paying you the big bucks to use and HDL to model digital circuits. The bad news is that there are no such high-level rules. The good news is that there are a few low-level rules you can (and should) follow so that you can successfully model digital circuits using Verilog without a ton of effort; we'll address those rules in the later chapters of this text.

3.5 Verilog Invariants

Some characteristics about modeling digital circuits with Verilog never change. We list these items before we discuss the modeling abilities of Verilog because these items are simple and 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 need 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, and you'll be immediately more popular at parties.

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 numerical value

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 "ADDEND" 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 from 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)[4]. Here are a few of the more obvious uses of white space in your code

- Your Verilog code must follow accepted indentation practices. Even though the synthesizer does not care, proper indentation in Verilog code transfers information to the human reader by making you Verilog models more readable. For this issue, check out the style-file used by your instructor or the more experienced members of your team.
- Use blank lines to separate different "ideas" such as modules in the vertical direction
- Use spaces to separate different components of the same idea in the horizontal direction. This generally means that if items in your source code have similarities, it is usually best to align them to enhance the readability of your code.

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[5]. Verilog has two types of comment; you may notice these are the same comments associated with the C programming language.

- 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 precedence rules, 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 operator precedence rules of the language.

3.5.2 Statement Termination

You must terminate all statements in Verilog with a semicolon. This is easy to state, but harder to put into practice when you're first learning Verilog because it's not always easy to initially know what if a construct is an actual statement or not. 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 reserved words, or 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

[4] 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.

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

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; consult Dr. Internet for the full list of 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 Verilog 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 SystemVerilog, 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.

3.7 Chapter Exercises

1) Briefly describe the original purpose of the Verilog HDL.
2) What is the main theme behind using an HDL to model digital circuits?
3) In your own words, briefly describe the notion of concurrency in a digital circuit.
4) Briefly describe the difference, if any, between the words concurrent and parallel in the context of a digital circuit.
5) Briefly describe whether computer programs operate in a parallel manner or not.
6) Briefly and in your own words, describe what we mean in by the term "model" in engineering.
7) Briefly explain whether the terms "Verilog code" and "Verilog models" are synonymous.
8) Briefly explain why using dataflow models when you can give the synthesizer fewer options.
9) Briefly explain why using behavioral models is a more powerful modeling approach compared to using dataflow models.
10) Briefly explain whether you can use numerical values in identifiers.
11) List and briefly describe the two types of comments we use in Verilog.
12) Briefly describe why the Verilog synthesizer does not care if your code is not readable by humans.
13) Briefly describe why you should never use the tab key when writing Verilog code in an editor.
14) Briefly describe a major way it is possible to get by without knowing the operator precedence rules in Verilog.
15) Briefly describe the two types of comments used in Verilog and the Verilog syntax that supports them.
16) Comments are information passed between two entities: which two entities are these?
17) Briefly describe how statements are terminated in Verilog.

4 Dataflow 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 or with dataflow 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 use the term "dataflow modeling" so as not confuse it with the raw power of behavioral modeling[1], 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 all 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.

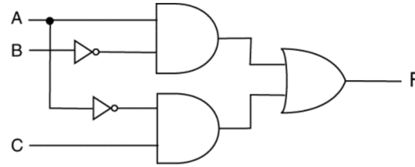
> ***External Interface:*** The external interface comprises of the signals the outside world knows about in regards it a given module, which as essentially the inputs and outputs to that module. We often refer to the external interface as simply the module's "interface". There are three types of signals 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 signals of type **`input`** and **`output`**. The external interface is analogous to the formal parameters (value sent to and returned from functions) in higher-level computer programming languages.
>
> ***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 modeling, and structural modeling. The internal circuitry is analogous to functions called by code in higher-level programming languages. This chapter describes dataflow modeling.
>
> ***Internal Interface***: Verilog uses internal signals as nodes, which means they are connections between the various modules (boxes) within a given module. Unlike external signals, the outside world (modules external to the given module) does not know any details about the existence of the internal interface. The internal signals are analogous to the local variables in function definitions in a higher-level programming language. 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.

At this point, we're ready to dive in and start using Verilog to design digital circuits. We'll make this dive using enlightening examples.

[1] 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 you'll recognize the term "dataflow".

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 one solution for this example. This is our first example of a Verilog model, so we have many things to say about it; but have no worries, we'll have less to say about future examples:

- We use a boldface font for the non-operator Verilog reserved words for convenience; your text editor may not support such a feature.
- The model contains a few nicely placed comments. The identifiers are rather wimpy, which is OK for simple examples such as this one.
- When you're 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. We later switch to another style in order to save space on the page, which we do at the expense of clarity of the model.
- 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. The input and output specifications represent the interface to this circuit as the outside world sees it. Note that we included a separate **`input`** line in the code for each input, which is the best modeling process; we could have include only one **`input`** and comma separated the three individual inputs.
- 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 each time an input changes. What this means for this circuit is that *anytime a signal on the right side of the equation changes, the equation is re-evaluated*. This enables the output to change anytime there is a change in any of the inputs. Keep in mind that this interpretation is by definition a function of the Verilog language, and is thus arbitrary, which makes this notion one of those items you simply need to accept when working in Verilog.
- The one continuous assignment statement forms 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[2]. 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

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

equations). The equation includes parenthesis around the AND operations, just to be sure. Yep, I probably should know the precedence rules[3]. But then again, using parenthesis allows me to get by without know the precedence rules, and has the added benefit of making equations such as the one in Figure 4.1 makes it easier for humans to read and thus understand.

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

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

endmodule
```

Figure 4.1: Solution for this example.

Figure 4.2 shows another solution to this example. We include this solution because we consider it an alternative form of the first solution. One thing you may start to note is Verilog's similarity with the C programming language. In this case, modules in Verilog are similar to function definitions in C, namely there are two different ways to specify the formal parameters of the function in the function definition. The model in Figure 4.2 places the interface labels inside the parenthesis rather than outside the parenthesis as in Figure 4.1. The approach in Figure 4.2 is the preferred solution.

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

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

endmodule
```

Figure 4.2: An alternative solution for this example.

4.3 Some Verilog Details

Now that you've seen an actual Verilog model that implements a simple circuit, it's time to toss in some more pertinent 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 4.1 shows Verilog's bitwise logic operators with examples. The previous example used three of these operators on 1-bit wide signals.

[3] 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***.

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**
`^`	binary	`X ~^ Y`	exclusive OR each bit of **X** with each bit of **Y**

Table 4.1: Bitwise logic operators in Verilog.

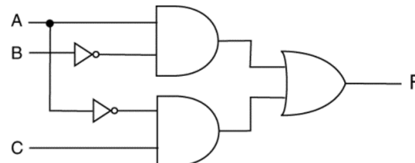
4.3.2 Verilog Nets and Variables

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 with each other. The Verilog **module** definition uses input and output specifications to define the module's external 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, or the internal 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**[4].

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 know about (they are not part of the external interface). What we need to do is decompose the continuous assignment statement in the previous example to one assignment per gate in this example. When we do it this way, we need to utilize internal interface signal, which we do in the solution below. Figure 4.3 shows one possible solution for this example. Here is the full smear:

- We use the net data type **wire** to declare the internal interface signal. The design requires two wire types: one for each AND gate output. The outputs of the inverters are also internal signals, but we opted to represent those signals in the using the inversion operator ("~").
- The internal interface signals are not part of the module's interface (inputs & outputs).

[4] As it turns out, when you specify inputs and output in the external interface of a module, if you do not explicitly state the a net type, the Verilog compiler assumes that inputs and outputs are **wire** data types. This will become an issue when we later use **reg** types in our models.

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;
   wire Y;

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

endmodule
```

Figure 4.3: Solution for Example 4.2

4.3.3 Summary of Verilog `wire` Nets

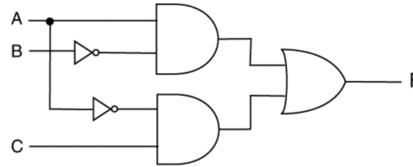
As you can see, we are now straddling two worlds here: modeling a digital circuit, and having the ability for some entity to use that model to synthesize actual digital hardware. Our first abstraction is the notion of a **`wire`**. As opposed to other entities, wires have a non-misleading name[5], as it is easy to think of a wire type as an actual physical wire in a circuit that connects an input to an output (or vice versa). Here are some other items to keep in mind when using **`wire`**s in your models:

- You use wires for connecting different elements, namely inputs to outputs. Because of this, you can think of wire as physical wires in a circuit, which somewhat mitigates the notion of abstracting a physical circuit from a text-based description.
- Your models can read or assign wires, which means they can serve as both inputs in equations as well as outputs. This is because unlike the external interface, we do not declare internal signals as inputs or outputs.
- **`wire`**s serving as outputs need to be driven by continuous assign statements or from an output port of a module (external interface).
- **`wire`**s do not have the ability to store data, which means you can't use a wire to represent a sequential element (which is a topic for another chapter).

[5] As you will see in a lather chapter, there is also a reg type, which turns out to be a misleading name.

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 4.4 shows the solution to this example. Notice that the model is 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 **wires**. While this implicit definition of wires seems somewhat useful, it's actually quite limited because the implicit definitions an only be one-bit wide. We have not yet mentioned signals wider than one bit, but most design have signals that are more than one-bit wide (vectors) rather than only one-bit wide (scalars).

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 mastering the Verilog language, or if you're trying confuse people. There are ways to force the Verilog compiler to not use implicit declarations, but we'll not cover that here. In the end, allowing implicit definitions typically cause problems based on designers forgetting to declare a vector signal and having the Verilog compiler implicitly declare a scalar signal instead.

```
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 4.4: 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 compiler uses the index range to determine the number of signals in

the bundle that you're declaring. 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 4.5 provides a few generously commented examples of vector declarations and bit access to vectors.

Note that the declarations are quite varied; it's best to keep your definitions as simple and as consistent throughout your design as possible. Keeping track of the LSBs and MSBs in designs can be troubling if when your declaration style is not consistent.

```
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 4.5: 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 the input is greater than or equal to 12, 2) when the input is less than 8, and 3) when the input 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 4.6 shows the BBD for this circuit.

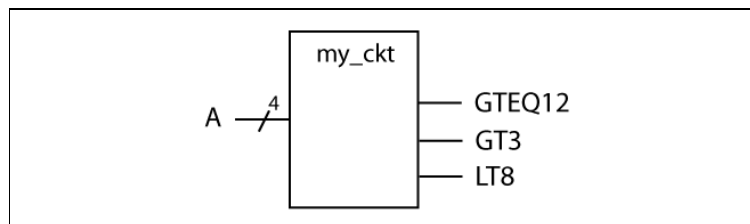


Figure 4.6: The black box diagram for circuit in this example.

Our next task is to generate the Verilog code that properly models the circuit solution. Figure 4.7 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. A better practice would be to make the model name more self-commenting by making the name more descriptive as to the circuit's functionality.

- The **A** signal is a 4-bit vector, as we indicate in the external interface declaration with the use of the bracket notation. Because we don't explicitly specify a type for **A**, it generally defaults to **`wire`** type.
- 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]).
- There are other ways to model this circuit, namely there are relational operators in Verilog that you can use. The point behind this example was to demonstrate how to access individual bits within a vector. We'll introduce relational operators in a later chapter.
- We make the continuous assign statements more readable by inserting a blank line between them.
- The circuit requires no internal interface (intermediate signals)

```
module my_ckt(
   input [3:0] A,
   output GTEQ12,
   output GT3,
   output LT8   );

   // internal circuity

   // both two MSBs are set
   assign GTEQ12 = A[3] & A[2];

   // either of two MSB are set
   assign GT3 = A[3] | A[2];

   // the MSB is zero
   assign LT8 = ~A[3];

endmodule
```

Figure 4.7: The solution for Example 4.4.

4.5 SystemVerilog Considerations

We can replace the notion of a **`wire`** in Verilog by the **`logic`** data-type in SystemVerilog. Designers included the notion of the logic data-type in SystemVerilog to remove some a misleading type name in Verilog, namely the **`reg`**-type (covered in an upcoming chapter). There are some other issues involved that we'll not address here, but in short, you can use a logic type anywhere you would normally declare a **`wire`**.

I strongly suggest using the **`wire`**-type for now when you can, and then using reg-types in other instances. I feel this is a better approach to learning Verilog based on the usage of **`wire`** and **`reg`**-types. We'll say more on this subject later once you have more experience using Verilog.

4.6 Chapter Summary

- We use the *Dataflow Modeling* to differentiate between official Verilog "behavioral modeling" done at a low level and on the true behavioral level (at a high level). 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, a topic we deal with in a later chapter.
- The basic Verilog model structure contains three items:
  1) External Interface: The inputs and outputs to the circuit that the circuitry outside of the models sees
  2) Internal Interface: signals internal to the model that the model uses to connect internal circuitry; circuitry external to the model does not see these signals.
  3) Internal Circuitry: sub-modules or blocks within the model that model the module's functionality; these block are connected to each other via the internal interface signals.
- 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.

4.7 Chapter Exercises

1) Briefly describe the two levels you can use in Verilog to describe a digital circuit.
2) Briefly describe whether the so-called "dataflow model" is actually behavioral modeling or not.
3) Name and briefly describe the three parts of a Verilog model.
4) Write the external interface portion of a Verilog model that describe the following block diagrams:

(a) (b) (c)

(d) (e)

(f) (g)

5) Provide black box diagrams that are defined by the following Verilog external interface specifications.

```
module ckt_a  (
     input   in_a,
     input   in_b,
     input   in_c,
     output  out_f );
```

(a)

```
module ckt_b  (
   input   LDA,
   input   LDB,
   input   ENA,
   input   ENB,
   input   CTRLA,
   input   CTRLB,
   output  OUTA,
   output  OUTB  );
```

(b)

```
module ckt_c  (
   input   [7:0] bun_a,
   input   [7:0] bun_b,
   input   [7:0] bun_c,
   input   lda,
   input   ldb,
   input   ldc,
   output  [7:0] reg_a,
   output  [7:0] reg_b,
   output  [7:0] reg_c);
```

(c)

```
module ckt_d  (
   input   [31:0] big_bunny,
   input   [31:0] big_wabbit,
   input   [1:0]  mx,
   output   byte_out  );
```

(d)

```
module ckt_e  (
   input   RAM_CS,
   input   RAM_WE,
   input   RAM_OE,
   input   [3:0]  SEL_OP1,
   input   [3:0] SEL_OP2,
   input   [7:0] RAM_DATA_IN,
   input   [9:0] RAM_ADDR_IN,
   output  [7:0] RAM_DATA_OUT  );
```

(e)

```
module ckt_ee_dept  (
   input  rss_bytes,
   input  rss_sux,
   input  rss_dogface,
   input  [23:0]  worthless,
   input  [23:0]  way_bad,
   input  [23:0]  go_away,
   input  [31:0]  big_joke,
   input  [31:0]  insecure,
   input  [31:0]  lazy,
   output [32:0]  SMD );
```

(f)

6) Provide Verilog models that implement the following Boolean expressions.

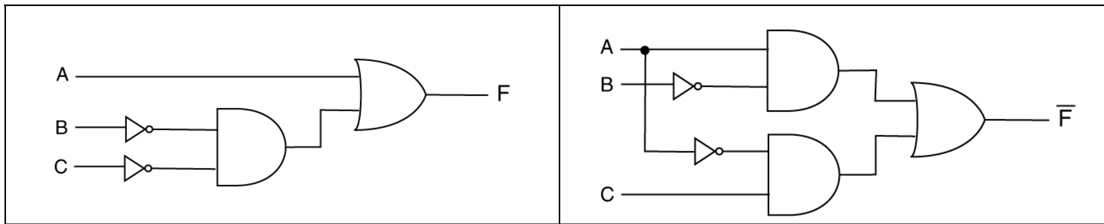
(a) $F(A,B,C) = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + A\cdot B\cdot\overline{C}$

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

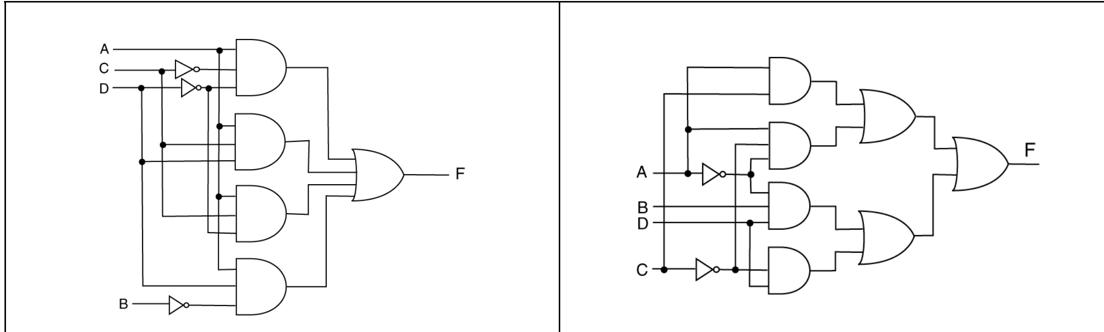
(c) $F(A,B,C) = (A+B+C)\cdot(A+\overline{B}+C)\cdot(A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)$

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

7) Provide Verilog models that implement the following circuit models



(a) (b)



(c) (d)

8) Provide a Verilog model that implements a half adder (HA).

9) Provide a Verilog model that implements a full adder (FA).

10) Write an equivalent equation for F in following Verilog model and draw the equivalent circuit.

```
module ex1_model  (
    input  A,
    input  B,
    input  C,
    output F   );


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

endmodule
```

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, which you can place in other boxes.

5.2 Modeling: Before We Start

We are embarking on using Verilog to model digital circuits. We of course base all this modeling on the notion that a model is nothing more than a description of something. Up to this point, we've used HDL-based dataflow models to model circuits, and also mentioned that we can use behavioral models to implement circuits (though this is a topic for another chapter). Now we are starting to talk about structural models. At this point, you may be confused.

To be clear, structural modeling is somewhat different form behavioral and dataflow models. It's somewhat hard to explain in words, but you it will start to make more sense when you see it in actual use, but here we go. You use dataflow and behavioral models to directly define how a circuit should operate at a base level[1]. In other words, if you examined the code associated with a dataflow or behavioral model, you would be able to know exactly how the circuits operate. This definition essentially means we are creating modules that we can call "boxes" and later use those boxes in other designs without having to first redefine the model. The definition of structural modeling means a design that includes the "inclusion" of a separately designed box (and defined) box rather than being a model that includes the full description of all the circuitry contained in that box (whether it be dataflow modeling or structural modeling, or some combination of both).

Our mission here is to model digital circuits efficiently. If we're redesigning the wheel in our designs, than we are not efficient designers. Most often in HDL modeling we're incorporating previously designed modules in our designs, which we don't necessarily need to do, but when we do so, we are officially using structural modeling. In the end, we can create a model that uses all these modeling techniques: dataflow, behavioral, and structural modeling. At this point, it's hard to make a statement about what type of modeling we're then actually using, but the best I can say is that if you use structural modeling in your design, than the most appropriate thing is to say it's a structural model. The truth is, at some point, no one cares what type of modeling your design uses; they only care that your design works, which includes working in an efficient manner. Attempting to attach a label to your design approach is a sign that you're new to HDL modeling, which is no big deal, but at some point realize that we don't need such labels for working designs.

5.3 Structural Modeling Syntax

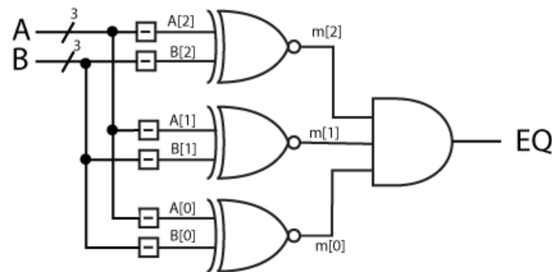
If your design is not some type of structural model, chances are that you will later use that design as a component in one of your later structural models. Structural modeling is one of the items that provides great modeling power to HDLs such as Verilog. The good news is that there is no new modeling concepts associated with structural models in terms of the languages ability to model circuits. The bad news is that you must learn some 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.

[1] We're trying hard not to say "low level" here.

Keep in mind, digital designs are inherently hierarchical, which means digital circuits are a bunch of boxes inside of boxes interfacing with other boxes. This means that when you are designing a digital circuit, you always doing one of two things: designing new boxes or using previously designed boxes in your design. This chapter is about structural modeling, which is thus about using previously designed boxes in your design. Since we don't yet have any meaningful previously design boxes, we'll make our own simple boxes in the examples in this chapter.

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 in order 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 those details later. What you need to realize is that you can consider the gates in this design as black boxes; we use gates because we have not discussed models that are more complex at this point.

The circuit is a 3-bit comparator, which we typically model using bundle notion on the two 3-bit inputs. In short, the EQ output is a '1' when the two inputs (A, B) are equivalent. Although our main goal for this example is to use structural modeling to describe the comparator, we also use vector notion to represent the input signals. Because we use the vector notation, we need a way to show how to access the signals within the 3-bit bundle in order to use them as 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[2]. We then name the signal to support our HDL model, but we generally do not include that level of detail in a BBD. The signal-naming notation in this example purposely uses Verilog syntax (namely the square brackets).

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 the top-level module, we define them as separate modules because we want to use them as modules in the top-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; for example A[2], A[1], and A[0].

[2] We don't use the bus width notation on lines that are 1-bit wide.

- 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 single bundle. We could have declared them as three individual **`wire`**s, but using bundle notation is usually clearer.
- This model uses two external modules: the XNOR gates and the AND gates. We have declared and defined these as part of this solution, but we could have defined them 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.
- Each instantiation has a unique label, which is the sole item that differentiates them from other instantiations of the same module. We always make these labels as self-commenting as possible.
- The AND and XNOR gate models don't need to be in the same file. We typically call the module that contains the instantiations as the *top-level module*. The top-level module simply needs to be able to "locate" the instantiated modules in your design, which is of function of the development software you are using and not worthy of comment here.
- There are many approaches to instantiating modules; this model 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[3]. There is actually yet another way to do this, but I feel it's confusing and stupid, and I truly can't recall exactly what it is. The way I instantiated the modules in this design is the best approach.
- 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.
- The lower-level modules should not have any idea of how they will be used on a higher level by some module that instantiates them. This being the case, do not attempt to use self-commenting lower-level signal names that indicate how you will use the signals at a higher level. Keeping things generic is always the best approach.
- 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 and placement of comments.
- The commented code makes the model more understandable for humans and extraterrestrials. .

[3] 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 (    //- definition of XNOR gate
   input  A,B,              //- combined on one line to save space
   output F  );

   assign F = ~(A ^ B);
endmodule

module my_and (     //- 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 (
    //- external interface signals
    input [2:0] A ,B,
    output EQ  );

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

   //- internal circuitry --------------
   //- XNOR instantiation
   my_xnor XNOR2 (     // XNOR2 is an arbitrary label
      .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.

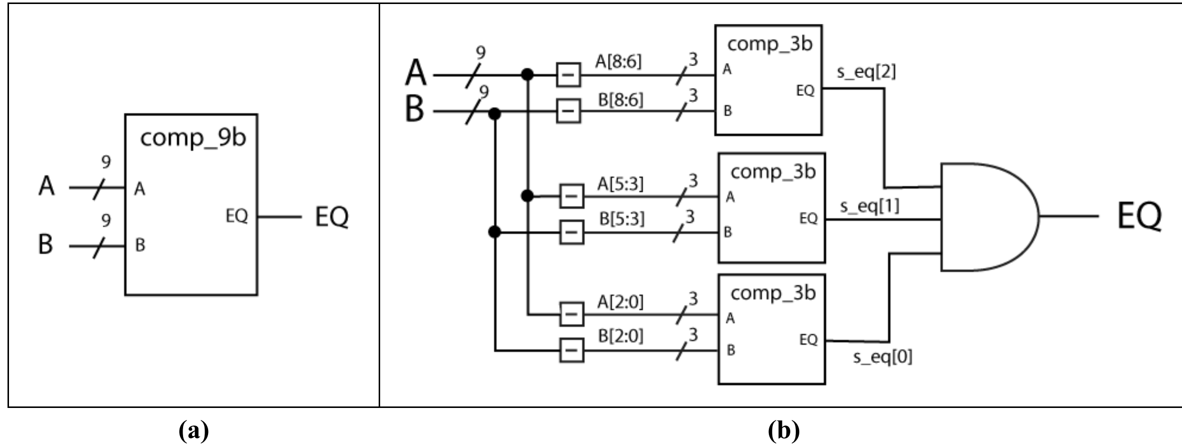
Example 5.2: Component-Based 9-Bit Comparator

Use structural modeling and three 3-bit comparators to model a 9-bit comparator. Use the 3-bit comparator from Example 5.1; 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 where ‘1’ indicates the inputs are equal. 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 being the optimal approach to modeling comparators.

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 of that module in this 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 instances is asserted. This is a verbal description of the circuit in Figure 5.2(b); note that 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 we commented out. We include this because it provides an example of a *reduction operator*, which can be extremely useful in some cases. Section 5.4 provides a full explanation of Verilog's reduction 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.**

5.4 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 genericity in 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 fragment of Verilog code.

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.

5.5 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 will be 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.

5.5.1 Structural Model's Relation to Higher-Level Computer Programming

You may not realize it, but Verilog has many syntactical similarities to the C programming language. Interestingly enough, these similarities extend beyond syntax. The notion of defining and instantiating modules in Verilog is quite similar to defining and calling functions in C programming[4]. Table 5.2 shows a table listing

[4] Sorry, at this writing, C programming is the only higher-level programming language I know. While this may seem like a limitation, C is what 99% of the world programs hardware with, and we all know that embedded systems are what keep the world running.

the similarities between C and Verilog; here is some more description designed to convince you of these similarities.

- The formal parameters in C are the values you send to a function and the value that the function returns. These sent and return values are directly analogous to the inputs and outputs of a module in Verilog.
- The definition of a function (as specified by the programming code) determines what the function does. The body of a Verilog model determines what the hardware generated by the Verilog module does.
- We call our functions in a programming language, which is analogous to instantiating modules in Verilog. Note that we have one definition of both functions and modules that we reuse by calling them in a higher-level language and instantiating them in Verilog.

“C” programming language	**Verilog**
Describe function interface (formal parameters)	the *module (inputs and outputs)*
Describe what the function does (coding)	the *body of the module*
Call the function from main program	*Instantiate the component* from top module

Table 5.2: Similarities between "C" and Verilog.

5.6 SystemVerilog Considerations

Structural modeling is the same for both Verilog and SystemVerilog. Move along… nothing to see here.

5.7 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.
- Structural modeling in an HDL is analogous to working with functions in a higher-level programming language. The functions formal parameters are analogous to the inputs and outputs of a module. The function definition is analogous to the module definition. The calling of functions in a higher-level language is analogous to the instantiations of modules in Verilog.

5.8 Chapter Exercises

1) Name the three types of models associated with the Verilog HDL.
2) Can a Verilog model be both a structural model and a dataflow model? Briefly explain.
3) Explain the concept that structural model is about using previously design boxes in your design rather than designing new boxes.
4) Briefly describe what we mean by the term *flat design.* .
5) Briefly explain why structural modeling exists. Be sure to mention the notion of flat designs in your explanation.
6) Briefly explain whether structural modeled designs and flat designs are functionally equivalent.
7) Briefly describes what differentiates separate instantiations of the same module at any given level of a design.
8) Briefly explain how it is that all structural designs eventually become flat designs anyways.

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 dataflow 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 (or low-level operation) of the circuit as we did with dataflow models. 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 dataflow 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 important.

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 circuit that does not work the way you intended it to. 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 we synchronize most of the module's actions to an active clock edge input to 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 because **`initial`** blocks are not synthesizable. The majority of this text uses **`always`** blocks.

Now's a good time to remind you that your overall 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 (line by line). As you read the following verbage and examples that follow, don't allow the vernacular and syntax to make you to think that what your coding exploits have any relation to software, even though the syntax seems similar. If creating digital models feels like writing software, you're probably doing something wrong (and your circuit probably will not work as you intended).

6.3.1 The `always` Block

The **`always`** block is officially an infinite loop which processes statements within the loop repeatedly. There is a certain list of signals the "tell" the loop to do an iteration. 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 another 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 building that house in an efficient manner. When you build a house, you start out with the foundation and work upwards; you'll certainly run into issues if you build the roof first.

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, namely with behavioral models. The synthesizer interprets the code that goes inside an **always** block in a sequential manner, thus the order of the statements within an **always** block matters. The order that the procedural 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 circuitry within the block. There are three main types of statements that we can place inside **always** blocks, which we refer to as simple assignments statements, 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

[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.

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

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 throughout this text. 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 the way you intend it to work.*** 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 Assignment 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, which 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 completes 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 Assignment 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`** block 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 Circuit Modeling

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 specific set of rules when generating circuits. These rules are relatively simple; we'll state them here for completeness, but it will all make more sense when we start working with actual circuit examples.

[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 are details that are actually more specific involved that we save for another chapter.

Your mission as a digital circuit designer requires that you can model both combinatorial and sequential circuits. There are two approaches to model a sequential circuit using Verilog; we'll describe one of those approaches here and leave the other approach for another chapter. Here are the rules; they are a bit strange, but relatively easy to remember and follow.

1) ***If you incompletely specify a circuit's outputs based on the circuit's inputs, the synthesizer generates a sequential circuit.*** This means that if your procedural block does not specify an output for every possible input combination, the circuit must "remember" the previous output in case that combination appears on the block's inputs.
2) ***If you completely specify a circuit's output for every possible input combination, the synthesizer generates a combinatorial circuit.*** This means that if you specify an output for all possible input combinations, then your circuit does not need to "remember" anything, and thus your circuit is combinatorial. This sounds harder than it is; as you will see, procedural assignment statements include the notion of catch-all statements that allow you to easily provide an output for all possible input combinations[5].

Additionally, in this text, ***we only model sequential circuits using the `always` procedural block***. Another way to say this is that we don't use continuous assignment statements to model sequential circuits. It's not that we can't use continuous assignment statements to model sequential circuits; it's that we should not. Moreover, we generally never have a need to, so it's a big deal. More specifically, we use the **`if-else`** and **`case`** procedural programming statements in a special manner to generate either sequential or combinatorial circuits. Table 6.1 lists the two important guidelines regarding the generation of combinatorial and sequential circuits using procedural blocks (**`always`** blocks). Note that in this context, when we refer to sequential circuits, we are referring to latches, which are inherently asynchronous.

Circuit Type	Guidelines
Combinatorial	**Model using blocking assignments only** **Completely specify all input conditions (and/or include a catch-all)**
Sequential	**Model using non-blocking assignments only** **Do not completely specify all input combinations (don't include a catch-all)**

Table 6.1: Guidelines for modeling combinatorial and sequential circuits using `always` blocks.

6.4 SystemVerilog Considerations

SystemVerilog contains three additional types of procedural blocks. 1) **`always_comb`**, 2) **`always_ff`**, and 3) **`always_latch`**. These three new procedural blocks provide several advantages over using the basic **`always`** block. As the procedural block names imply, designers use the **`always_comb`** block when they are modeling combinatorial circuits, and use the **`always_ff`** and **`always_latch`** to model sequential circuits.

The issue here is that most of the advantages associated with these three new blocks compared to the plain **`always`** block involve more advanced modeling and verification techniques. This being the case, we opt to not cover them extensively in this text. We'll show a few equivalent uses of these blocks in later chapters once we start modeling actual circuits. Here are a few worthy comments regarding the use of these new procedural blocks or not.

- The clearest advantage of these three procedural blocks is that they indicate to the human reader, the synthesizer, and the simulator what the designer's intent is. In this way, the human

[5] Catch-all statements are a topic for another chapter where we are using actually examples.

gets a quick idea what is going one and the tools will generate an warning or error if you violate the notion of the block. Specifically, if you use **`always_comb`** and attempt to generate a sequential circuit, the tool will generate an error or warning.

- Using these blocks does not absolve the designers from knowing how to properly model circuits using procedural blocks. Designers still need to follow basic rules in order to properly design either sequential or combinatorial circuit; using these new procedural blocks does not magically make your models generate the correct circuit if you don't know what you're doing.
- These blocks are specific to SystemVerilog; you may be required to use Verilog and thus need to know how to properly use only **`always`** blocks in your models.
- There are some specific reasons people new to Verilog should be using only **`always`** blocks. We'll mention those later in the text, but suffice to say here that using **`always`** blocks as a beginner helps you develop the proper understanding of modeling digital circuits with an HDL such as Verilog.

6.5 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 contains two types of procedural blocks: **`initial`** blocks and **`always`** blocks.
- 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 then assigned. The actual assignment of non-blocking assignment statements is "scheduled" to occur when the procedural block terminates.
- Always model combinatorial circuits in Verilog using blocking assignments; make sure all possible input condition possibilities contain a specified output or use some type of catch-all clause in the **`always`** block.
- Always model sequential circuits in Verilog using non-blocking statements; make sure that these statements do not include any type of catch-all clause.
- SystemVerilog contains three new procedural blocks: 1) **`always_comb`**, 2) **`always_ff`**, and 3) **`always_latch`**. These blocks have some advantages over the basic **`always`** block, but most of the advantages are associated with advanced Verilog modeling.

6.6 Chapter Exercises

1) Briefly describe how behavioral modeling differs from dataflow modeling.
2) List the two types of procedural blocks that Verilog uses.
3) List the three ways you can model a "box" using Verilog.
4) Briefly describe the relation, if any, between sequential circuits and sequential statements within an `always` block.
5) List the two main types of statements we place in an `always` block.
6) List the two types of procedural assignment statements used in Verilog.
7) Briefly describe the difference between blocking and non-blocking assignment statements.
8) Briefly describe whether you can use procedural statements outside of procedural blocks.
9) Briefly describe one way that Verilog uses the notion of "completely specified circuits" and "incompletely specified circuits" to generate combinatorial or sequential circuits.
10) Briefly describe a way you can ensure your procedural block models a combinatorial circuit.
11) Briefly describe a way you can ensure your procedural block models a sequential circuit.
12) List the three new procedural blocks available in SystemVerilog.
13) List a few reasons why using only `always` blocks (and not the new SystemVerilog procedural blocks) is a good idea.

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.

Recall that decoders are combinatorial circuits, and are typically relatively complex circuits. We mention the notion of relatively complex because provides us with the direction of using Verilog procedural blocks (**`always`** blocks) to model decoders that than continuous assignment statements. Despite the fact that decoders are relatively complex circuits, modeling them using Verilog is straightforward, which is why we use them to start our Verilog modeling journey. We'll first describe decoders before we begin modeling.

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.

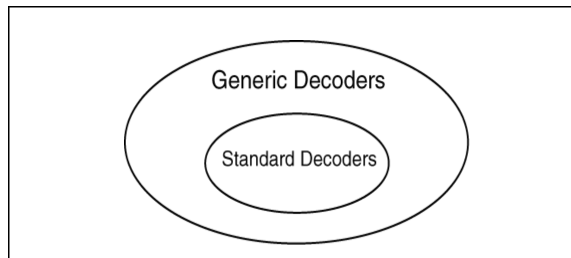


Figure 7.1: Venn diagram showing the hierarchy of decoders.

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.

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 the table does not explicitly define some input combinations).

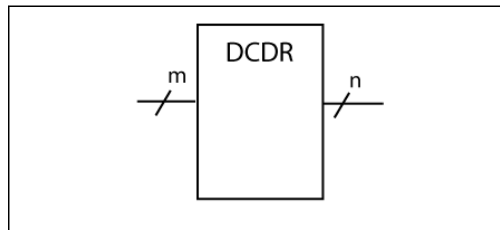


Figure 7.2: A black box diagram of a generic decoder.

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 to have no control inputs or status outputs. Table 7.1 provides a description of all the inputs and outputs to the generic decoder.

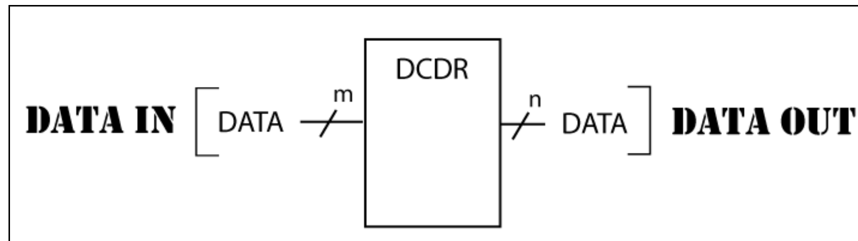


Figure 7.3: Data signals for a generic decoder.

	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.

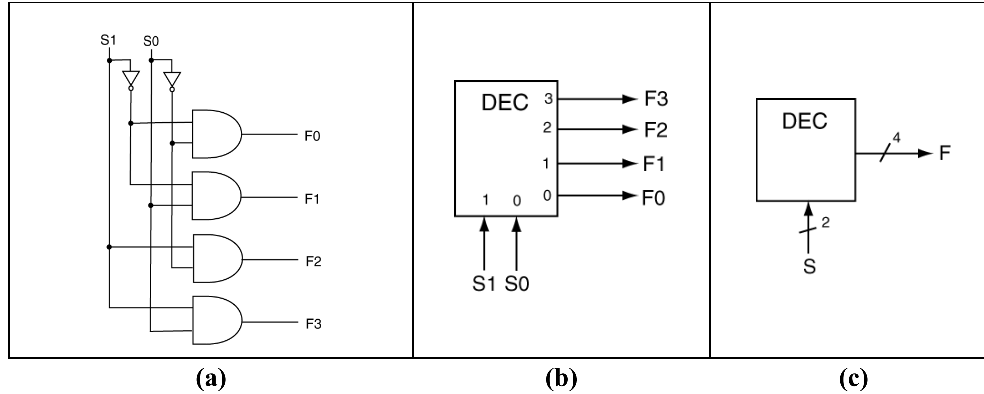
7.2.2 Standard Decoders

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. We could easily change the outputs to be active low to create a one-cold 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.

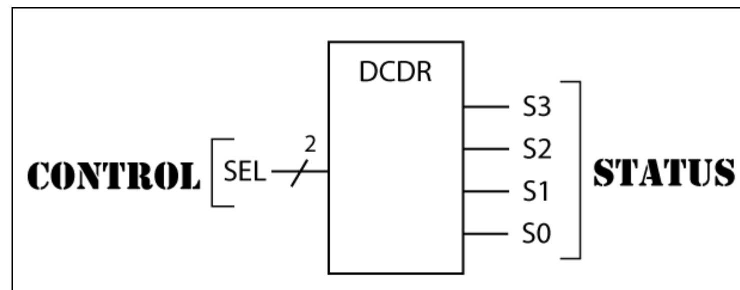


Figure 7.5: Control and status signals for a 2:4 standard decoder.

	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 Variable Type for Behavioral Modeling: the `reg`

Up until now, we have modeled circuits using only wires, which is a net-type in Verilog. This approach worked fine because we were only modeling relatively simple circuits, and all of the circuits were combinatorial. We are now entering modeling more complex circuits, which effectively necessitates the use of behavioral rather than dataflow modeling. Although the examples in this chapter are still combinatorial circuits, we need to introduce another type of modeling that supports the use of **`always`** blocks (behavioral models).

The **`reg`**-type is officially a variable type in Verilog; this is different from the **`wires`**, which are net types. There are many underlying issues with using **`reg`**-type vs **`wires`**, but it's relatively easy not to confuse the types if you follow a few rules. The **`wire`** and **`reg`** types are items you can assign values to; the big difference lies when and where you assign those values.

The first thing to note about a **`reg`**-type is that the name is horrifically misleading. While the name seems to imply that the type models some type of register, this is not *necessarily* the case, meaning that the **`reg`**-type can be a register (sequential circuit), but is not always a register (thus is a combinatorial circuit). You can actually use the **`reg`**-type in two ways: to model something with memory (a register) or model something that does not have memory (such as a **`wire`**)[1]. The way you use a **`reg`**-type in your model determines whether the synthesizer treats your **`reg`**-type as memory or a simple connection.

Though you *can* use **`reg`**-types to create registers, they can only do so within a procedural block, namely an **`always`** block. You will soon see that **`always`** block contain assignment statements within the body of the block; these assignments exist within procedural assignment statements, namely as simple assignments, **`if-else`**, and **`case`** statements. There are two important items to note here. First, all assignments within an **`always`** block must be to **`reg`**-types; it is thus not possible to assign a **`wire`**-type within an **`always`** block. Second, structuring of the assignments within the **`if-else`** and case statements determines whether the synthesizer induces memory on that **`reg`**-type or not. We can best describe these ideas using examples, which we do later in this chapter that discusses sequential circuits.

7.3.1.1 Module Instantiation Types: `reg` vs. `wire`

This may seem confusing at first, and it certainly is, but it becomes clearer when you start working with actual examples. Note that when introduced structural modeling in a previous chapter, we did so without the notion of

[1] Recall that the main use of wires was to connect parts of the circuit.

a **reg**-type. Figure 7.6 shows a diagram that outlines reg and wire usage for module instantiations. Here are the pertinent issues regarding Figure 7.6:

- The inputs to modules must be **wire**-types.
- The output of modules can be **wire** or **reg** types. The notion here is if the output is assigned from inside an **always** block, you must declare the output as a **reg**-type.
- When instantiating module, the outputs of that module must be mapped to a **wire**-type even if the modules declared the output as a **reg**-type.
- When instantiating modules, you can assign either a **reg**-type or **wire**-type to the module input, even though the module must declare inputs as **wire**-types with the module definition.

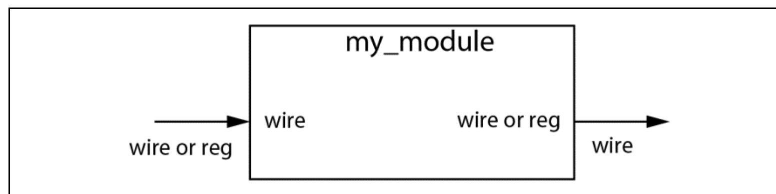


Figure 7.6: Visual explanation of reg vs. wire usage for module instantiations.

7.3.2 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 to enhance 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. The concatenation 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 model assigns the result of the concatenation operation to another signal, which can be a wire for continuous assignment statements or a reg for assignment within an always block. There must be at least two signals in the comma-separated list of signals, but you can concatenate as many signals as necessary.

Figure 7.7 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. In addition, worthy of noting is the different syntax for using the concatenation operator from inside and outside of **always** blocks.

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

wire [8:0]   new_sig_1;
wire [9:0]   new_sig_2;
wire [10:0]  new_sig_3;
wire [14:0]  new_sig_4;
wire [14:0]  new_sig_5;

reg [8:0]   new_sig_6;
reg [9:0]   new_sig_7;
reg [10:0]  new_sig_8;
reg [14:0]  new_sig_9;

assign new_sig_1 = {A,B};          //-  9-bit signal; A[3:0] are MSBs
assign new_sig_2 = {A,C};          //- 10-bit signal; A[3:0] are MSBs
assign new_sig_3 = {C,B};          //- 11-bit signal; C[5:0] are MSBs
assign new_sig_4 = {B,C,A};        //- 15-bit signal; B[4:0] are MSBs
assign new_sig_5 = {{4{A[3]}},B,C}  //- 15-bit signal; four A[3] bits are MSBs

always @ (my_vars)
begin  // for combinatorial circuit

   new_sig_6 = {A,B};    //-  9-bit signal; A[3:0] are MSBs
   new_sig_7 = {A,C};    //- 10-bit signal; A[3:0] are MSBs
   new_sig_8 = {C,B};    //- 11-bit signal; C[5:0] are MSBs
   new_sig_9 = {B,C,A};  //- 15-bit signal; B[4:0] are MSBs

   // other parts of always block …
```

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

7.3.3 Representing Integer Numbers

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

Figure 7.8 shows the format for specifying numbers in Verilog. Here is a description of the various fields in that format. Figure 7.8 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[2]. If you don’t assign a radix value, the synthesizer interprets the number as a decimal value.

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

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

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

Figure 7.8: 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)`
`'hE`	unsized	16	`0… 0 1 1 1 0 (32 bits)`
`8'd34`	8	10	`0 0 0 1 0 0 0 1 0`
`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) because they form the heart of behavioral modeling. 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 **always** blocks is to generate some actual models. Figure 7.9 shows a general form of the **always** block; the following describes the various fields in the **always** block 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.

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

Figure 7.9: The generic form of an always block.

In addition, 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. A good general rule in this situation is to include a **begin-end** pair the code requires it or if doing so makes the model more readable for humans.

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 that use **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[3]. There is more to this issue than we cover in this text; we instead follow a few simple rules that help ensure our models work properly.

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.*** Another way to say this is to include every **always** block input 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 unwanted 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 depending on whether we are modeling combinatorial or sequential circuits because we 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 7.10 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 7.10**(a). We must declare the output as a **reg** if the variable is assigned within the **always** block. **Figure 7.10**(b) contains an internal declaration of a **reg**; in this example, the **always** block will use the variable as it needs to but the variable is not an output of the block. We'll see this in actual use in later examples.

```
//-----------------------------------------
//- 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;
```

(a)

```
//-----------------------------------------
//- 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;
```

(b)

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

Figure 7.10: Code fragments showing reg-type variable usage.

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-else`** and **`case`** types. We consider the **`if`** statement a form of the **`if-else`** statement. These two 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 7.11** 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 7.11**(a) and **Figure 7.11**(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 7.11**(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. A good general rule in this situation is to include a **`begin-end`** pair if your given model it or if doing so makes the model more readable for humans.
- 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.

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

(a)

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

(b)

Figure 7.11: 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, which is a topic we cover in the chapter on sequential circuits. What we boldly state here is that *your **`always`** blocks always need some sort of catch-all to ensure the synthesizer generates a combinatorial circuit.*

(a)	(b)

`// for modeling combinatorial circuits`
`if (expression)`
`   one statement`
`else   //- catch-all`
`   one statement`
	`// for modeling combinatorial circuits`
	`if (expression)`
	`begin`
	`   statement(s)`
	`end`
	`else   //- catch-all`
	`begin`
	`   statement(s)`
	`end`

Figure 7.12: Two forms of the if-else statement

Figure 7.13 shows 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 catch-all, which guarantees that at least one statement in the **case** occurs, and thus prevents latch generation.

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

Figure 7.13: 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. This chapter describes only combinatorial circuits, which means we will only be using blocking assignments.

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

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

Figure 7.14: An example of blocking and non-blocking procedural statements.

7.4.5 Important Statement Considerations

The most important consideration when using **if-else** and **case** statements is to never lose track of the type of circuit you're modeling. More specifically, ensure your circuit synthesizes as combinatorial circuit if you're modeling a combinatorial circuit and as a sequential circuit if you're modeling a sequential circuit. While stating this sounds rather funny, we state it because the synthesizer can do unexpected things if you allow it to do so; knowing how to control the synthesizer is one of the main themes in this text.

Recall that if there is a specified output for every possible set of input conditions, the synthesized circuit will be combinatorial. This is of course a good thing if that is what you intended. The main problem area here is that often time you intended to model a combinatorial circuit, cut you forgot to provide an output for a specific input combination, which results in the synthesizer delivering a sequential circuit. You've probably noticed

that the **`if-else`** and **`case`** statements are similar, as is also the case in higher-level programming languages. However, the **`if-else`** and **`case`** statements differ in how they handle catch-all statements, which is the key to modeling either combinatorial or sequential circuits.

The **`if-else`** statements model combinatorial circuits when the associated model provides an output for every possible input combination. This leaves the circuit designer two options. First, they can provide an **`else-if`** clause for every possible combination, or, second, they can provide an **`else`** statement to end the **`if-else`** clause[4]. Good circuit designers always provide an **`else`** statement when modeling combinatorial circuits even if the **`else-if`** clauses completely specify the circuit. Providing the **`else`** clause allows humans to know immediately that the associated code models a combinatorial circuit. If the **`else`** clause was not there, the human reader typically won't be able to easily discern whether the circuit is combinatorial or not without a lot of extra time and effort.

The **`case`** statement models a combinatorial circuit when the model provides an output for every possible input combination. Similar to the **`if-else`** statement, there are two ways to provide an output for every input combination. First, the designer can provide a "case_item" for every possible input combination. Second, the designer can use a **`default`** statement to end the **`case`** statement. The **`default`** statement serves as the catchall statement to ensure the circuit synthesizes a combinatorial circuit. If the "case_item" clauses have provided an output for every possible input condition, the **`case`** statement does not require a **`default`** clause. In this case, good designers always provide a default case anyways as a message to humans reading the model that the code is intending to model a combinatorial circuit[5].

7.4.6 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`** statement 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: A partial list of Verilog's equality operators.

7.5 Solved Example Problems

This section contains several solved problems. These problems offer multiple solutions for each problem to provide you with a good feel for the flexibility of the Verilog language. There are more possible solutions for each of these examples; we provide what we feel are the better models.

[4] Doing both is also a useful option.

[5] Unfortunately, doing so can generate obnoxious warning messages.

Example 7.1: Decoder Example #1

Provide a Verilog model that describes the table 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 7.15 shows one 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.

- We've nicely aligned similar parts of the text; 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 must declare the output port **F** as a **`reg`** in the external interface.
- 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, the only inputs are the ABC bundle. We typically include all inputs in the sensitivity list unless we have a good reason not to.
- The **`always`** block uses blocking-type assignment statements because we are modeling a decoder, which 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).

```
module dcdr_generic_ex1(A, B, C, F);
    input  A, B, C;
    output reg F;     // declared as reg because it's assigned in always block

    //- declare bundle to make the solution cleaner
    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 7.15: Solution to Example 7.1.

There are many different approaches to implementing this model. Verilog syntax allows for many different coding styles; you should pick the one that generates the clearest model for what you're doing. The model in Figure 7.15 uses an **`if-else`** statement, but there is a preferable approach. Figure 7.16 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 7.16.

- 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 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;     //- catchall
       endcase
    end

endmodule
```

Figure 7.16: Another 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 7.17** shows a BBD for the solution.

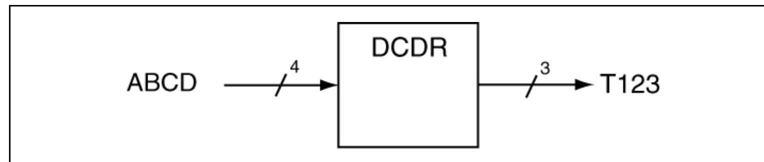


Figure 7.17: Black box diagram for Example 7.2.

Figure 7.18 shows one solution to this problem using an **`if-else`** statement while Figure 7.19 and Figure 7.20 show an alternate solution. A few things of merit in the solution in Figure 7.18:

- We aligned the code in all the models and added liberal amounts 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 **T123** output as a **`reg`**-type because we are assigning the output value from a procedural block (the **`always`** block).
- For the **`if-else`** version, we use decimal format in the **`if-else`** 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;    // catch-all
    end

endmodule
```

Figure 7.18: Yet another solution to Example 7.2.

Figure 7.19 shows an similar and functionally equivalent solution to this example. The solution in Figure 7.19 differs from the previous solution that the if clauses use binary notation to check the individual conditions.

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

    always @(ABCD)
    begin
       if      (ABCD == 4’b0000)  T123 = 3'b110;
       else if (ABCD == 4’b0001)  T123 = 3'b000;
       else if (ABCD == 4’b0010)  T123 = 3'b100;
       else if (ABCD == 4’b0011)  T123 = 3'b010;
       else if (ABCD == 4’b0100)  T123 = 3'b000;
       else if (ABCD == 4’b0101)  T123 = 3'b101;
       else if (ABCD == 4’b0110)  T123 = 3'b000;
       else if (ABCD == 4’b0111)  T123 = 3'b101;
       else if (ABCD == 4’b1000)  T123 = 3'b010;
       else                       T123 = 3'b000;     // catch-all
    end

endmodule
```

Figure 7.19: And yet another solution to Example 7.2.

Figure 7.20 shows yet another functionally equivalent solution to this problem using a **`case`** statement. This solution is probaby the best model in that the **`case`** statement better supports a long list of options. Using the notion of “better” essentially means “easier for humans to read”, which is most often the best approach[6].

[6] Unless of course you’re into job security, in which case you should make your code as unreadable as possible.

```
module dcdr_ex2c(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;  // catch-all
       endcase
    end

endmodule
```

Figure 7.20: Yet one more solution to Example 7.2.

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 (note that this signal is active high), the circuit acts like a normal decoder; when CE is not asserted, the circuit outputs zeros. Figure 7.21 shows the BBD for our solution; Figure 7.22 shows the associated Verilog model.

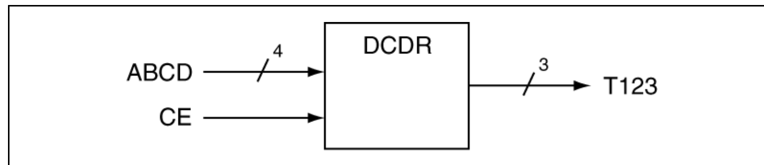


Figure 7.21: BBD for Example 7.3.

There are many different approaches to modeling this circuit; the model in Figure 7.22 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. Note that the case statement is essentially the complete model from the previous example.
- 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.
- We used blank lines to make the **`else`** clause stand out, which enhances readability for humans.

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

    always @(ABCD)
    begin
       if (CE == 1)       // CE active high
          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

       else              // the "disabled" case
          T123 = 3'b000;

    end  //- end of always

endmodule
```

Figure 7.22: 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 7.23 shows a block diagram for our 2:4 standard decoder; Figure 7.24 and Figure 7.25 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 in the two solutions.
- For both solutions, we provided an output for every possible set of input conditions, but we also provided a catchall statement (**`else`** for the **`if-else`** statement and a **`default`** for the **`case`** statement). Although this practice could cause the synthesizer to generate warnings, it enhances the understandability for humans because they know immediately that the code is modeling a combinatorial circuit.

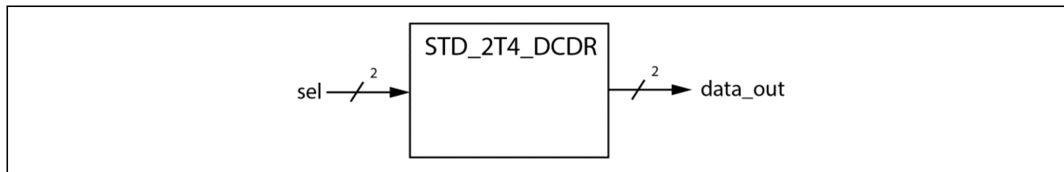


Figure 7.23: A BBD for a standard 2:4 decoder.

```
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 7.24: 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 7.25: Solution to Example 7.4 using a `case` statement.

Additionally, Figure 7.26 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 simplify the code and to avoid declaring a separate signal for the concatenated values.

```
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 as part of case statement
          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 7.26: Solution to Example 7.4 using a `case` statement with contenation of the select inputs.

7.6 SystemVerilog Considerations

We have two options that we can use to model combinatorial circuits using SystemVerilog. We always have the option to use logic types in place of both **`wire`** and **`logic`** types, but we now have the option to use the SystemVerilog **`always_comb`** procedural block in place of the **`always`** block.

Using the **`always_comb`** procedural block can be helpful in your models. Using this SystemVerilog feature has two advantages. First, it tells the synthesizer that you're intending to model a combinatorial circuit. This means that if you use the **`always_comb`** and the code in the body of the procedural block models a sequential circuit, the synthesizer will generate an error. Second, it provides a message to other humans reading your code that the give code is intending to be a combinatorial circuit.

Example 7.5: Standard Decoder

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

Solution: Figure 7.27 shows the SystemVerilog version of the 2:4 standard decoder. As you can see, the model is similar to that of the Verilog version. Here are a few differences of notes.

- We arbitrarily used **`logic`** types to represent the inputs and outputs in the external interface. Note that we previously specified the inputs as **`wires`** and the outputs as **`regs`**. We could have still used the **`always_comb`** procedural block with the **`wire`** and **`reg`** types.
- The **`always_comb`** procedural block does not have a sensitivity list. When you use the **`always_comb`** block, the block's sensitivity list defaults to every external signal used on the right side of the of assignment operators.

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

    always_comb
    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 7.27: A SystemVerilog version of the 2:4 standard decoder.

7.7 Chapter Summary

- If we can model the operation of a digital circuit using a truth table, we have essentially defined a decoder. Decoders are 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*. Two main types of procedural programming statements include **`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.
- When deciding whether to use a **`case`** statement or an **`if-else`** statement, the general rule is that if you have more than three items test then you should use a **`case`** statement; otherwise you should use an **`if-else`** statement.
- Including **`begin-end`** pairs in Verilog is sometime optional. The general rule to follow is that if include the **`begin-end`** pair makes the code more readable/understandable for humans, than include it.

7.8 Chapter Exercises

1) Using Verilog vernacular, what is the type for **`wire`**?
2) Using Verilog vernacular, what is the type for a **`reg`**?
3) Briefly explain why **wires** are only associated with combinatorial circuits.
4) Briefly describe what determines whether a **`reg`** induces memory or not.
5) Briefly describe why is it good practice to always use begin and end clauses when you use **`always`** blocks.
6) Briefly describe what we typically mean by the term "incompletely specified" output for a circuit.
7) List the two ways you can use an **`if-else`** clause to model a combinatorial circuit.
8) List the two ways you can use a **`case`** statement to model a combinatorial circuit.
9) Briefly describe the general rule of using a **`case`** statement vs. an **`if-else`** statement.
10) Briefly describe the good general rule as to use **`begin-end`** pairs where required in Verilog modeling.
11) Provide a Verilog model for an 3:8 standard decoder with one-cold outputs.

8 Multiplexors

8.1 Introduction

When you hear the word multiplexor, or MUX, you probably think "selector circuit". A multiplexor, or 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 set 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.

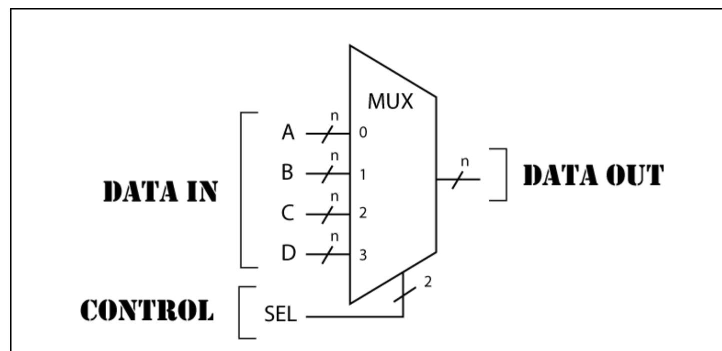


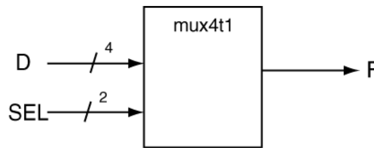
Figure 8.1: Data and control signals for a 4:1 MUX

	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 $\mathbf{2^{SEL}} \geq$ to the number of data inputs.
STATUS	**n/a**	-

Table 8.1: The foundation matrix for a 4:1 MUX.

Example 8.1: 4:1 MUX Model

Provide a Verilog model for 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 8.3 and Figure 8.4 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. For the sake of clarity, Figure 8.2 provides BBD for the MUX in this problem that expands the data input into individual bits and uses the definitive MUX shape.

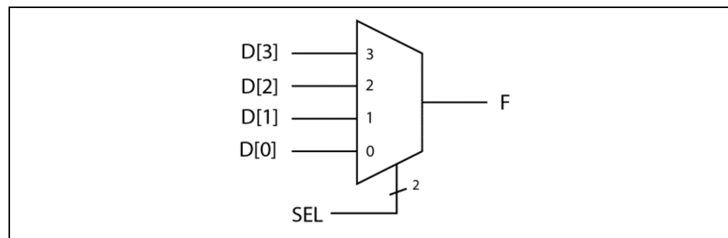


Figure 8.2: A more appropriate BBD for this problem.

- 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).
- We explicitly include a separate `if` clause for every possible input combination. Note that we could have relied on the `else` clause to handle the case when **SEL** is three, but explicitly listing all possible inputs is the better option.

```
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 8.3: The solution to Example 8.1 using an `if-else` statement.

Figure 8.4 shows a possible solution using a **`case`** statement. Here are a few things to note about this solution:

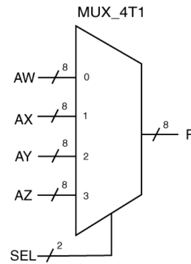
- We explicitly include a default case even though we have included a clause for every possible input condition. We could have relied on the **`default`** clause to handle the case when **SEL** is three, but explicitly listing all possible inputs is the better option for human readers.
- The **`case`** statement represents one statement, which is why we opted to not include a **`begin-end`** pair for the **`always`** block. My general rule in this situation is to include a **`begin-end`** pair the code requires it or if doing so makes the model more readable for humans.

```
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 8.4: 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 8.5 and Figure 8.6 show two solutions to this example; the model in Figure 8.5 uses an **`if-else`** statement while the model in Figure 8.6 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;       // on one line to save space
   output reg [7:0] F;

   always @(SEL, AW, AX, AY, AZ)      // all inputs in sensitivity list
   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 = 8'd0;   // 8-bit decimal value
   end

endmodule
```

Figure 8.5: 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)  // includes all inputs
      case (SEL)
         2'b00: F = AW;
         2'b01: F = AX;
         2'b10: F = AY;
         2'b11: F = AZ;
         default  F = 8'd0;
      endcase

endmodule
```

Figure 8.6: The solution to Example 8.2 using a `case` statement.

8.3 Generic Modeling in Verilog

As with all coding, we do our best in Verilog modeling to not "re-invent the wheel". What this means in terms of digital circuit modeling using Verilog is that we strive to write generic modules whenever possible.

Additionally, we also strive to re-use our previous models when possible. This brings up two distinct cases in Verilog: 1) writing models that exploit some of the features of Verilog to describe your circuits, and 2) writing models with variable data widths. As for the first matter, it is possible to model circuits in an iterative manner using some of the loop-type features of Verilog. Verilog actually contains several types of loop constructs, which fall under the category of procedural programming statements[1]. We only discuss the second type of genericity in this text.

8.3.1 Using Verilog parameters To Create Generic Models

Verilog uses the notion of a **`parameter`** to allow for generic datawidths in digital circuits. The parameter construct in Verilog allows modules to be reused but with different datawidth specifications. We present this topic this chapter because the MUX provides our first opportunity to effectively use parameters to model generic circuits. MUXes work great for this discussion because we often need to use the same type of MUXes in our digital designs (such as a 2:1 MUX or a 4:1 MUX), but those MUXes have different datawidths[2]. In this case, we certainly don't want to re-model the MUX for each individual datawidth we require. The better approach is to model a MUX in a generic manner (create a generic module out of it), and then be able to select the datawidth of the MUX when we instantiate the MUX in our model.

It's important to note here that this form of genericity only works with structural modeling. What is also interesting to note here is that using Verilog parameters is similar to function calls in a higher-level programming language in that the parameters are analogous to arguments you sent to a function. This means that I can send the function different data each time I call it. Similarly, I can model a digital module with different datawidths each time I instantiate it.

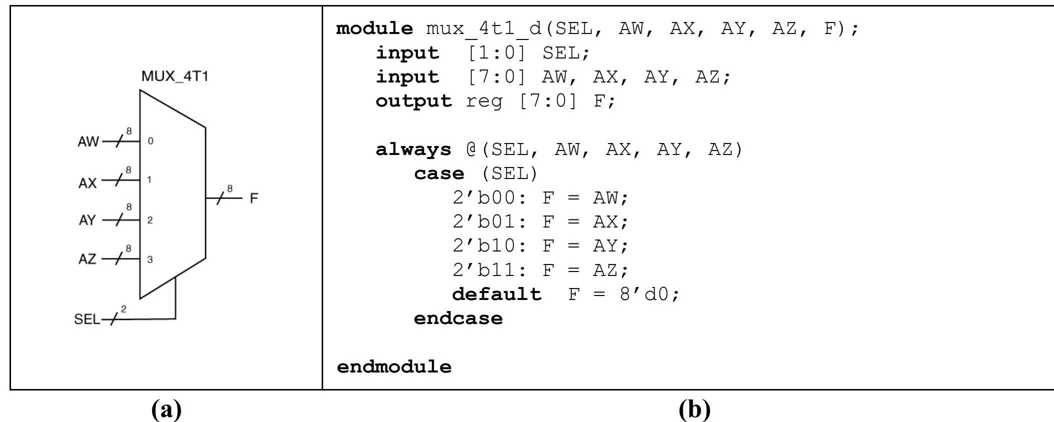
To be clear and official, there are two major types of parameters in Verilog: *module parameters* and *specify parameters*. This text only discusses module parameters. You'll find these to be quite powerful and relatively simple to use. Additionally, there is much more to parameters that we don't include in this text; our goal in this text is to get you up and going relatively quickly. We encourage the gentle reader to research the topic your own to obtain the full story and be able to become yet an even better Verilog modeler.

8.3.2 Module Parameters

The approach to using the parameter construct in your models is relatively simple. The first part of the process is to design your models using a default parameter. The second part of the process is you possibly override that default parameter when you create an instance (instantiate) that module. We'll demonstrate the use of parameters by parameterizing our model of a 4:1 MUX that we described earlier in this chapter, which we show again in Figure 8.7.

[1] Recall that the only procedural programming statements we use in this text are **`if-else`** and **`case`** statements. As you become a more advanced Verilog modeler, you should definitely check out other sources for information on these topics.

[2] This means the difference between a 2:1 MUX with 8-bit data compare do to a 2:1 MUX with 16-bit data, for example. In this case, the width of the control input (the data selection input) is the same, but the width of the data that the MUX selects is different.

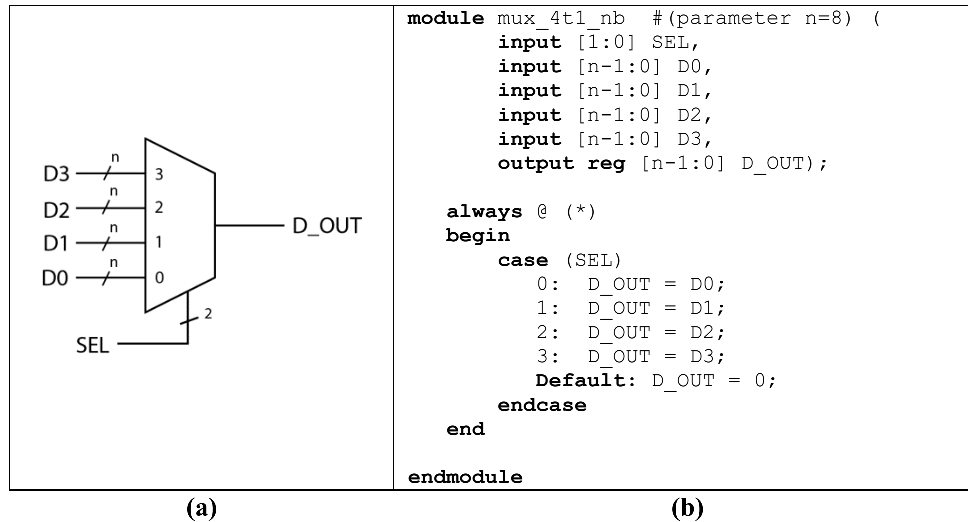


(a) **(b)**

Figure 8.7: An 8-bit 4:1 MUX BBD (a) and Verilog model (b).

The model in Figure 8.7 is good as it is, but we can make better by making it generic with the use of parameters. Making the model generic is a relatively simple process so we introduce the topic by showing a parameterized example and describing the meaningful features. Figure 8.8 shows the parameterized version of the 4:1 MUX. Here are the important things to note.

- The model defines the parameter value in this example on the first line of the code. The parameter in question is “n”.
- The code assigns an “8” to the parameter value, which is officially the default parameter value. You’ll see later that we can override this default value when we instantiate the module, or we can opt to not override the default value. If we don’t override the default value during instantiation, the datawidth of the MUX will be eight.
- We use the parameter “n” when we declare the datawidths in the model, which includes the four inputs and the single output. We use the notion of “n-1” because we typically like to ensure the right-most bits of the data are associated with the index of “0”. Note that the parameter definition and datawidth declarations are the only times the parameter appears in the model.
- We changed the names of the input and output data to something more “generic-looking” than the original model.
- The always block in this example uses the “*” in the sensitivity list, which is shorthand notation for including all inputs that the design uses (all the signals names appearing on the right side of the “=” assignment operator) into the sensitivity list.




```
module mux_4t1_nb  #(parameter n=8) (
       input [1:0] SEL,
       input [n-1:0] D0,
       input [n-1:0] D1,
       input [n-1:0] D2,
       input [n-1:0] D3,
       output reg [n-1:0] D_OUT);

   always @ (*)
   begin
       case (SEL)
          0:  D_OUT = D0;
          1:  D_OUT = D1;
          2:  D_OUT = D2;
          3:  D_OUT = D3;
          Default: D_OUT = 0;
       endcase
   end

endmodule
```

(a) **(b)**

Figure 8.8: The BBD (a) and parameterized model of a 4:1 MUX.

That's half of the parameter story; the other half of the story is how to override the parameter when you instantiate the MUX. Figure 8.9 shows an example instantiation associated with the model in Figure 8.8. Here's the good stuff to know about this instantiation.

- Then module name in the instantiation matches the module names in the module definition.
- We use the label value to indicate the instantiated MUX is 16 bits.
- We use "#()" notation to override any parameters. We call out the associated "n" parameter by name in this example, though we did not need to since there is only one parameter in this model[3].
- We inserted "xxxx" for items that you would need to map to more intelligent signals within an actual design. Once again, "xxxx" means nothing for this example; it serves only as a placeholder to remind the modeler that something should be mapped to those values when you instantiate the module.
- Note that that only the first line in this instantiation differs from other instantiations we've worked with up to this point in this text.

```
mux_4t1_nb  #(.n(16)) my_4t1_mux_16bit  (
     .SEL   (xxxx),
     .D0    (xxxx),
     .D1    (xxxx),
     .D2    (xxxx),
     .D3    (xxxx),
     .D_OUT (xxxx) );
```

Figure 8.9: Instantiation of a 16-bit 4:1 MUX.

Figure 8.10 shows another instantiation of this module. In this instantiation, we don't list the parameter value, which effectively prevents the instantiation from overriding the default parameter value given in the original model. This being the case, the code in Figure 8.10 instantiates an 8-bit 4:1 MUX because the parameter value in original MUX model defaults to eight.

[3] If there were multiple parameters in the associated module, you would need to override them individually in the instantiation.

```
mux_4t1_nb   my_4t1_mux_8bit  (
     .SEL   (xxxx),
     .D0    (xxxx),
     .D1    (xxxx),
     .D2    (xxxx),
     .D3    (xxxx),
     .D_OUT (xxxx) );
```

Figure 8.10: Instantiation of an 8-bit 4:1 MUX.

8.4 Unused Inputs and Outputs in Module Instantiations

Often times when we reuse previously designed modules in our structural models, there can be both inputs and/or outputs in those modules that our particular design does not use. In the context of this chapter for example, maybe we need a 3:1 MUX. In this case, our only option would be to instantiate a 4:1 MUX and "not connect" one input[4]. Additionally, if there is an output in a parameterized model we don't use in our design, we also do not map those outputs in our instantiation. The issue of "not connecting" or "not mapping" inputs and outputs in module instantiations is different; here are the pertinent differences.

> **Unused Instantiation Inputs:** Recall that we are modeling digital circuits. This generally means that all the inputs to a circuit require that they have a value assigned to them. Because of this, we always both keep the unused inputs in the instantiation and we always assign them to be either a '1' of a '0'. If we do not map unused inputs in our instantiations, the synthesizer assigns these values for you. The best example of why this is potentially problematic is that the synthesizer may assign an unused control input to an active state when the synthesizer should have assigned to the inactive state[5]. In short, you simply don't want to give the synthesizer this amount of control over your circuit.
>
> **Unused Instantiation Outputs:** The unused outputs of a digital circuit are inherently different from the circuit's inputs in the context of device electronics. In any digital device, we always assign all the inputs to keep them "under control", which prevents them from floating. Conversely, if we have unused outputs in our digital circuits, they generally are not connected to any other inputs in the circuit and we thus can allow them to float. This means in module instantiations, we don't remove the outputs from the instantiation template, but we simply don't map them to any other circuit inputs. We "don't map them" by leaving the parenthesis empty.

Figure 8.11 shows a model of a 4:1 MUX that is essentially instantiated as a 3:1 MUX. Note that in Figure 8.11, the D3 output is hardcoded to a constant value. In this case, the hardcoded value is all zeros (eight zeros for this MUX). Note that even though we are not using the D3 input, we keep in the input in the instantiation and map it to a value. The inputs and outputs in this model prefixed with the "my_" would actually be signals in the circuit level that the instantiation appears.

[4] What you really don't want to do here, which newbies often do, is remove the unused input from the instantiation template. As horrible as this sounds, it's much better than the other newbie tactic of modifying the parameterized model in any way.

[5] There's probably some rule of how the synthesizer assigns unused inputs, but I don't know what it is. I don't know what it is because it a really bad idea to rely on such rules, as they may change if you use a different synthesizer.

```
mux_4t1_nb   my_4t1_mux_8bit  (  //- 3:1 MUX using 4:1 MUX parameterized model
     .SEL   (my_sel),
     .D0    (my_d0),
     .D1    (my_d1),
     .D2    (my_d2),
     .D3    (8'b0000_0000),      // unused input
     .D_OUT (my_dout) );
```

Figure 8.11: Instantiation of an 8-bit 4:1 MUX that is essentially a 3:1 MUX.

8.5 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.
- One method Verilog uses to support the notion of generic modeling is the use of parameters. Using parameters allows modelers to reuse code by allowing them to override model default parameters when they instantiate the modules.
- Using a "*" in place of the sensitivity list for a procedural block provides a shorthand notation for including all inputs used in the procedural block into the block's sensitivity list.
- When instantiating modules, our particular design often times don't use some inputs and outputs contained in the instantiated module. In these cases, we never remove the inputs/outputs from the design. In these instances, we map the inputs of a constant value (assign it some input that won't cause the module to act strangely) and we don't map the outputs (we level the parenthesis blank).
- Best Verilog coding practices include never using procedural programming catchalls to represent specific valid inputs. It is better to explicitly list every possible input combination with an `if` clause or `case` item clause.

8.6 Chapter Exercises

1) Briefly describe the connection between completely specified outputs and combinatorial circuit in procedural blocks.
2) Briefly describe what the term “catch-all” means for procedural programming statements.
3) Briefly describe whether you need to include a catchall statement in your procedural programming statements if your model specifies an output for every possible input combination.
4) Briefly describe why generic model approaches support the basic tenets of digital design.
5) Briefly describe why it is important to model digital modules such as MUXes with generic modeling techniques in Verilog.
6) Briefly describe why it is the best idea to not rely on catch-all statements for valid input combinations.
7) List the two *parameter* types used in Verilog.
8) Briefly describe the functionality using a “*” in the sensitivity list for a procedural block provides.
9) Provide a Verilog model for a 2:1 MUX that uses 1-bit data.
10) Convert the model from the previous problem to use a parameter for the data width.
11) Provide a Verilog model for an 8:1 MUX that uses 1-bit data.
12) Convert the model from the previous problem to use a parameter for the data width.
13) Repeat the previous problem but include a control input: CS. When this input is asserted (active high), the MUX acts like a standard MUX; when not asserted, the MUX output is zero.
14) Briefly describe why we never remove unused inputs and outputs from modules when we instantiate them into our design.
15) Briefly describe what happens when you do not provide a signal name for an input in an instantiated module.
16) Briefly describe the correct approach to mapping unused inputs in module instantiations.
17) Briefly describe the correct approach to mapping unused outputs in module instantiations.
18) Briefly describe why it is good practice to never allow procedural assignment statement catchalls to include expected cases.

9 Comparators

9.1 Introduction

A comparator is a combinatorial digital device that compares two numbers and provides a status (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's behavioral modeling.

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. 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**.

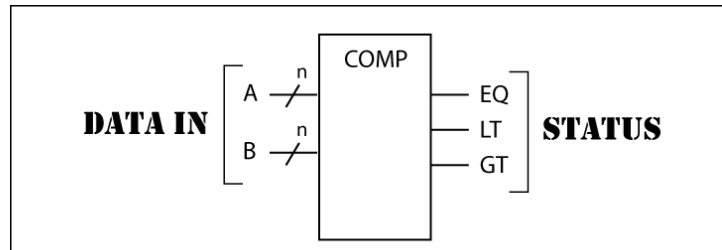


Figure 9.1: Typical data, and status signals for a comparator

	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

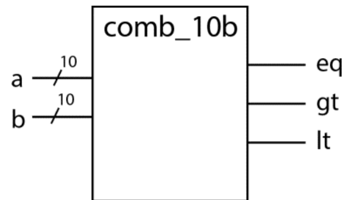
Verilog contains of set of relational operators that we typically use as expressions in procedural assignment statements. The most common use for relational operators is in `if` clauses in `if-else` statements. **Table 9.2** shows the set of Verilog's relational operators.

Operator	Example	Description
`==`	`(X == Y)`	Does X equal Y? (true or false; 1-bit result)
`!=`	`(X != Y)`	Does X not equal Y? (true or false; 1-bit result)
`>`	`(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 unsigned 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 9.2 shows the solution to this problem. The solution contains some new items, which we describe in the following bullets.

- The model uses three relational operators to determine the **eq** (==), **gt**(>), and **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. By the way we wrote the code, the **`else`** statement will never be evaluated because one of the previous **`if`** statements will always evaluate as true.
- Each **`if`** clause assigns all three outputs; if we assigned less than three outputs per **`if`** clause, the model would necessarily induce memory (generate a latch), and it would thus not be a proper comparator (comparators are combinatorial circuits). Stated differently, if we forgot to assign one of the outputs in an **`if`** clause, the circuit's outputs would be incompletely specified, and thus cause the synthesizer to induce a latch in the resulting hardware. There is a different approach to this problem in the provided alternate solution.
- 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 solution.
- This model only works for unsigned number; if you were to consider the inputs as signed numbers, the status outputs would be incorrect if your circuit interprets one or both of the inputs as a signed number.

```
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.

Figure 9.3 represents an alternative form of the solution for this example. Being that the comparator is a combinatorial circuit, one of our unstated goals in modeling the circuit was to ensure that the model does not induce the synthesizer to generate a latch. The method we used to prevent latch generation in the previous solution was to assign all three of the outputs in each of the three **if** clauses. While this approach is valid, we present another solution to highlight the internal workings of the Verilog synthesizer. Here are the interesting things to note about this alternative solution.

- We assign each of the three outputs upon entering the **always** block. By assigning all the outputs at this point, we ensure that the **always** block assigns each of the three outputs each time the **always** block is evaluated. Recall that the **always** block will be evaluated each time there is a change in the **A** or **B** signals. At this point, we evaluate the inputs with the **if** clauses; when one of inputs evaluates as true, the model *re-assigns* one of the three output signals, but relies on the initial assignments (previously scheduled assignments) for the other two output signals. This works because the model does not actually assign the signals until evaluation of the **always** block completes. What this means is that it is no big deal to "reassign" these values because the values have not been officially assigned in the first place[1]. While this approach may seem strange, it represents typical and expected Verilog modeling practices.
- The always block must use a **begin-end** pair because the always block contains four statements: three assignment statements and one procedural programming statement (the **if-else** statement).
- The model places blank lines between the **if-else** clauses to enhance readability of the model for humans.

[1] Note that this is one of the huge differences between Verilog (and other HDLs) and programming languages. In programming languages, when something is "assigned", stuff really happens in hardware. In Verilog, the assignment operators represent something that is "scheduled" for assignment after the procedural block terminates.

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

    always @ (a,b)
    begin
       eq = 0;  lt = 0;  gt = 0;   //- prevent latch generation

       if (a == b)  eq = 1;

       else if (a > b)   gt = 1;

       else if (a < b)   lt = 1;

       else
          eq = 0;
    end //- always

endmodule
```

Figure 9.3: An alternative solution for this example.

9.4 Generic Comparator Models

The comparator is a common digital module that finds its way into many digital designs. These designs 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 width. 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 9.5 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". Figure 9.4 provides a new BBD for this generic comparator. As you can see, the parameterized model in Figure 9.5 is almost identical to the unparameterized model in Figure 9.2. Figure 9.6 provides an example that instantiates a 24-bit comparator using our parameterized n-bit comparator module.

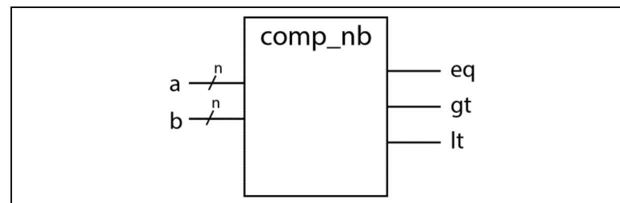


Figure 9.4: BBD of an n-bit comparator.

```
module comp_nb  #(parameter n=8) (a, b, eq, lt, gt);
    input  [n-1:0] a, b;    //- on one line to save space
    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 9.5: The parameterized solution for an n-bit comparator.

```
comp_nb  #(.n(24)) comp_24b  (
     .a    (xxxx),
     .b    (xxxx),
     .eq   (xxxx),
     .lt   (xxxx),
     .gt   (xxxx)    );
```

Figure 9.6: Instantiation of a 24-bit comparator.

Example 9.2: Module-Based 12-Bit Comparator

Provide a Verilog structural model for a 12-bit unsigned comparator by instantiating 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 truly needed a 12-bit comparator, we would use the generic model of Figure 9.5. 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 9.7 shows a BBD for the solution while Figure 9.8 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 9.7. We included the signal names to make the diagram better match the model. The choice of how to distribute 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, which 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 and inside the parenthesis.

- 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 parameters and to explicitly list the parameterized values, which makes your models more readable to humans.
- 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 (removed them from the instantiation), but that would have made human readers wonder whether we forgot to include them or that we truly did not need them. Including them and leaving them blank is the best approach.

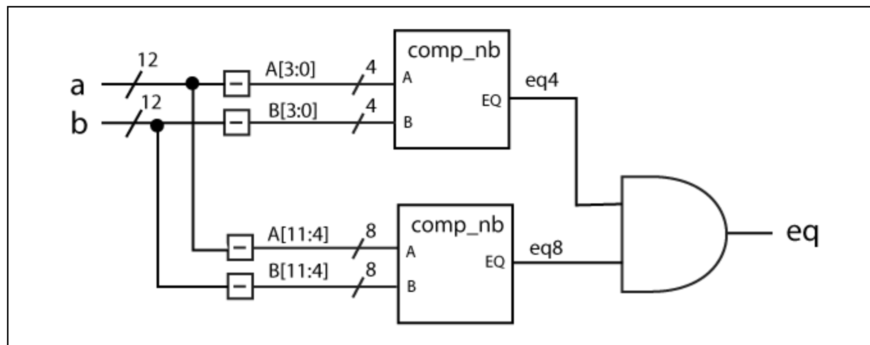


Figure 9.7: BBD for the solution to Example 9.2.

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

   //- intermediate signal declarations
   wire eq4,eq8;

   assign eq = eq4 & eq8; // continuous assignment

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

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

endmodule
```

Figure 9.8: The solution to Example 9.2 using two instances of a generic comparator.

Figure 9.9 shows an alternative solution to this example. This solution highlights the notion that how we chose the eight and four-bit comparators for the original solution was arbitrary. In this alternative solution, we change the 8-bit comparator from the upper eight bits to the lower eight bits; similarly, we change the 4-bit comparator from the lower four bits to the upper four bits. One of the major highlights of the change is the fact that the once you know what to change in the models, the actual required change is minimal because it only involves the mapping of the **a** and **b** inputs to the underlying comparator instantiations. Note that often times when working with Verilog you will be primarily be using predefined modules, which makes the modeling process fast based on cutting and pasting the module templates rather than defining the underlying modules themselves.

```
module comp_12b2(a, b, eq);
   input  [11:0] a,b;
   output eq;

   //- intermediate signal declarations
   wire eq4,eq8;

   assign eq = eq4 & eq8; // continuous assignment

   //- instantiate 4-bit comparator
   comp_nb  #(.n(4)) MY_COMP4 (
      .a  (a[11:8]), //- most significant 4 bits
      .b  (b[11:8]),
      .eq (eq4),
      .gt () ,
      .lt ()    );

   //- instantiate 8-bit comparator
   comp_nb  MY_COMP8 (
      .a  (a[7:0]), //- least significant 8 bits
      .b  (b[7:0]),
      .eq (eq8),
      .gt (),
      .lt ()   );

endmodule
```

Figure 9.9: The solution to Example 9.2 using two instances of a generic comparator with different comparator mappings.

9.5 Working with Signed Numbers in Verilog

When you're first learning Verilog, the tendency is to stick with unsigned numbers. All the examples in the text up to this point have used unsigned numbers, which is because we did not explicitly declare the numbers are signed values. We try to stick with only unsigned numbers as part of the "learning Verilog process" because working with signed and unsigned numbers can be rather tricky at first. Working with signed numbers is not an overly complicated, but it does require you to understand or what you're doing and to memorize a few rules. We'll go over the basics in this section but as of this writing, we still do our best to use mostly signed number representations in this text.

We place this discussion in the chapter dealing with comparators to highlight the fact that the comparator models we used in this chapter were only work as expected with unsigned numbers. Once you understand a few concepts, you'll see that it's straightforward to convert our comparator models to work with signed numbers. The examples in this chapter only deal with area of working with signed numbers; we'll revisit this topic in a later chapter with more examples.

9.5.1 The Big Overview

Before we start, let's recall what we're doing. We're modeling digital circuits with Verilog; we then typically synthesize these models to eventually implement an actual circuit. Thus, modeling using Verilog is a matter of writing "text" and having that text magically transformed into a working circuit. The important issue here is that we need methods to communicate our circuit-based desires with the synthesizer. We've dealt with many of

these communication issues up to this point; we now need to deal with more of them when we start working with signed and unsigned numbers.

Recall that when we work with digital circuits, everything is either a one or a zero; what makes these 1's and 0's interesting (and more complicated) is the notion that our digital circuits need to interpret those sets of 1's and 0's differently. Note once again that the hardware does not know the difference between a set of bits that represents a signed or unsigned number: the hardware only sees a set of bits. The mission of the hardware modeler is to ensure the hardware does the correct action with those bits based on our intended use of those bits.

9.5.1.1 Signed and Unsigned Numbers in Verilog

Initial versions of Verilog didn't provide any special support for signed numbers, which means that Verilog only dealt with unsigned number representations. For this reason, modelers had to do be more explicit in the way they handled a signed vs. an unsigned number. The 2001 version of Verilog added support for signed numbers, which had two effects. First, it made dealing with signed and unsigned numbers a bit less complicated for modelers. Second, it required modelers to understand the "rules" of how Verilog worked with signed number. These rules are somewhat intuitive, but mostly arbitrary. The good news is that there relatively few rules to deal with, particularly when working with relatively simple circuits.

9.5.2 Operator Expansion Rules

Up to this point, all the operations we've done in this text have been on operands of the same size, which includes both sides of the assignment operator. This approach was handy for learning the Verilog basics, but it often not convenient to do in your models. We've seen that we can explicitly concatenate a set of signals to form new signals, but that was something "we" the modeler did. You need to realize that Verilog has a general set of expansion rules for cases when the operands in an expression as not all of the same size. We mention this here as a prelude to working with signed and unsigned numbers in Verilog because it's an action done by the synthesizer and not explicitly by the modeling.

The general rule in Verilog is that the synthesizer expands all operands in an expression to the width of the largest operand in the instruction. This rule includes operands on both sides of the assignment operator. Additionally, if your expression contains a concatenation or replication operator associated with a given operand, the synthesizer completes those operations before implementing required expansion on any given operand[2]. Expansion of a given operand means that the synthesizer must arbitrarily "add" more bits to that operand. The question here is what bits does the synthesizer add? The answer is relatively simple in that it relates to working with signed number representations in basic digital design. Verilog bases the two expansion rules on the "signedness" of the operand:

> **<u>Signed Values</u>**: the synthesizer uses sign extension, which means the synthesizer replicates the sign-bit (left-most bit) as many times as is required to expand the value to the required width. We sometimes refer to this operation as *left-extending* with the sign bit.
>
> **<u>Unsigned Values:</u>** the synthesizer uses zero extension, which means the synthesizer adds the required number of zero bits to the left side of the value in order to obtain the required width. We sometimes refer to this as *left-extending* with zero.

9.5.3 Verilog Signed and Unsigned Values

We deal with signed and unsigned values in this chapter in two different areas: 1) telling the synthesizer the permanent meaning of a set of bits representing a number, or 2) tell the synthesizer of the temporary meaning of a set of bits representing a number. As for meaning of a set of bits, we mean whether we intend the number to be either a signed or unsigned number, which do when we declare the number. As for temporary meaning of a number, we can *cast* the value in a given expression.

[2] Sort of like putting parenthesis around items in a mathematical expression.

9.5.4 Operations and the Signedness of Operands

As you have seen, we work with many expressions in Verilog[3]. The general rule in Verilog is that if the synthesizer considers any operand in the expression as unsigned, the synthesizer considers the entire operation to be unsigned. The more functional way of saying this is that if you want your expression to have a signed result, all the operands must also be signed values as well. This is an arbitrary choice made by the designer of the Verilog language. This requires some thought by the person doing the modeling. If all your expressions are either all signed or all unsigned values, then the synthesizer does some of the grunt work for you. On the other hand, if you're using expression that have operands with different signs, then you really need to know what you're doing to provide the operands to the synthesizer in such a way to allow it to do what you want they synthesizer to do.

9.5.4.1 Permanent "signedness"

We have two main approaches to permanently assigning a signedness to values: 1) we can either include the signedness in the declaration of the value in the module's external interface (an input our output port), or 2) we declare the signedness of the signals as intermediate signals declarations within the body of a module. Here is more information on those two instances; additionally, Figure 9.10 shows a few examples with self-explanatory comments in code fragment

Port Interfaces: Up until this point, we've simply been declaring inputs and outputs as nothing other than their width specifiers. By not explicitly including a sign specification in the declaration, we allow the synthesizer to take the default sign specification, which is unsigned. It we want our signals to be signed, then we need explicitly state that the numbers are signed in the declaration; this is true for both inputs and outputs. Keep in mind that some module outputs may also be of **`reg`**-types, which would then have both a sign and width modifier in the declaration.

Internal Signals: We often run across the need to use internal signals within our modules. These internal signals can either act as connecting signals between modules or can model memory (such as register), depending on usage. Accordingly, modelers can assign a signedness to these values.

```
module my_ckt (
       input signed [1:0] SEL,  // external interface
       input signed [7:0] D0,
       input signed [7:0] D1,
       output signed reg [7:0] D_OUT);

   reg  signed  [7:0] signed_reg;  // intermediate signals
   wire signed  [7:0] signed_wire;
   wire         [7:0] unsigned_wire;

   //- meaningful code goes here
   //-  ...

endmodule
```

Figure 9.10: A code fragment with examples of declaring signed and unsigned values.

9.5.4.2 Temporary "signedness": Typecasting

There's probably a better way to describe this, but the notion of "temporary signedness" is all can think of at this writing. The notion of typecasting means that we want the synthesizer to temporarily consider a value to be of a type different from the type associated with its declaration. The notion of casting is simply a message to the synthesizer to treat a value in a way different from how the model originally declared the value; in other words, casting overrides the signedness of the signal when the signal was declared. We generally use type casting in two different ways: 1) to tell the synthesizer how to expand

[3] Such as an assignment operation.

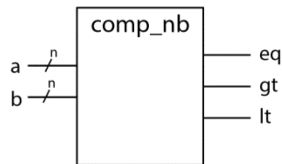
the value as it appears in an expression, and 2) to tell the synthesizer how to interpret the value when it appears in an equality clause. Here the notes on these two items.

Expanding Values in Expressions: We already know that the Verilog synthesizer will expand the bitwidth of operands in expression to the width of the operand with the largest bitwidth in the expression. The synthesizer bases this expansion on the declared signedness of the operand. Specifically, to *fill* the extra bits, the synthesizer sign-extends signed numbers and zero-extends unsigned numbers.

Equality Operations: The equality operator in Verilog understands the signedness of numbers. This means that you can make both signed and unsigned comparisons as part of procedural programming statements (**`if-else`** statements for example); the synthesizer bases these comparisons on the declared signedness of the variables [4]. Using casting in this manner instructs the synthesizer as to how to interpret the set of bits that signal represents: either as an unsigned value or a signed value if the interpretation needs to be different from how the model declared the value.

Example 9.3: Module-Based 12-Bit Comparator

Model the generic n-bit unsigned comparator we previously design as an n-bit signed comparator. Provide two solutions: one solution should use typecasting and the other should not.



Solution: The starting point from this problem is the previous generic solution. We modify that code and show the first solution in Figure 9.11. Here are some pertinent observations:

- The only significance in this code is that we declared the inputs as signed values in the internal interface.
- We include an **`else`** statement to let the world know we are modeling a combinatorial circuit. Note that this **`else`** statement will never be evaluated based on the conditions associated with the other three **`if`** clauses.

[4] Note here that it only makes sense to compare two values of the same sign.

```
module comp_nb_signed_1  #(parameter n=8) (a, b, eq, lt, gt);
    input signed [n-1:0] a, b;   // input values declared as signed
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       eq = 0;  lt = 0;  gt = 0;

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

endmodule
```

Figure 9.11: An n-bit signed comparator module with signed inputs.

Figure 9.12 shows the other solution to this problem. This particular solution implicitly declares the two inputs as unsigned values. The solution then casts the two input values as signed values for comparison purposes.

```
module comp_nb_signed_2  #(parameter n=8) (a, b, eq, lt, gt);
    input [n-1:0] a, b;   // input values are unsigned
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       eq = 0;  lt = 0;  gt = 0;

       if (a == b)                              eq = 1;
       else if ( $signed(a) > $signed(b) )  gt = 1;
       else if ( $signed(a) < $signed(b) )  lt = 1;
       else                                     eq = 0;
       end
    end

endmodule
```

Figure 9.12: An n-bit signed comparator module with unsigned inputs and typecasting in conditional clause.

9.6 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.
- If you omit the sign of a variable in a declaration, Verilog will interpret that number (that set of bits) as an unsigned value. This means that signed values must be explicitly declared as such.
- The typical comparators work with either two signed operands or two unsigned operands.
- When modeling comparators or any other type of combinatorial circuit, you must be sure to define all outputs from a procedural block (**`always`** block) each time the block is evaluated. Another way to say this is that each time an input to an **`always`** block changes, you must assign all the outputs from the block if you are modeling a combinatorial circuit.
- The Verilog 2001 provided built-in support for signed data including typecasting operators. The signedness of a number provides two messages from the modeling to the synthesizer. First, it gives the synthesizer instructions on how to expand the value when required (values are sign-extended for signed numbers and zero-extended for unsigned numbers). Second, the signedness of a number directs the synthesizer how to interpret the number for comparisons.
- We can use typecasting operators to instruct the synthesizer to interpret values in formations other than how the modeler originally declared them. The two typecasting operators are *$signed(val) and unsigned(val).*

9.7 Chapter Exercises

1) Briefly explain why we often list all the outputs of an **always** block at the beginning of the **always** block (after the **begin** clause).
2) Briefly explain what happens when an **always** block does not assign all the outputs once an input change causes the **always** block to be evaluated.
3) Briefly describe what happens if you don't override the parameters when you instantiate a parameterized module.
4) Briefly describe why explicitly listing all parameterized when instantiating a parametrized model is a better idea than relying on a parameterized model's default values.
5) Provide a model for a 22-bit comparator using at least two instantiated comparator modules. For this problem, use the following comparator model as the module to instantiate.

```
module comp_nb  #(parameter n=8) (a, b, eq, lt, gt);
    input  [n-1:0] a, b;     //- on one line to save space
    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
```

6) Briefly describe whether a given set of bits in a digital circuit know whether they represent a signed or unsigned number.
7) Briefly describe what we mean by the "signedness" of a number.
8) Briefly describe what happens if you don't state the sign of a value in a declaration.
9) Briefly describe if the values in a given design are signed or unsigned.

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 primarily 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. We can extend the use of the RCA by including a full adder in the LSB position, which makes taking two's complements and cascading RCAs easier.

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 internet searches are for…amirite?).

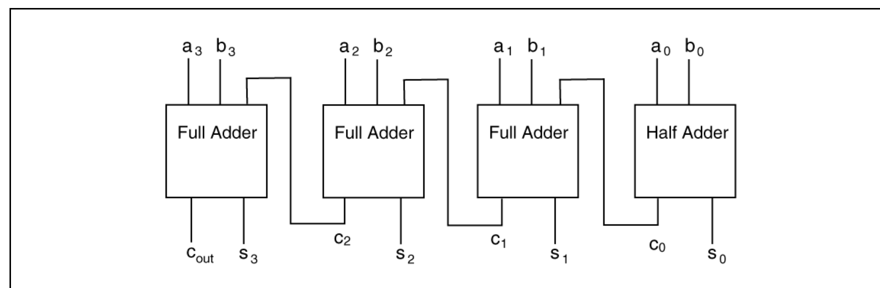


Figure 10.1: Lower-level BBD for a 4-bit Ripple Carry Adder.

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-in 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.

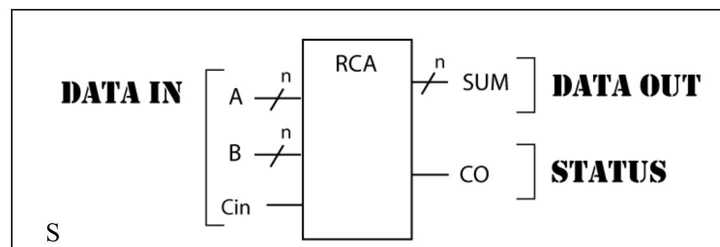


Figure 10.2: Data, control and status signals for a RCA.

	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. However, being that Verilog is an HDL, there are some other issues involved that are less obvious but particularly more important than 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: A partial list of Verilog's arithmetic operators.

10.4.1 Using Mathematical Operators

It's quite easy to start using Verilog mathematical operators in your code. The important issue here is to prevent this easiness from distracting you from the notion that you're designing hardware and not writing a computer program. For example, when we use a plus operator ("+") in a computer program, the compiler/assembler generates code that uses existing hardware on the "computer" your program will run on in order to complete the addition operation. The point is that the hardware required for the addition already exists; thus issuing some type of **add-type** instruction will not make your hardware larger or more compWhen you use an addition operator in Verilog, you've essentially make a request to the synthesizer to provide the hardware that will perform that addition for you. The point here is that it's easy to plop down a mathematical operator in our

Verilog models, but always come at a price; that price becomes higher as you use more "computationally expensive" operators[1].

10.4.2 The Magic of Using Addition and Subtraction Operators

While the title of this section sounds interesting, it brings up a few points to keep in mind. You always must know how exactly your synthesizer handles operators. For example, while it may seem comfortable to use a division operator wherever you want, you must know how the synthesizer handles creating the hardware to implement that division operation. It may be something simple, but it may be complicated thusly generating significant amount of hardware and/or slowing down the operation of your circuit. The point here is that you simply always need to know what's really going on (or at least you really should know).

One of the first things you learn in a basic digital design class is that you can use an RCA to perform both addition and subtraction. The RCA of course adds two operands and creates a result (such as A = B + C). We can then use the same hardware for subtraction by multiplying one of the operands by -1 before we perform the addition (such as A = B + (-C)). While any multiplication in hardware sounds tough, computer hardware generally uses two's complement notation to represent negative values[2]. Here are some questions that arise; I truly don't have the answer to all of these. I could figure it out if I threw a bunch of time into it, but as of this writing, I have not thrown a bunch of time at it[3].

- If my design uses both an addition and subtraction operator based on the same two operands, does the synthesizer instruct the hardware to use the same adder or does it generate separate adding and subtracting units? If the hardware uses the same adder, then it must have a way to perform a two's complement on one of the operands, which means requiring extra hardware.
- If I'm using an FPGA, does that FPGA have specialized adders on the chip that the synthesizer can call out for use in the resultant hardware?
- If my FPGA does have specialized hardware, what happens when my models request so many adders that I run out of the specialized hardware on the FPGA?

10.4.3 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, which requires the digital designer to know a few things about the synthesis process in order to ensure the synthesized circuit works properly. We currently have the ability to mix two issues: 1) the widths of value, and 2) the signedness of values. These two issues would create a huge mess if Verilog did not provide some rules to deal with these issues. Here are the general rules that Verilog uses; if you can understand these rules, you'll be a Verilog super-shero. We'll use these rules throughout examples in the remainder of this chapter.

> **<u>RULE 1:</u> Expressions containing values of different widths:** In this case, the synthesizer expands the data width of all the operands to equal the datawidth of the operand with the largest datawidth. This includes both sides of the assignment operator.
>
> **<u>RULE 2</u>: Designer's Responsibility:** The designer must allow the appropriate bit-width in the result for mathematical operation. The synthesizer assumes you know what you're doing. If the datawidth of your result operand is not sufficient to properly represent the value, the result will experience truncation in the left-most bits. Conversely, if the width of the result is larger than

---

[1] For example, a shift-left by one bit is less computationally expensive than an addition operation, which is likewise less computationally expensive than a divide operation.

[2] Recall in two's complement notation, to multiply a number by -1, we simply take the two's complement of that number. Taking the two's complement of a number involves inverting the value and adding 1, which is a fairly simple operation in hardware, particularly FPGA hardware. This is one operation where using a full adder in the LSB position of an RCA is really helpful.

[3] In general, for better or worse, we're modeling at a high level and relying on all the underlying tools to "do the right thing" for us. The right thing is to generate the circuit we're modeling and to do so in a relatively efficient manner.

required, the extra bits filled in (the current value is expanded) based on the sign of the result (see RULE 3).

RULE 3: The Signedness of Numbers Direct Operand Expansion: The synthesizer expands values by sign-extending signed values and zero-extending unsigned values.

RULE 4: Signed vs. Unsigned Operations: Every operand in an expression on the right side of the assignment operator must be signed in order for the operation to be considered signed.

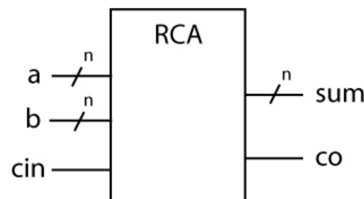
To be clear, there are two special issues we need to deal with. Note this is confusing and you may never need to deal with it, but it is worth noting here in case you do need to deal with it.

1) A ramification of RULE 4 is that if any unsigned operand appears on the RHS (right hand side) of the assignment operator, this will cause all the operands in the expression to be considered unsigned. This means that the unsigned value effectively overrides the signedness of other signed operands appearing in the RHS, which effectively changes how the expansion of the operands when needed.

2) Related to the previous item, the signedness of a set of bits only affects the expansion of the set of bits (representing a number) when that set of bits needs expanding as a result of some other operand in the expression that has a larger bit-width. By the time the LHS (left hand side) of the assignment operator is assigned, all the operands in the expression have been expanded. At this point, the declared signedness of the result has no meaning. In this case, the signedness of the result only has to do with how that value is expanded (if need be) when it appears in another expression.

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

Provide a high-level Verilog model for the following n-bit RCA.



Solution: The notion here is that there are many ways to model a RCA. This problem states that we want to model the RCA on a high-level, which essentially means to use Verilog mathematical operators rather than instantiating a series of full adders. Note also that the diagram in the example shows bitwidth of "n" bits, so we know this going to be a generic RCA. Designers have the option of declaring a value for "n" when they instantiate the module. 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.
- If designers don't override the given parameter value, the data-width of the RCA defaults to eight.
- 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 while the **sum** becomes the remaining bits.
- The largest vector in the statement is the three n-bit vectors: **sum**, **a**, & **b**. The fact that we use a concatenation operator on the left side of the assignment operator does not cause the synthesizer to interpret the bitwidth of the result greater than the bitwidth of the individual operands. The synthesizer does however increase the bit-width of the **cin** input to match the bitwidth of the other operands. Because we did not specify the signedness of the **cin** input, the synthesizer assumes it is unsigned and zero extends the value to become the size of the other operands. This works because the **cin** input is either one or zero; zero-extending a one or zero results in a one and zero, respectively.
- The **co** output is zero-extended to n-bits; the synthesizer then truncates the extra to forma a one-bit signal for the result.

```
module rca_nb #(parameter n = 8) (a, b, cin, sum, co);
    input   [n-1:0] a,b;
    input   cin;
    output  reg [n-1:0] sum;
    output  reg co;

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

endmodule
```

Figure 10.3: The solution to Example 10.1.

The cool thing to note about this model is that it works flawlessly (sort of), and is not dependent on the signedness of the inputs. You may have not have thought of implementing the RCA this way, but there are some techniques you should take away from this example. Remember, the hardware does not know or care about any signedness of the operands: it simply adds the number; the 2's complement format takes care of the grunt work.

Yep, this hardware adds the values as expected, but that's not the end of the story. The results of the operation include both a **sum** and **co**, but is the **sum** correct? Is there something in the hardware that magically indicates whether the operation was valid or not? We know that the **sum** and **co** are "correct" because that is what we asked the hardware to do, but do we know whether the sum and co are valid? The RCA itself does not know the type of numbers you are adding, so it can't tell you whether the result is valid or not.

The truth is that you result is always correct, but it is thus up to the "some other entity" to discern whether the result of an addition is correct or not. This notion exists with both signed and unsigned addition, and some combination thereof. We typically use extra hardware that uses the **co** to check the validity of the results (for unsigned numbers) or checks the sign bits of the operands and result for signed numbers. The point working with Verilog and arithmetic operators is you always need to check the validity of the result, particularly when also dealing the special rules associated with working with Verilog and signed operations.

Example 10.2: n-Bit Ripple Carry Adder (RCA) with a Problem

Describe the problem with the following RCA model and provide a solution that fixes the problem.

```
module rca_nb #(parameter n = 8) (a, b, cin, sum, co);
    input   [n-1:0] a,b;
    input   cin;
    output  reg [n-1:0] sum;
    output  co;

    reg  [n:0] tmp_sum;

    always @(a, b, cin)
    begin
       tmp_sum = a + b + cin;
    end

    assign co = tmp_sum[n];     //- co is MSB of sum
    assign sum = tmp_sum[n-1:0];

endmodule
```

Solution: The problem with the model in this example is that it does not work in all situations. To be specific, it only works for unsigned operations. The problem is that the synthesizer expands each of the operands on the right side of the assignment operator to match the datawidth of the value on the left side of the assignment operator. Note from the declaration that sum is one bit wider than the **a** & **b** operands. The RCA does know whether it is adding signed data or not, but the synthesizer will expand the input operands as if they are unsigned values because that is the way the model declares them in the external interface. Because of this, the synthesizer will correctly expand unsigned values but incorrectly expand signed value, namely the **tmp_sum** result.

Be sure to recall that the model defines a module that you can instantiate in your model. While the module that instantiates this RCA may be working with signed or unsigned data, by the way we wrote this RCA model, it only works properly when the two input operands are unsigned data. This model does not know or care about the signedness of the data you connect to it.

Figure 10.4 shows one possible fix for this problem. This example fixes the problem with the Verilog expansion the operands on the right side of the assignment operator by “manually” expanding the operands as part of the model. The code in Figure 10.4 uses the concatenation operator to expand the values by appending an extra “left-most bit” to the left side of the value. In this way, we’ve manually sign extended the values by one bit, which prevents the synthesizer from zero-extending these values such that the match the datawidth of the **tmp_sum** result. The synthesizer expands the **cin** value by using zero-extension (because it is declared as an unsigned value) to match the width of the other operands, which will work because zero-extension the **cin** value remains as either a zero or a one after the expansion.

```
module rca_nb #(parameter n = 8)  (a, b, cin, sum, co);
    input   [n-1:0] a,b;
    input   cin;
    output  reg [n-1:0] sum;
    output  co;

    reg [n:0] tmp_sum; [

    always @(a, b, cin)
    begin
       tmp_sum = {a[n-1],a} + {b[n-1],b} + cin;
    end

    assign co = tmp_sum[n];          //- co is MSB of sum
    assign sum = tmp_sum[n-1:0];

endmodule
```

Figure 10.4: One possible fix to this example.

Example 10.3: n-Bit Ripple Carry Adder (RCA) With a Problem

Describe the problem with the following RCA model and provide a solution that fixes the problem.

```
module rca_nb #(parameter n = 8) (a, b, cin, sum, co);
    input  signed [n-1:0] a,b;
    input  signed cin;
    output signed reg [n-1:0] sum;
    output  co;

    reg signed [n:0] tmp_sum;

    always @(a, b, cin)
    begin
        tmp_sum = a + b + cin;
    end

    assign co = tmp_sum[n];       //- co is MSB of tmp_sum
    assign sum = tmp_sum[n-1:0];

endmodule
```

Solution: This example is similar to the previous example in that it attempts to model a generic RCA. This time, we declare the two inputs and the vector output as signed values. Because of the signed declarations on the vectored inputs, we know that the synthesizer sign-extends the value to match the datawidth of the sum output (because it has the largest datawidth of all the operands in the expression). Sign-extending values declared as signed guarantees that the expanded value has the same value as the unexpanded value based on the signed interpretation of the value. Note again here that the actual RCA hardware is not aware of signedness of the numbers it's adding; the synthesizer on the other hand, is primarily interested in presenting the appropriate values to the RCA

The problem with this example lies with how the synthesizer expands the **cin** input, which the model declares as signed. The synthesizer then uses sign-extension to expand the **cin** input to n bits. When **cin** is zero, the sign-extension is correct, but when **cin** is one, the synthesizer sign-extends to a value to "n" ones, which is effectively a -1 (negative one), and not the single-bit "1" on the **cin** input.

Figure 10.5 provides one solution to the problem in this example. In this solution, we declare the **cin** value as unsigned, which ensures the synthesizer always extends the value in such a way as the result of the expansion is either an n-bit '1' or an n-bit zero (and never all '1's). The only potential problem with this solution is that it mixes signed and unsigned values on the right side of the assignment operator; while this "works", the synthesizer will probably bark out a warning message.

```
module rca_nb #(parameter n = 8) (a, b, cin, sum, co);
    input  signed [n-1:0] a,b;
    input  cin;
    output signed reg [n-1:0] sum;
    output  co;

    reg signed [n:0] tmp_sum;

    always @(a, b, cin)
    begin
       tmp_sum = a + b + cin;
    end

    assign sum = tmp_sum[n-1:0];
    assign co  = tmp_sum[n];         //- co is MSB of tmp_sum

endmodule
```

Figure 10.5: A possible fix for this example.

Figure 10.6 shows another possible fix for this example, one in which we solve the “synthesizer warning” problem potentially present in the previous example. In this solution, **cin** remains declared as signed value, but now we use typecasting to modify how the synthesizer interprets and thus expands **cin** when it encounters **cin** in an expression when it requires expansion. This solves the mixed type warning issue. The basic approach is rather clever. Note that we know the RCA is more than 1 bit wide, so we append a “0” to the left side of the **cin** value and cast the result of the concatenation as a signed number. The synthesizer now sees a value with the left-most bit being a ‘0’, which forces the sign extension mechanism to append more 0’s and thus not change the intended value of the **cin** input.

```
module rca_nb #(parameter n = 8) (a, b, cin, sum, co);
    input  signed [n-1:0] a,b;
    input  cin;
    output signed reg [n:0] sum;
    output  co;

    reg signed [n:0] tmp_sum;

    always @(a, b, cin)
    begin
       tmp_sum = a + b + $signed({1’b0,cin});
    end

    assign sum = tmp_sum[n-1:0];
    assign  co = tmp_sum[n];       //- co is MSB of tmp_sum

endmodule
```

Figure 10.6: Yet another possible fix for this example.

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.
- The four rules for doing higher-level arithmetic in Verilog:
  - **RULE 1:** the synthesizer expands the data width of all the operands in an expression to equal the datawidth of the operand with the largest datawidth.
  - **RULE 2:** The designer must allow the appropriate bit-width in the result for mathematical operation.
  - **RULE 3:** The synthesizer expands values by sign-extending signed values and zero-extending unsigned values.
  - **RULE 4:** Every operand in an expression must be signed in order for the operation to be considered signed.
- Typecasting instructs the synthesizer how to interpret values. Typecasting can also be used to control how the synthesizer expands values in expressions.
- Verilog RCA models always generate the correct sum, but that sum may not be valid. Typically designers use the carry-out bit and a combination of sign bits to discern the validity of the results.

10.6 Chapter Exercises

1) Briefly describe why using a full adder in the LSB position of an RCA provides more flexibility.
2) Briefly explain the relationship between using a Verilog mathematical operator in a model and the resultant circuit generated by the synthesizer.
3) Briefly describe the difference between a *correct* result and a *valid* result for a given arithmetic operation.
4) Which format does Verilog use to represent signed numbers?
5) Briefly describe if it possible to have a 1-bit signed number using 2's complement notation.
6) List the two situations where we use typecasting in Verilog models.
7) Briefly describe what determines when Verilog automatically expands values.
8) Briefly describe what determines how Verilog expands numbers when it needs to.
9) What conditions needs to be in place in order for an expression to be evaluated as a signed value.
10) Briefly describe if you the Verilog modeler know any specifics about how the hardware implements mathematical operators used in a given model.
11) Briefly explain why the hardware generated from a good RCA model will always generate the correct result, but that result may not be valid.
12) Briefly explain how modelers determine the validity of the sum output of signed operations when using an RCA.
13) Briefly explain how modelers determine the validity of the sum output of unsigned operations when using an RCA.
14) Briefly explain whether RCA can have both signed and unsigned inputs and still generate the correct and valid sum.

11 D Flip-Flops and Latches

11.1 Introduction

D flip-flops and latches are the starting points for memory in digital design. While D flip-flops and latches are similar in that they are both 1-bit storage elements, they do have some significant differences. There are two main types of latches in digital design: NOR and NAND latches; there are also three main types of flip-flops: D flip-flops, T flip-flops, and JK flip-flops. This chapter only covers D flip-flops as they essentially form the basis for many of the circuits we work with in that they are both sequential and synchronous. Additionally, we deal with D flip-flops at a relatively high level, and certainly not at the gate-level. And finally, we expend a significant amount of effort preventing our circuit models from generating latches. This chapter provides the basics of sequential circuit modeling using Verilog.

The D flip-flop is essentially a 1-bit register, but we choose to 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. 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

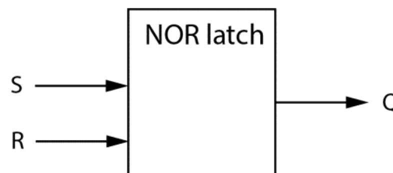
There are two distinct approaches to modeling sequential circuits using Verilog. The primary difference between these two approaches is the notion of an asynchronous vs. a synchronous sequential circuit. We consider asynchronous circuits in this text to be some type of latch[1].

11.2.1 Modeling Latches: An Asynchronous Circuit

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.

Example 11.1: Modeling a NOR-latch in Verilog.

Model the following NOR latch using Verilog. Your mode should be a behavioral model and not a gate-level design.



Solution: We assume the gentle reader knows how a NOR latch operates[2].

[1] Recall that the difference between a simple latch and a flip-flop is that the latch is asynchronous and the latch is asynchronous.

Figure 11.1 shows an example of a model of an SR latch; here are a few things to note about this model. Note that there are many different ways to model a latch using Verilog.

- 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. In this case, the sequential circuit is asynchronous.
- This is model fits the definition for a NOR latch because the **S** and **R** inputs are positive logic.
- We could have included an **`else-if`** clause that describe the hold conditions for the latch (**S**=0 & **R**=0), but we opted not to. This works because the model only includes the cases where the input conditions cause the latch to change state.
- The ordering of the two **`if`** clauses does not matter, namely, we could have checked for the set condition before the reset condition. Once any if clause evaluates as true, the **`always`** block terminates.
- The **`if`** statement only specifies two conditions, and there is no catchall. This means if one of the two conditions evaluates as true, then the code makes the associated assignment. If neither of the two conditions evaluate as true, the circuit does not make a state changes. Another way of saying this is that the circuit does not make state changes when the inputs direct the latch to hold state. Either the '1' or '0' state can be "held" by the latch; the notion of "holding" is the memory action of the latch.
- Because we are modeling a sequential circuit, we use non-blocking operators in the model's assignments.

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

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

endmodule
```

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

So, that is the story of the latch. In reality, you may never need to model a latch; you most likely be modeling synchronous sequential circuits such are register. Conversely, we expend a great deal of effort into ensuring our circuits do not model latches when we intend to model a combinatorial circuit.

11.2.2 Modeling D Flip-flops: A Synchronous Circuits

All the sequential circuits in this text are synchronous in nature, as we try to avoid working with latches in our designs. 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).

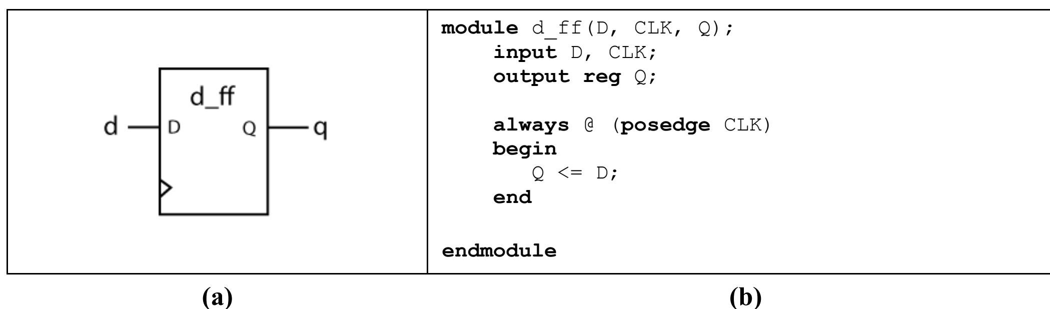
[2] A NOR latch sets when SR = "10", clears when SR = "01", holds when SR = "00", and spontaneously combusts when SR = "11".

The important item to note here is that this form of a sequential circuit is sensitive to edges of signals, otherwise referred to as *active edges*. Verilog uses two keywords to specify that the **`always`** block is sensitive to the edge of a given signal. These keywords are **`negedge`** (for negative edge sensitivity) and **`posedge`** (for positive edge sensitivity). The use of either of these Verilog keywords instructs the synthesizer to generate a synchronous sequential circuit. Specifically, when you use one of these keywords, every variable you assign in your circuit is now a register. This is a key point, particularly since the methods in Verilog to produce circuits with synchronous or asynchronous memory are distinctively different.

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.
- We declare the **Q** output as a **`reg`** because **Q** appears on the left side of an assignment operator in the body of the **`always`** block.
- The sensitivity list does not include the **D** input. The circuit is only sensitive to the active clock edge; the active clock edge causes the **D** input to latch into memory. The **Q** output indicates the value the device is currently storing. Note that this approach to modeling sequential circuits is significantly different from the approach we take to modeling combinatorial circuits using **`always`** blocks.
- The body of the **`always`** block has one non-blocking assignment statement. When we intend on synthesizing sequential circuits, we always use non-blocking assignments.
- The **D** flip-flop is a synchronous sequential circuit; the posedge in the sensitivity list ensures the synthesizer will generate a synchronous sequential circuit.




```
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).

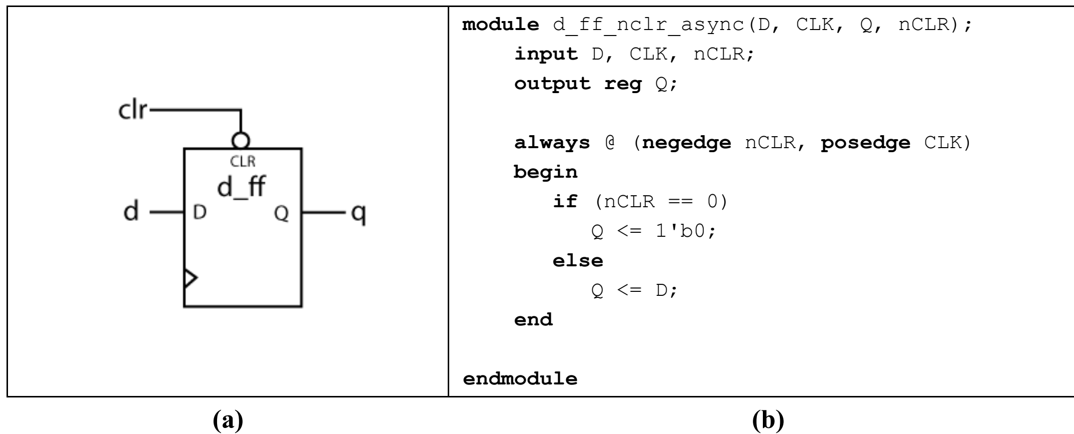
11.2.2.2 The D Flip-flop with an Asynchronous Clear

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 **`negedge`** keyword indicates a falling edge-triggered signal.
- We use the label "nCLR", which is a CLR with an n in front of it, to indicate the signal is active low, or a negative logic signal. This is an example of proper self-commenting label practice.
- We interpret the **nCLR** input 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, which is because the model evaluates the nCLR before it makes the **D** assignment to **Q**.

- 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, which is a result of the sequential nature of the statements in the body of the **always** block. Another way to look at this is that either the **nCLR** or **CLK** signal caused the process to be evaluated; if it was a falling edge on the **nCLR** signal, the value of the **nCLR** signal will be '0' and the first **if** statement will evaluate and then the **always** block terminates.
- Even though the **always** block contains an **else** statement, which is a catch-all statement, the block still generates a sequential block based on the use of the **negedge** and **posedge** keywords. One way to look at this is to the two keywords overrides the notion of a catchall statement being there, which we typically use in combinatorial models to prevent the model from generating a latch.



(a) **(b)**

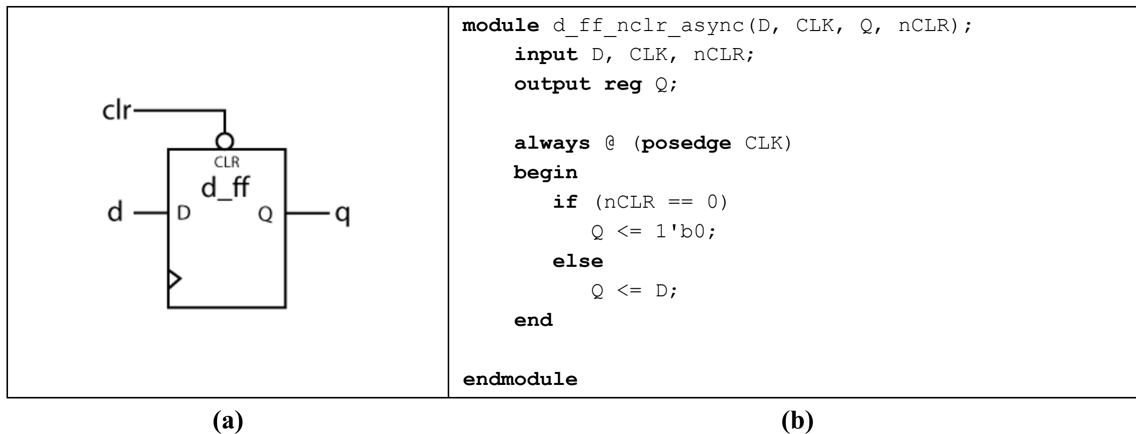
Figure 11.3: The basic D flip-flop with asynchronous active low clear BBD (a) and associated model (b).

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 we ordered the **if** and **else** statements in the code.

- 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.



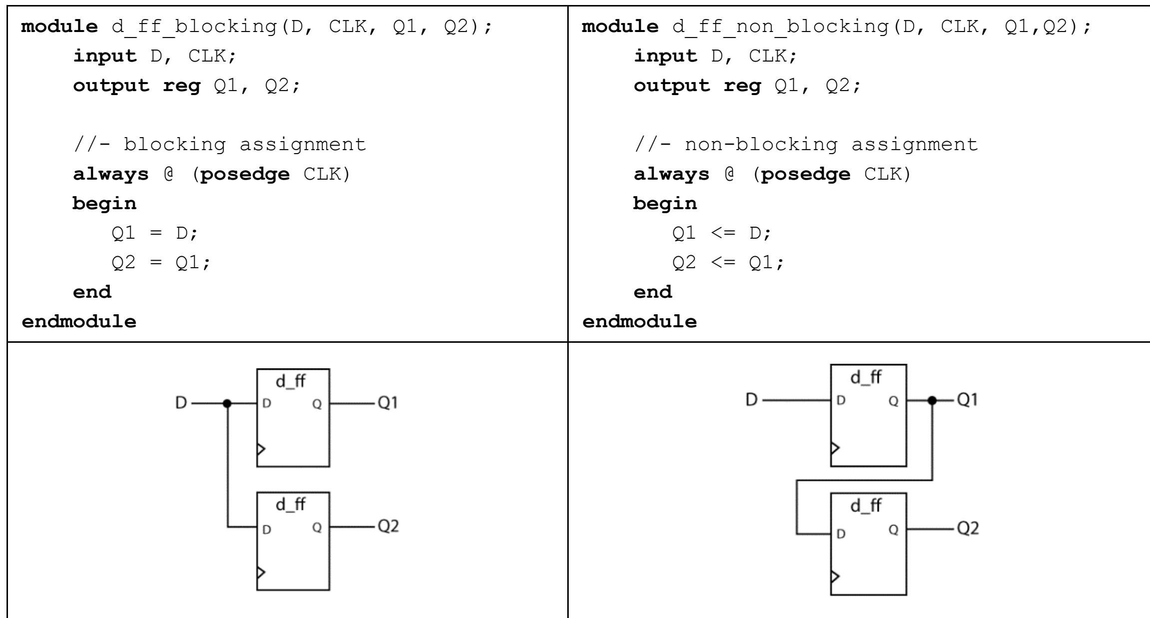
(a) **(b)**

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

11.3 Blocking vs. Non-Blocking Assignment Statements

We previously stated that we always need to 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 in its act of generating circuits. Figure 11.5 provides an example of how the synthesizer 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 modified variable value in the always block until evaluation of the **`always`** block terminates. Using this information and starting at the resultant circuit diagrams should give you a good idea as to how the synthesizer interprets blocking and non-blocking assignment statements.
- There are two ways to look at non-blocking statements. First, the model does not actually make the assignments when it encounters them in the model. The proper wording is that the assignment is “scheduled” to be made, so all references to that variable use the value of that the variable was when the **`always`** block was triggered. The ramification of this is that your sequential models can assign values as many times as they wish in the **`always`** block: the only assignment that matters is the latest one seen in the **`always`** block before the model terminates. Note that we can view the assignments in **`always`** blocks this way because we only use non-blocking statements when modeling sequential circuits.



```
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 generates.

11.4 Always Blocks for Sequential/Combinatorial Circuits

We've now used always blocks for both combinatorial and sequential circuits. We summarize the rules in this section, as we'll be using them in later chapters of this text.

`always` Block Usage	
Combinatorial Design	**Sequential Designs (synchronous)**
Sensitivity list should contain all signals accessed inside the block (but never **`posedge`** or **`negedge`**)	Sensitivity list should contain a **`posedge`** or **`negedge`**
All assignment should be ***blocking***	All assignments should be ***non-blocking***
All variables should be updated for all possible input combinations, or include a catchall statement	

Table 11.1: Overview of always blocks for combinatorial and sequential design.

11.5 Systems Verilog Considerations

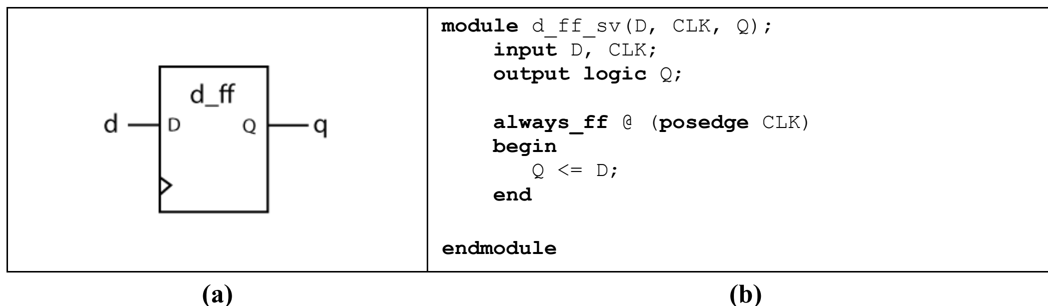
SystemVerilog introduces three new types of always block, which we sometime refer to as ***`always_type`*** blocks. The three types of blocks are 1) **`always_latch`**, 2) **`always_comb`**, and 3) **`always_ff`**. We generally don't have a great need to use the **`always_latch`** block as we rarely intentionally require latches in digital design. We've dealt with **`always_comb`** blocks in a previous chapter dealing with combinatorial circuits. As we've previously stated, you don't need to use any of these new constructs as they don't provide new "features" that we can use and they are very similar to **`always`** blocks.

The reason these ***always_type*** blocks exist is to provide a means of communication between the Verilog modeler and the synthesizer. The tendency for newbie SystemVerilog modelers is to think that using one of the ***always_type*** blocks is going to magically make your Verilog code synthesize the way you want it to. For example, using an **always_ff** block will not override the modeler's responsibility to properly construct the blocks' code. The only thing using these operators do is induce the synthesizer to generate a warning (that's a warning, and not an error) if the synthesizer feels you did something incorrect. For example, if you write your **always_comb** block in such a way that it causes the synthesizer to induce any type of memory, the synthesizer will generate a warning.

11.5.1 SystemVerilog D Flip-Flop Model

Figure 11.6 shows a SystemVerilog version of a basic D flip-flop. We refer to this as a SystemVerilog version because we include two SystemVerilog features in the model. The model in Figure 11.6(b) is the same as the Verilog version of the D flip-flop except in the following to areas:

1. We declare the **Q** output as a **logic** type. The Verilog version declared this as a **reg** type.
2. We use an **always_ff** block. The Verilog version used an **always** block. Note that unlike the **always_comb** block, the **always_ff** block requires a sensitivity list.




```
module d_ff_sv(D, CLK, Q);
    input D, CLK;
    output logic Q;

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

endmodule
```

(a) (b)

Figure 11.6: The SystemVerliog version of a basic D flip-flop BBD (a) and associated model (b).

In case you were wondering… If we don't include a type specification for the output port signals, they default to wire types when modeling in Verilog. When modeling in SystemVerilog, the signals still default to **wire**-types, and not to **logic** types as you may guess it did. This being the case, the code Figure 11.7 will not synthesize.

```
module d_ff_sv(D, CLK, Q);
    input D, CLK;
    output Q;       // must be reg or logic type

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

endmodule    // will not synthesize!!
```

Figure 11.7: A non-working example; Q defaults to a net-type, and thus can't be assigned from the procedural block.

11.6 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, which means if you place one of these keywords into a sensitivity list, all the values assigned the procedural block will be registered.
- A simple latch is a 1-bit asynchronous (level sensitive) storage element.
- The D flip-flop is the basic 1-bit synchronous storage element in digital design.
- The word latch generally has two meanings, depending on whether you use the word as a verb or a noun. As a noun, a latch refers to an asynchronous 1-bit storage element. For example: “there are two types of simple latches: NAND latches and NOR latches”. As a verb, the word latch refers to putting data in to a storage element (not necessarily a 1-bit storage element, however). For example: “the register latched the data on the rising edge”.

11.7 Chapter Exercises

1) List the two main ways modelers can create sequential circuits using Verilog.
2) Briefly describe what the term “level sensitive” refers to in the context of a simple latch.
3) List the two contextual definitions of the word “latch”.
4) Describe the similarities, if any, between the sequential nature of statements within an **`always`** block and sequential circuit.
5) Provide the Verilog model for a D flip-flop that has active low asynchronous preset and clear. The preset takes precedence over the clear.
6) Repeat the previous problem but make the clear has precedence over the preset.

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. The main issue with our D flip-flop models was that they did not contain any control inputs, an issue we solve in our discussion of basic registers. As you'll see from our generic register model, the models for D flip-flops and registers only differ in the associated data widths and control signal inputs.

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**)[1]. Table 12.1 shows a complete description of the registers input and output signals.

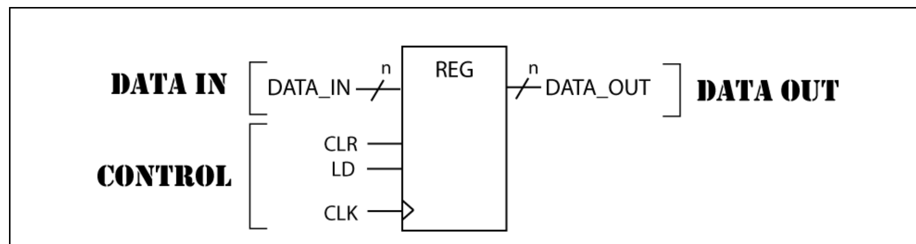


Figure 12.1: Typical data and control signals for a register.

[1] Signals such as clears can be either synchronous or asynchronous, depending on how they are modeled.

	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**	-

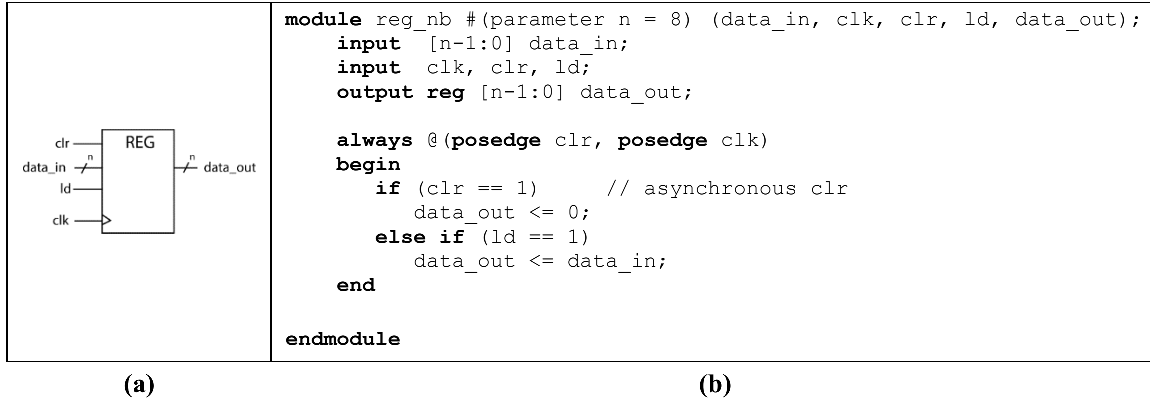
Table 12.1: The foundation description for a simple register.

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.
- The model also includes a positive logic asynchronous **clr** signal. This shows that it is possible to have a sensitivity list that contains multiple edge "triggers"; have the **`always`** block active to both a **`posedge`** and **`negedge`** on the same signal would generate a synthesis error[2].
- By the way we designed the module, the **clr** input has precedence over the **ld** input because the **clr** signal appears first in the **`if`** statement. Recall that Verilog analyzers **`if`** clauses in the order they appear in the **`if`** statement.

[2] Good to note here that any signal can't be sensitive to both the negative and positive edge of the same signal.




```
module reg_nb #(parameter n = 8) (data_in, clk, clr, ld, data_out);
    input  [n-1:0] data_in;
    input  clk, clr, ld;
    output reg [n-1:0] data_out;

    always @(posedge clr, posedge clk)
    begin
       if (clr == 1)      // asynchronous 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).

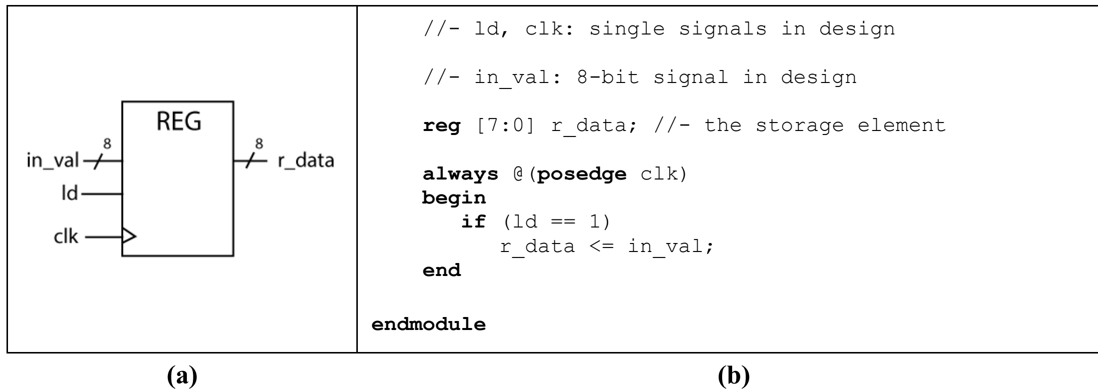
12.4 Registers vs. `reg`-Type Variables

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 the code assigns it 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.



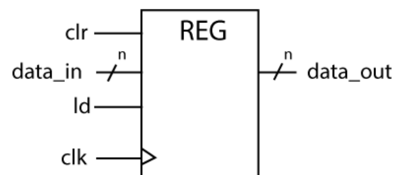

```
//- ld, clk: single signals in design

//- 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).

Example 12.1: Modeling a Register in Verilog.

Change the basis register model to make clear signal both synchronous and active low. Ensure that the **clr** signal has precedence over the **ld** signal.



Solution: We include this example to show you how straightforward it is to change some of the basic operating characteristics of register model. We made two modifications to complete this problem, and make some other comments as well.

- We removed the reference to the **clr** signal from the **`always`** block sensitivity list. The model is now only responsive to the positive edge of the **clk** signal.
- We changed the **`if`** statement to test for the clr signal to be low rather than high.
- The ordering of the clauses in the **`if`** statement definitely matters. The problem requested that we make **clr** have precedence over the **ld** signal; we do this by placing the test for the **clr** signal before the test for the **ld** signal in the **`if`** statement. In this way, if the **clr** signal is asserted, the register will clear, otherwise the model will go on to test the **ld** signal. The ramifications of this are that if **clr** and **ld** are both asserted when an active clock edge appears, the device will clear.

```
module reg_nb #(parameter n = 8) (data_in, clk, clr, ld, data_out);
    input  [n-1:0] data_in;
    input  clk, clr, ld;
    output reg [n-1:0] data_out;

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


endmodule
```

Figure 12.4: Typical data and control signals for a register.

12.6 SystemVerilog Considerations

One known issue with Verilog is its use of the **reg**-type variable. The natural inclination is to think that because you declare a variable as a **reg**, the synthesizer automatically associates storage capabilities (make it a register) with that variable, which is of course not the case. SystemVerilog addresses this problem by introducing the **logic**-type variable, which modelers can use in place of ether a **wire**-type or a **reg**-type. Using the SystemVerilog **logic**-type does have this definite advantage, as well as some other advantages that are beyond the scope of this text.

Figure 12.5 shows the generic register model of Figure 12.2(a) written using SystemVerilog. The only modification we made was to declare all the signals in the internal interface as **logic**-types.

```
module reg_nb_sv #(parameter n = 8) (data_in, clk, clr, ld, data_out);
    input  logic [n-1:0] data_in;
    input  logic clk, clr, ld;
    output logic [n-1:0] data_out;

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

endmodule
```

Figure 12.5: Typical data and control signals for a register.

12.7 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.
- Typical register control inputs include clears, resets, and loads, which modelers can model as synchronous or asynchronous. Additionally, the method by which you structure your model also determines the precedence order of control inputs.
- Modeling registers in Verilog is similar to modeling D flip-flops, as the D flip-flop is essentially a 1-bit register.
- Declaring a port of variable as a reg-type does not ensure it will be able to store data. The way modelers use the variable in their Verilog code determines a variable’s storage capability.
- Modelers can define registers as separate modules modeled in an “inline” manner as part of the internal code of a module.

12.8 Chapter Exercises

1) List the various types of control inputs a typical register can have.
2) Briefly explain whether you can know if register control inputs such as clr, ld, preset, etc. are synchronous or asynchronous by looking at the schematic diagram.
3) Briefly explain whether you can know if which typical register control inputs such as clr, ld, preset, etc. have a higher precedence.
4) Briefly explain whether an **`always`** block be active to both the positive and negative edge of a signal.
5) Briefly explain whether the order in which if clauses appear in the body of an **`always`** block affect the operation of the device.

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 states relate to the state registers. Once you gain more Verilog modeling skills you may consider using a different approach to modeling FSMs.

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 to modules in the circuit that the FSM is controlling. 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).

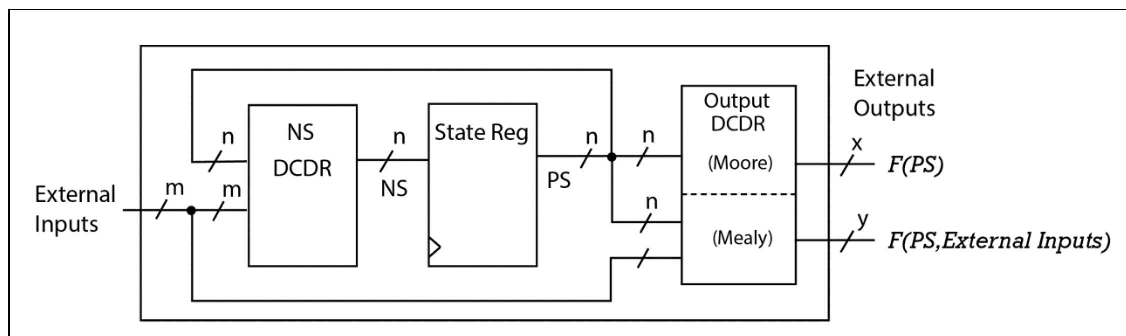


Figure 13.1: The lower-level BBD for a generic FSM.

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 form; 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 Verilog behavioral 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. What we effectively do is combine the next state and output decoders into one module: the COMB_CKT module.
- 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 next_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.

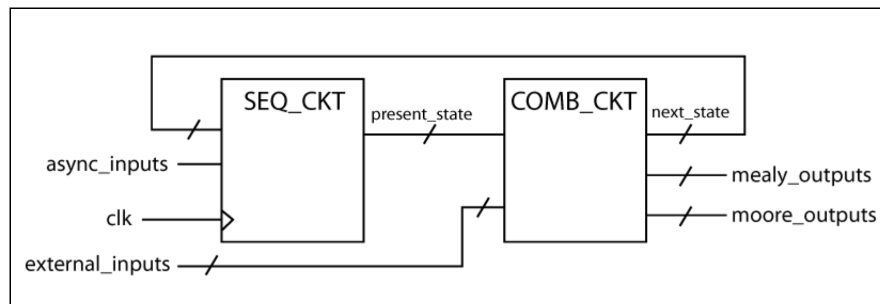
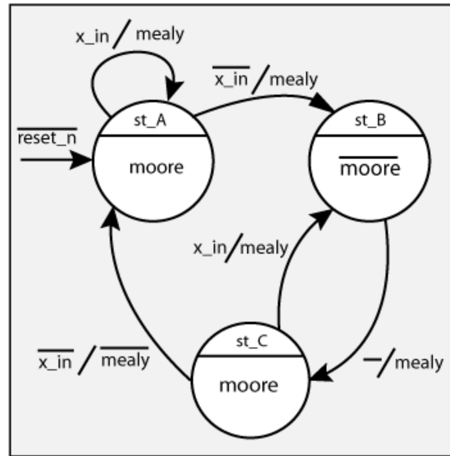


Figure 13.2: A BBD showing the Verilog behavioral modeling approach to implementing FSMs.

Example 13.1: Verilog FSM Model Implemenation #1

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



Solution: This state diagram is noticeably contrived, designed to include every possible characteristic of a state diagram. Note that from the state diagram we have both a Mealy and Moore-type outputs, an asynchronous reset input, and one external input. It never becomes 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.

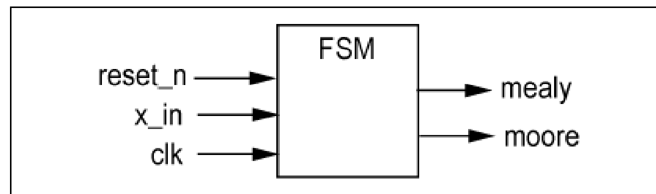


Figure 13.3: The BBD for this problem's FSM.

- The modeling approach uses two **`always`** blocks, one block is for the combinatorial process that handles issues specific to the output and next state decoders; 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 to represent the 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. Had this example actually be controlling something, we would have given the states better self-commenting names.
- 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[1] nature of the code in the body of the **always** block. All signal assignments in the **always** block are non-blocking because this block represents the state registers, which are sequential.

- The second **always** block (the combinatorial **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 combinatorial **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 and for preventing latch generation.
- 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 those assignments always appear first in the associated case clause. 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 and effectively functions like a Moore output in this state. 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 the FSM finds itself in an invalid state. Note that we didn't need to assign any outputs in the **default** clause because we previously assigned all outputs a value before the **case** statement in this **always** block.

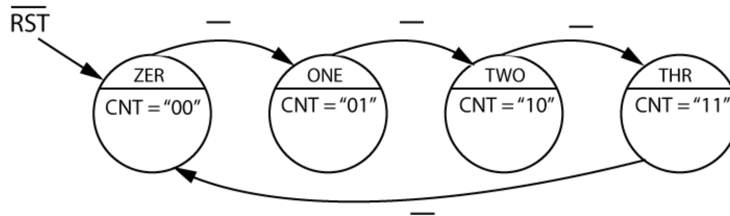
[1] This means sequential as the always block reads the statements in the order they appear; this does not mean sequential in relation to a type of digital circuit.

```
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 (defined as constants)
    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 = 1;
                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 2-bit 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.

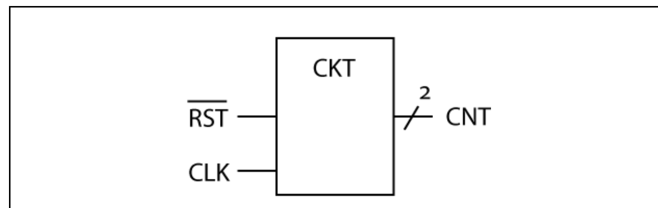


Figure 13.5: The BBD for this problem's FSM.

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

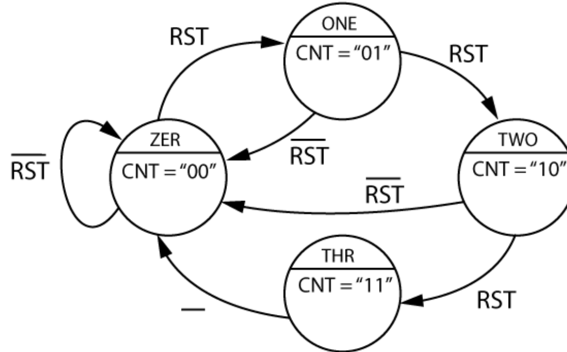
- The code in the model uses white space (blank line & indentation) to enhance readability. We adequately commented the code as well.
- All state transitions are unconditional and the output is a Moore-type, so the individual clauses in the `case` statement do not include `if-else` constructs.
- The 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 and active low. 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.

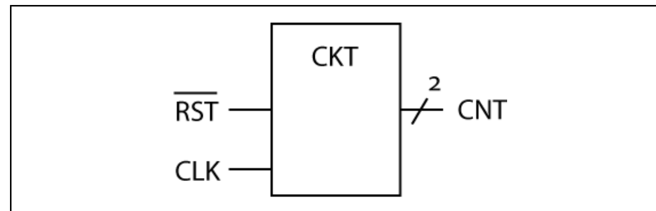


Figure 13.7: The BBD for this problem's FSM.

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:

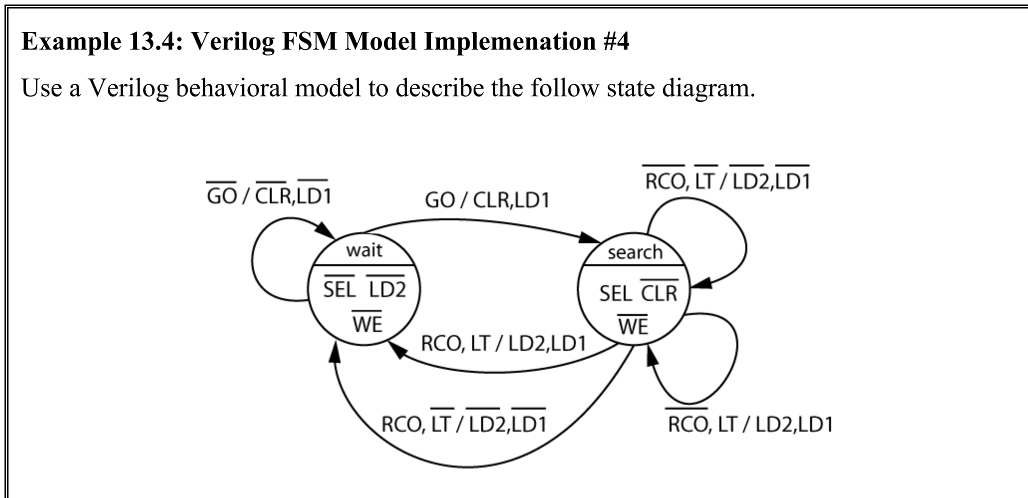
- The model using the asynchronous **RST** signal handles the signal in the sequential process while the asynchronous version handles **RST** in the combinatorial process.
- Because we now model **RST** in the combinatorial process, each of the clauses of the **case** statement now contain **`if-else`** 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)
           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.



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.

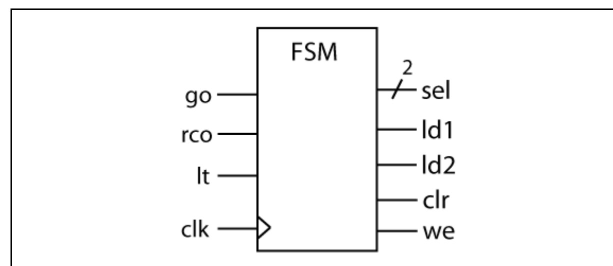


Figure 13.9: The top-level BBD for the FSM.

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;  NS = st_wait;
             end
             else
             begin
                clr = 1;  ld1 = 1;  NS = st_search;
             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   // case clause

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

Figure 13.10: The solution to Example 13.4.

13.4 Common Mistakes in Verilog FSM Modeling

There are several common mistakes people make when first using Verilog to model state diagrams. We list these mistakes in this section so people new to the process can have a checklist to help ensure their models perform as intend. One of the problems with modeling FSM using Verilog behavioral modeling is that when you make an error, it can be hard to find during the testing phase of your project. As you will probably find out, the model may synthesize correctly, but not work properly in a design.

- The number of bits for the state registers needs to be adequate to represent the number of states in the FSM.
- The bit patterns for the state registers need to be unique.
- All asynchronous signals should be associated with the sequential process.
- The **`always`** block associated with the state registers should be sensitive to the system clock edge
- All assignments associated with the state registers should be non-blocking.
- All assignments in the combinatorial process should be blocking.
- All outputs (Mealy and Moore) should be assigned before examining cases in the **`always`** block modeling the decoders.
- Each state in the FSM should be modeled as one state in a **`case`** statement; the **`case`** statement should also include a **`default`** clause, which can handle self-correction issues and ensure the **always** block is modeling a combinatorial circuit.
- Mealy outputs should be models using conditional statements within each case clause. Moore outputs should be outside the conditional statement in each case clause.

13.5 SystemVerilog Considerations

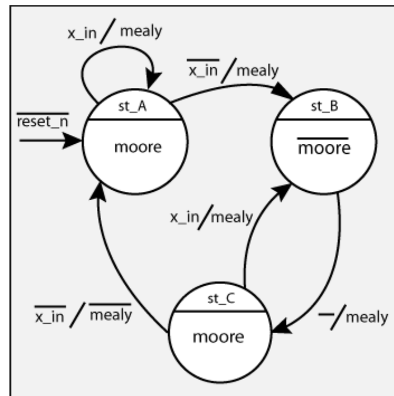
Our goal in this section is to present a few SystemVerilog that you can use in your behavior FSM models. The information presented in this section is by no means a complete discussion of the topics involved. The goal of is section is to allow you to see a few techniques you can use to encode your FSMs using SystemVerilog, while leveraging your knowledge of FSM implementation using Verilog. The best idea is to consult other SystemVerilog sources for the full story.

SystemVerilog includes the notion of enumeration types, which we can use to make Verilog FSM behavioral models more readable. Enumeration types allow the designer to specify a set of named values, which you can then effectively use as a new data type. Additionally, you can assign values to the names; the trick here is that if you don't assign values the synthesizer makes an assignment for you and you'll have to work to figure out what the synthesizer assigned.

Based on the flexibility of Verilog, you can model FSMs in many different ways. SystemVerilog offers even more options to describe your FSM. We won't attempt to show you all the options; we instead show one quite useful option that utilizes enumerated types in SystemVerilog.

Example 13.5: Verilog FSM Model Implemenation #1

Model the following state diagram using both Verilog and SystemVerilog. This is a repeat of a previous problem implemented in Verilog, so you only need to show the differences in between the Verilog and a SystemVerilog model.



Solution: Figure 13.11(a) shows the previous Verilog solution while Figure 13.11(b) shows similar SystemVerilog version. Here are the differences that you should be aware of and also a basic description of the SystemVerilog enumeration type:

- We declare the inputs and outputs of the Verilog model implicitly as wires. In the SystemVerilog version, we declare the inputs and outputs as logic types. This change was not a requirement but it makes the model seem more consistent.
- The Verilog approach uses reg-types to model the FSM's state registers on line (05). The **PS** variable forms the state registers because the always block modeling the state registers assigns the **PS** value. The **NS** value does not represent memory. The Verilog code then declares a set of constants to use as the state register values on line (07).
- The SystemVerilog code uses an enumeration type to perform the same functionality. On line (05) we use the **`typedef`** and **`enum`** keywords to create a new type; we name this type: **State_type**. The declaration of the enumerated type included the width of the data; in this case, since there were three states we needed *at least* two bits. We could have used more bits, but two bits represent the minimum number of bits we require for three states. We next declare the PS and NS variables as our newly defined type, State_type, on line (07). The issue we face here is that we can reference the states using the names we provided in the enumeration type declaration; we leave the actual bit assignments up to the synthesizer. Note we state a wider bit-width as part of the enumeration declaration.
- The code in Figure 13.11 shows the parts of the models that are different; after the initial differences, the models are identical.

(a)	(01)
	(02)
	(03)
	(04)
	(05)
	(06)
	(07)
	(08)
	(09)
	(10)
	(11)
	(12)
	(13)
	(14)
		`module fsm_template(reset_n, x_in, clk, mealy, moore);`
		`    input  reset_n, x_in, clk;`
		`    output reg mealy, moore;`
		
		`    reg [1:0] NS, PS;`
		
		`    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;`

(b)	(01)
	(02)
	(03)
	(04)
	(05)
	(06)
	(07)
	(08)
	(09)
	(10)
	(11)
	(12)
	(13)
	(14)
		`module fsm_sv_template(reset_n, x_in, clk, mealy, moore);`
		`    input  logic reset_n, x_in, clk;`
		`    output logic mealy, moore;`
		
		`    typedef enum logic [1:0] {st_A, st_B, st_C} State_type;`
		
		`    State_type PS, NS;`
		
		`    //- model the state registers`
		`    always @ (negedge reset_n, posedge clk)`
		`       if (reset_n == 0)`
		`          PS <= st_A;`
		`       else`
		`          PS <= NS;`

Figure 13.11: The differences between the Verilog (a) and SystemVerilog (b) templates for a behaviorally modeled FSM.

Using the enumeration type, we can specify the underlying bit patterns for the individual states. **Figure 13.12** shows the model of Figure 13.11(b) specifying but specifies the underlying bit patterns associated the states.

```
module fsm_sv_template(reset_n, x_in, clk, mealy, moore);
    input  logic reset_n, x_in, clk;
    output logic mealy, moore;

    typedef enum logic [1:0] {st_A=2’b00, st_B=2’b01, st_C=2’b11} State_type;

    State_type PS, NS;

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

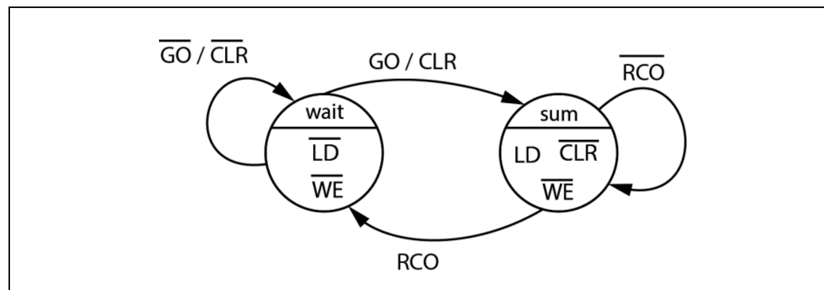
Figure 13.12: A partial SystemVerilog FSM model that specifies underlying bit patterns for states.

13.6 Chapter Summary

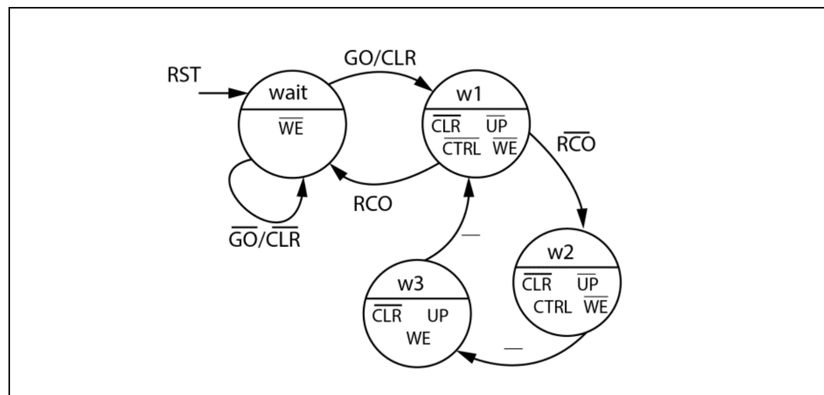
- Many different technical fields use FSMs to model the operations of processes; digital design primarily uses FSMs as digital circuits that control other digital circuits.
- The engineering step in FSM design is the creation of the state diagram; most everything else beyond that point is grunt-work 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.

13.7 Chapter Exercises

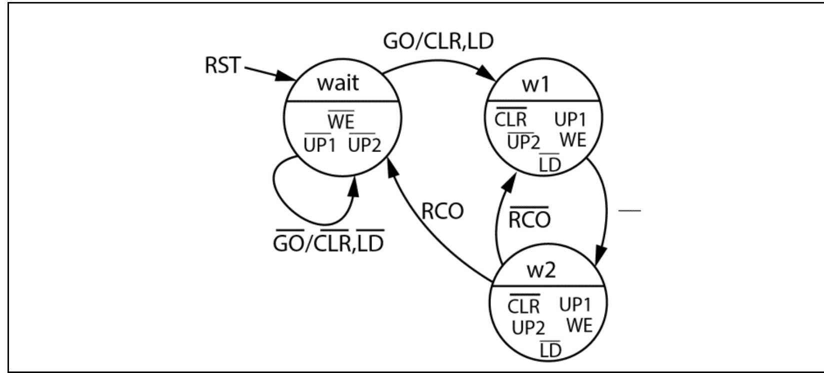
1) List the three different parts in a finite state machine.
2) Briefly describe the differences between the two types of outputs in a FSM.
3) Briefly describe the most efficient way for humans to describe and understand the operation of FSMs.
4) Briefly explain the function of an **`if`** statement within a state clause for behaviorally modeled FSMs.
5) Briefly explain how we make FSM self-correcting when modeling them with Verilog.
6) Briefly explain it is good practice to assign all outputs the state of the combinatorial process in an FSM.
7) Briefly describe what happens in a single state we don't assign all the outputs in a behaviorally modeled FSM.
8) We can classify the outputs of an FSM as either Mealy or Moore outputs. How is it then that we can say that in a given state, a known Mealy output acts like a Moore output?
9) Briefly explain under what conditions we can model a Mealy output as a Moore output.
10) Briefly explain under what conditions we can model a Moore output a Mealy output.
11) Provide Verilog behavioral models for the following state diagram.



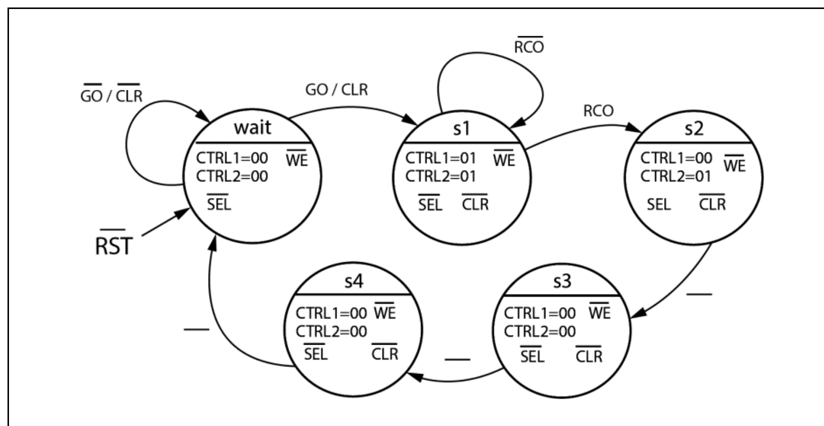
12) Provide Verilog behavioral models for the following state diagram.



13) Provide Verilog behavioral models for the following state diagram.



14) Provide Verilog behavioral models for the following state diagram.



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 counters, 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.

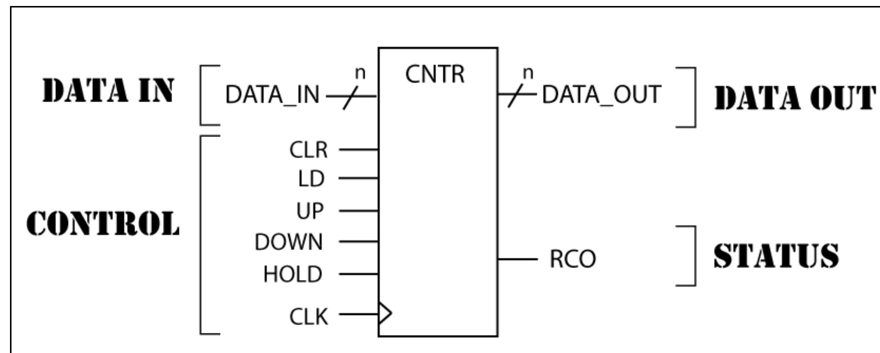


Figure 14.1: Typical data, control and status signals for a counter.

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 in Table 14.1.

	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.

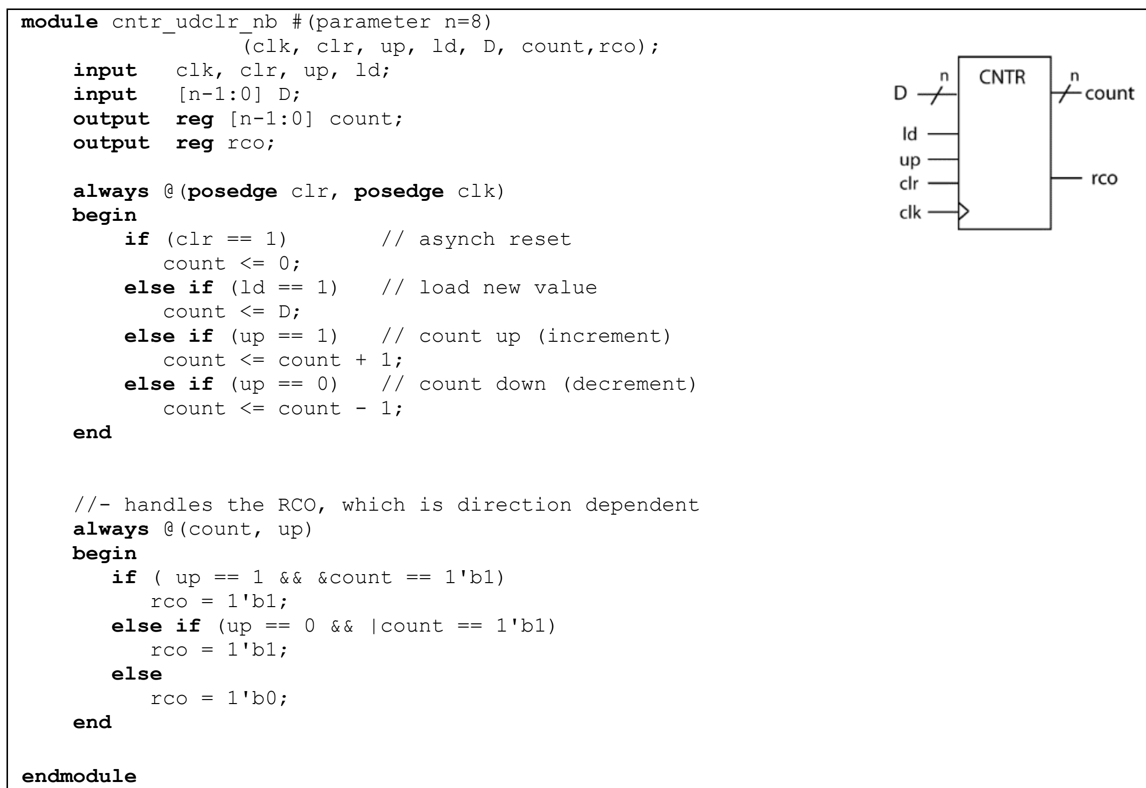
14.3 Generic N-Bit Up/Down Counter Model

Figure 14.2 shows a BBD for a generic n-bit up/down counter with asynchronous clear and 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 can override this default value when we instantiate the module.
- This counter is an up/down counter; it either counts up or down based on the value of the **up** input. We could have included a "down" input as well, but we always try to reduce the number of signals in the external interface.
- 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. We add or subtract a single bit in the assignment operators, which Verilog expands to the width of the "n"; Verilog zero-extends the value.
- The ordering of the clauses within the `if` statement in the model is important. Recall that the `if` statement evaluates the clauses in order and that once one clause evaluates as true, the `if` statement terminates further evaluations. That means by the way we wrote this model, the **ld** signal has precedence over the **up** signal, which means if the **ld** signal is asserted, the model loads and ignores the **up** input.
- This counter is inherently a binary counter. The addition and subtraction operators increment or decrement the count value by '1'. The counter automatically "rolls over" (goes from all 1's to

all 0’s) when the current count is all 1’s and the **up** input is asserted; the counter automatically “rolls under” (goes from all 0’s to all 1’s) when the current count is all 0’s and the **up** input is not asserted. This is typical binary counter operation, though you could modify the counter to do something different.

- The model includes a case for both the asserted and non-asserted **up** signal. We could have omitted the final **`if-else`** clause and inserted a catchall else statement, but we opted to list an **`if-else`** for when up is not asserted, which supports our previous notion of not relying on catchall statements for valid cases.
- 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 whether the **rco** should be asserted or not.
- The lower **`always`** block generates **rco**; note that this model is combinatorial as evident from the **`else`** statement in the **`always`** block (it does not use **`posedge`** or **`negedge`** as well). The **rco** calculation uses two applications of reduction operators in the **`if`** statement to determine if the count is all 0’s or all 1’s. This is a situation where we had to use reduction operators based on the generic nature of the circuit.

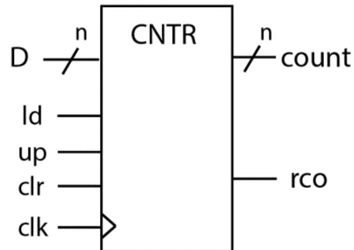



```
module cntr_udclr_nb #(parameter n=8)
                  (clk, clr, up, ld, D, count,rco);
    input   clk, clr, up, ld;
    input   [n-1:0] D;
    output  reg [n-1:0] count;
    output  reg rco;

    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.

Example 14.1: Verilog Up Counter Model

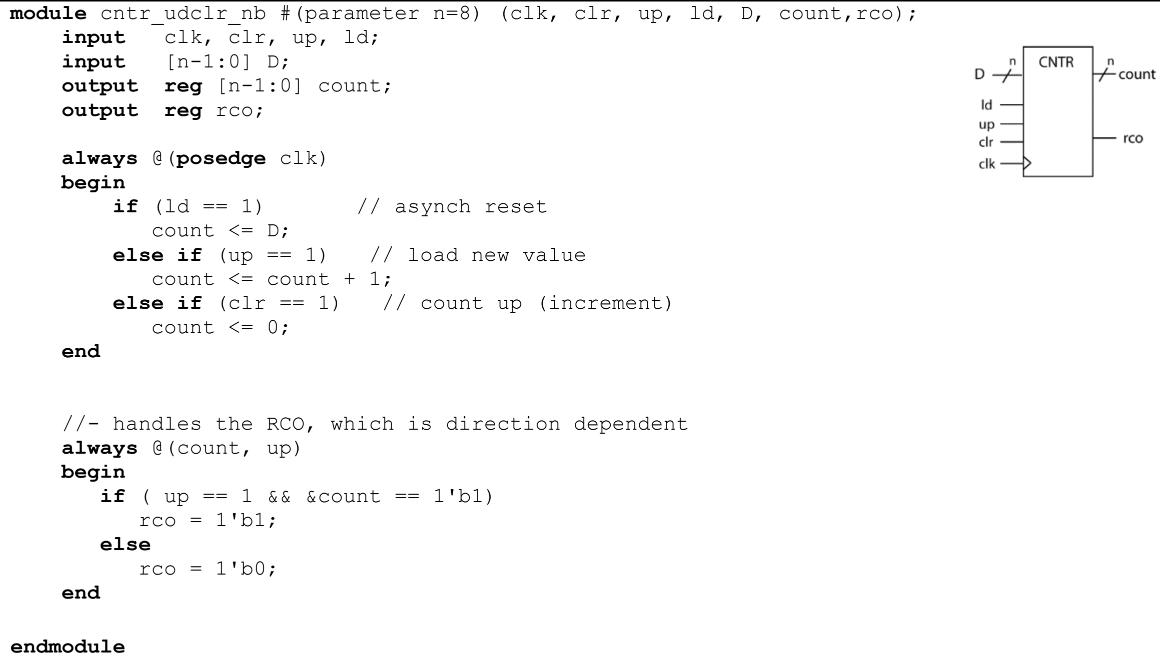
Provide a Verilog model that describes the operation of the following counter according to the included diagram. All control signals are synchronous. The **ld** signal has precedence over the **up** and **clr** signals and the **up** signal has precedence over the **clr** signal. When up signal is asserted the counter increments (according to precedence rules); otherwise the counter holds it value.



Solution: About the last thing you want to do with this example is to start modeling from scratch. This is a counter and counters are relatively straightforward to model, so we'll start with the solution from the previous example. The required modifications to the previous example are relatively few; we can look at specification for this problem and see we need to do the following; Figure 14.3 shows the resulting code.

1. Make the **clr** signal synchronous
2. Adjust the input signal precedence
3. Remove the counting down feature since this is an up counter
4. Adjust the rco to only reflect maximum value output

- We made the **clr** signal asynchronous by removing the **`posedge`** associated with the **clr** signal in the **`always`** block sensitivity list.
- We changed the ordering of the **`if`** clauses to support the new precedence called out by this problem.
- We removed any action from happening for when the up signal is not asserted by removing that particular if clause.
- Since the counter only counts up, we don't require the counting down version of the **rco**, so we remove an **`if`** statement in the combinatorial **`always`** block.




```
module cntr_udclr_nb #(parameter n=8) (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;

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


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

endmodule
```

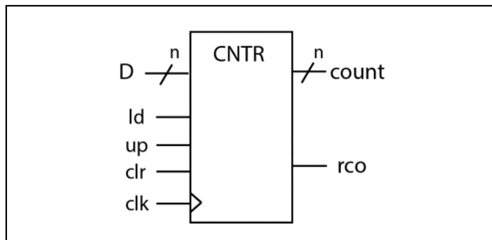
Figure 14.3: A BBD and a Verilog model for an n-bit up counter with synchronous clear.

14.4 Chapter Summary

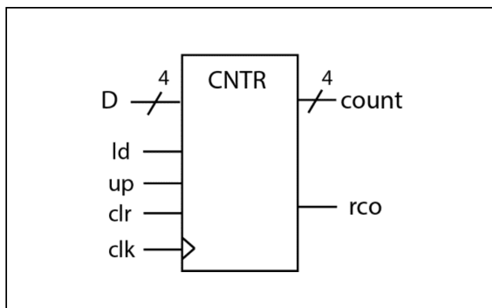
- Counters are synchronous sequential circuits; they are essentially registers with a special repeatable output sequence we refer to as the count.
- Counter generally have the feature set of register, include clear, and load inputs (depending on how you model the counter).
- The precedence of counter control inputs is determined by the ordering of clauses within the model code.
- The synchronicity of counter control inputs is determined by which signals edges the always block is sensitive to.
- Counters often include status outputs such as RCO (ripple carry out) that indicate when the counter has reached its terminal count.

14.5 Chapter Exercises

1) Briefly explain why we consider counters to be a special type of register.
2) Briefly describe the accepted relationship between counting up and down and incrementing/decrementing.
3) Briefly explain how Verilog determines the precedence of counter control inputs.
4) Briefly describe how Verilog determines which signal edges the counter's actions are synchronized to.
5) Briefly explain why the ripple carry out status signal from an up/down counter is dependent upon current count direction.
6) Provide a Verilog model that describes the operation of the following up/down counter according to the included diagram. All control signals are active high. All control signals are synchronous except for the **ld** signal, which is asynchronous. The **clr** signal has precedence over the **up** signal. The **ld** signal has precedence over the **clr** and **up** signals. When **up** signal is asserted the counter increments (according to precedence rules); otherwise the counter decrements.



7) Provide a Verilog model that describes a 4-bit decade up counter. This counter has an asynchronous clear and an RCO that indicates when the output has reached its maximum value. The **up** input serves as a hold when not asserted. The **clr** has precedence over the **ld** and up signals; the **ld** signal has precedence over the **up** signal.



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 primarily 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 circuit as well as being a controlled circuit. We typically consider all shift register operations synchronous, except for the **CLR** input, which is sometimes asynchronous. Because shift registers are straightforward to model in Verilog, we typically only include (or connect) inputs and outputs as we need them. The width of the **SEL** input needs to be sufficient to support the shift register’s set of operations. Figure 15.1 shows the foundation module for a universal shift register.

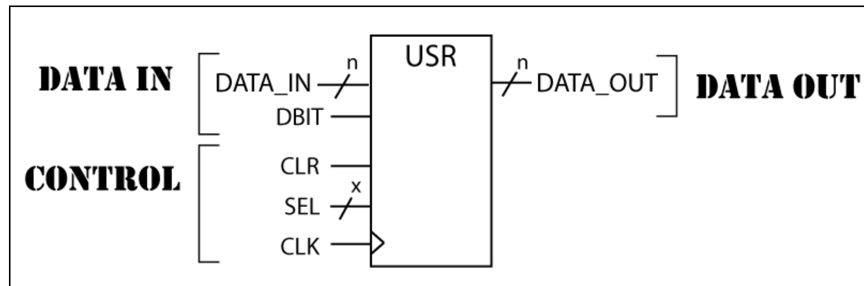


Figure 15.1: Typical data, control and status signals for a universal shift register.

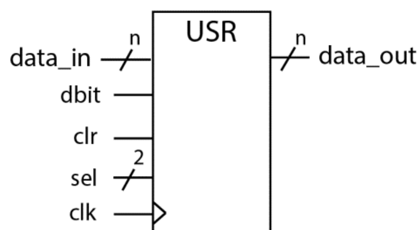
The shift register model we provide is similar to a simple register model, but includes some extra features. We only included a load, hold, shift left, and shift right in the model, but we organized the model such that it is straightforward to add other special features that our circuits may require. 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 shift can typically load data in to the device's storage elements. The DATA_IN input is the data that is loaded to the shift register.
	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 shift register'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.

Example 15.1: Universal Shift Register Model

Provide a Verilog model that describes the operation of the following universal shift register according to the included diagram. All control signals are synchronous, except for clr, which is asynchronous. The sel signal selects the operation the USR performs according to the included table. Don't use any special operators your model.



sel	operation
00	hold
01	load
10	shift left
11	shift right

Solution: There are many different ways to model a universal shift register's operations. This example states that you should not use any special operators in your design, which is a large hint that there are shift operators included in the Verilog specification. We feel that not using these operators is a useful exercise for newbie Verilog modelers; we'll use the shift operators in a later example. Figure 15.2 shows the solution to this example, which of course includes the following useful verbiage:

- The model is parameterized; this means that if we don't override the parameter when we instantiate this module, the USR default to an 8-bit datawidth.

- The model looks similar to our previous registers in that it contains an **always** block with a **posedge** or **negedge** keyword in the sensitivity list.
- The **data_out** variable represents the memory for the model because we assign **data_out** a value in the body of the **always** block that contains a **posedge** specification in the sensitivity list.
- The body of the **always** block contains an **if** statement. The **if** part of the **if** statement checks the **clr** input and clears the register is asserted; the **else** portion of the **if** statement handles the selection portion of the shift register.
- We include a clause for the hold condition in the model to make the model more readable for humans. We could have not included this case and the model would have worked.
- The shift left and shift right cases use the concatenation operator to assign the output. In both cases, the model insert the value of the **dbit** input to the new bit position in the shift register, which the left side for the right shift operation and the right side for the left shift operation. The values for the data bits is the upper "n-1" bits of the current value in the shift register for the right shift operation and the lower "n-1" bits of the current value in the shift register for the shift left operation.
- The model includes a default case for completeness; we could have omitted this **default** clause because we both provided a clause for every possible **sel** value and because the **posedge** keyword in the sensitivity list creates a register based on the **data_out** value.

```
module usr_nb #( parameter n = 8) (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;


    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 this example.

Example 15.2: Universal Shift Register Model

Redo the previous example, but use the Verilog shift left and shift right operators in your design.

Solution: The first thing to notice about this problem is that we need to use shift operators rather than concatenation for the shift operations. When you looking into how Verilog handles the details of the shift operation, you see that the left shift (“<<”) and right shift (“>>”) operators are binary operators, which means they require operands on both sides of the operator. Additionally, the specification states that Verilog fills the bit position created by the shift operation using the shift operators with a zero bit. This means the left-most bit becomes a ‘0’ using the shift right operator (“>>”) and the right-most bit becomes a ‘0’ when using the shift left operator (“<<”). Because Verilog does not give you direct control of the “shifted in” bit, we have no use the **dbit** input from the previous example. Figure 15.3 shows the new black box diagram for this example.

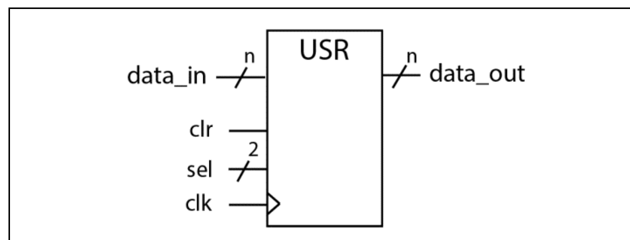


Figure 15.3: The new BBD for this example.

Figure 15.4 shows the solution to this example. As you can see, this solution is similar to the original solution, with the exception of the two lines of code associated with the shift operations (clause 2 & 3 of the **`case`** statement). In these two lines, replace the concatenation operations with shift operator. Notice that the left side of the shift operator represents the value being shifted while the right side of the shift operator represents the number of bits to shift the value by.

```
module usr_nb #( parameter n = 8) (data_in, dbit, sel, clk, clr, data_out);
    input  [n-1:0] data_in;
    input  clk, clr;
    input  [1:0] sel;
    output reg [n-1:0] data_out;


    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 << 1;  //- shift left
              3: data_out <= data_out >> 1;  //- shift right
              default: data_out <= 0;
           endcase
    end

endmodule
```

Figure 15.4: The solution to this example.

Example 15.3: Universal Shift Register Model

Redo the previous example, but use model the code such that Verilog uses a ‘1’ for the new bit.

Solution: Before we start this problem, we’ll note that we there are multiple ways of completing this example, but we only provide one of those ways. Also keep in mind that we show this problem as a demonstration of modeling techniques in Verilog; it’s not necessarily a good solution. Figure 15.5 shows the solution to this example; here is some other fun stuff to note about this solution:

- We can use the black box diagram from the previous solution.
- This solution is similar to the previous problem; all the required changes are on the lines of code that model the shifting.
- Our approach to ensuring the new bit is a ‘1’ for the left shift operator is to OR the right-most bit of the shifted result with a decimal ‘1’ before the model assigns the shift result to the internal storage. The approach for the right-shift operator is similar, but we OR the result with a ‘1’ value that we first shift left by as many bits required to have the ‘1’ bit appear in the left-most bit position. The left-shift operator in this case fills all the other bit positions with zeros before it performs the OR operations.

```
module usr_nb #( parameter n = 8) (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;


    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 << 1) | 1;             //- shift left
              3: data_out <= (data_out >> 1) | (1 << (n-1));  //- shift right
              default: data_out <= 0;
           endcase
    end

endmodule
```

Figure 15.5: The solution to this example.

15.3 Final Notes on Verilog Shift Operators

There are two other issues we need to mention related to Verilog shift operations. Verilog does have other support for shift operators; this chapter only mentions one type. On a related note, we also need to discuss the notion of signed and unsigned numbers are they relate to shifting.

15.3.1 Shift Operators and Signedness of Numbers

The notion of signed numbers often comes up when discussing shift operations and shift operators. The truth is that the underlying hardware has no notion of whether a number is signed or not; the hardware simply shifts

stuff. The modeler in responsible for two items when using shift operators: 1) how much to shift the bits, and 2) what to put into the undefined bits produced by the shift operation. The notion of how much to shift can either be a constant or variable value and must be specified when using the given shift operator. The less apparent part of using shift operators is what to put in the undefined bits.

For the simple shift left and shift right operators ("<<" and ">>"), the synthesizer generates hardware that fills undefined bits with zero. Once again, Verilog knows nothing about the intended signedness of the values. Note because of this, if you shift a value representing a negative number (in 2's complement notion) to the right using the right-shift operator, your resultant value will no longer be interpreted as being negative if it was a negative number before the shift operation. When you use a left shift operator on signed values, you may alter the signedness of the number based on the underlying bits that move into the sign-bit position.

15.3.2 Additional Verilog Shift Operator Support.

Verilog also specifies two arithmetic shift operators. The notion of an arithmetic shift is that the shift operator does not change the sign of the number your shifting if you're interpreting the number as signed. The two arithmetic shift operators in Verilog are ">>>" for right-shifting and "<<<" for left shifting. We sometimes referred so these as the signed shift operators. Their usage is sort of tricky but attempt we sum up the operations in this section.

The two shift operators are similar in that they both retain the sign bit (left-most bit) from the pre-shifted value in the result. In this way, the sign of the result is always the same as the pre-shifted value. This is true for both the arithmetic left and right-shift operators. The differences here are a function of what the synthesizer replaces the lost bits with, which is subsequently based on the declared "signedness" of the number.

	Operator	Sign Bit	Lost Bits	Added Bits
signed	>>> (right shift)	retained	Bits starting with the right-most bit and working to left	Sign bit (left-most bit) propagates to right to fill lost bits
unsigned	>>> (right shift)	Zero propagates left to right	Bits starting with the right-most bit and working to left	Zeros propagate to right to fill lost bits
signed	<<< (left shift)	retained	Bits start with bit to the right of the left-most bit and works to right	Zeros fill lost bits for starting on right side and works left
unsigned	<<< (left shift)	Shifted left (lost)	Bits start with the left-most bit and works to right	Zeros fill lost bits for starting on right side and works left

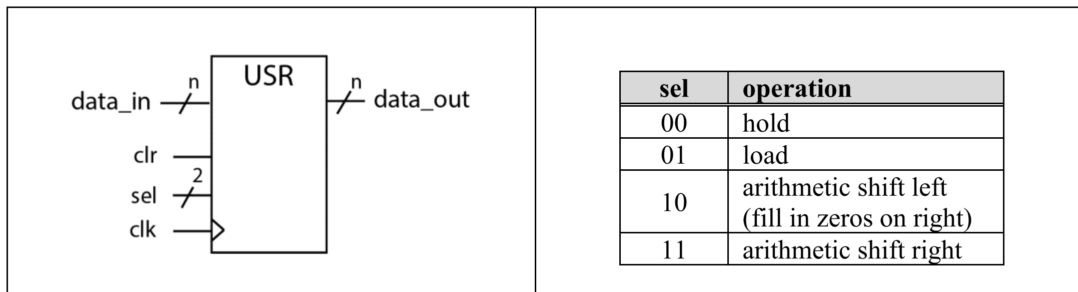
Table 15.2: A table showing how Verilog handles signed shift operators.

15.4 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.
- The Verilog specification includes several types of shift operators; these shift operators constrain the value that is insert into the bit position created by a single bit-shift.
- The Verilog specification include signed shift operators, which preserve the signedness of values and controls which numbers are placed into missing bit positions after the shift operation.

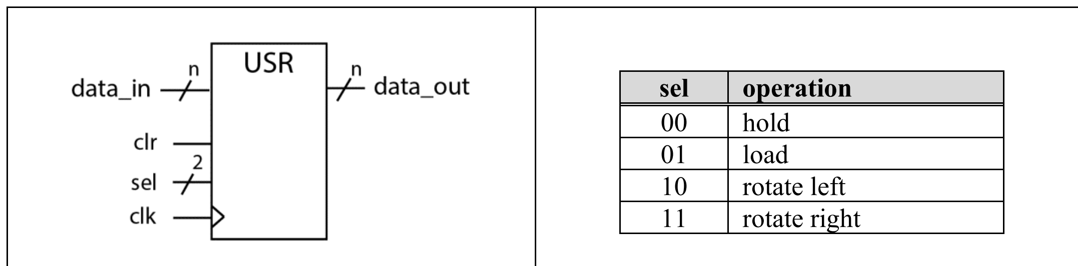
15.5 Chapter Exercises

1) Briefly explain why we often use shift registers in designs that rely on "integer math".
2) Briefly explain why using Verilog shift operators is less flexible than coding the shift operations directly using concatenation operators.
3) Briefly describe whether the basic Verilog shift operators need to know the declared signedness of numbers in order for the operator to work correctly.
4) Briefly describe whether the underlying hardware ever knows about the signedness of the numbers that it will shift.
5) Write a Verilog model for a generic n-bit shift register that does the following operations: do not use arithmetic shift operators in your solution. The clr signal is an asynchronous active high reset signal.



sel	operation
00	hold
01	load
10	arithmetic shift left (fill in zeros on right)
11	arithmetic shift right

6) Write a Verilog model for an n-bit shift register that does the following operations: use arithmetic shift operators in your solution when possible. The clr signal is an asynchronous active high reset signal.



sel	operation
00	hold
01	load
10	rotate left
11	rotate right

16 Testbenches

16.1 Introduction

Testbenches are an important part of the circuit development process using an HDL. The two main issues are 1) that we're humans, and we're going to make mistakes when we model our circuits, and, 2) 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, but you must always work with the synthesizer to obtain the circuit you.

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 generates for you. In other words, designing a circuit is half the battle, testing the circuit is the other half[1].

Designers know that many 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[2]. 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.

This chapter presents a limited view of writing and using testbenches to verify your models. In the real world, you could design a testbench that is relatively automatic, in that running the testbench will tell you all sorts of information regarding the model you're testing. Specifically, the Verilog HDL as the ability to tell you if you testbench executed without errors, but it can also tell you where the error occurred in both time and in your circuit. We'll take the non-automatic approach in this chapter, which means our testbenches will provide the inputs and run the simulation. It is then up to a human to view the simulation results to determine whether they reflect what the modeler intended or not. There are two options for generating results: output data to an external file or use the output data to generate a timing diagram.

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. However, 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 circuits. 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

[1] 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.

[2] SystemVerilog has even more constructs that are designed specifically for verification, meaning the language constructs are not synthesizable.

hardware. Once again, this text does not go there; but be prepared if your instructor or boss (or interviewer for that cool job) needs you to do some viable verification.

16.2.1 Types of Verification

There are many types of "verification" you can use to test your circuit. The two main differences are testing the circuit using idealized conditions, or testing the circuit using some form of realistic conditions. We generally have two issues we need to deal with when testing circuit, which effectively breaks the test process into two steps.

> Behavioral Simulation: The first step in verifying that our model is working as intended is focus on only the logic in the circuit. In this type of simulation, we purposely ignore real issues associated with a physical implementation of the circuit (such as time delays)[3]. The notion here is that you're circuit will never work if the underlying logic is incorrect. In general, behavioral simulations indicate whether your model's logic is correct or not.
>
> Functional Simulation: Having the correct logic is a good first step towards a working circuit, but you also need to deal with issues associated with the actual physical circuit. A physical circuit will have time delays that you ignored in the behavioral simulation. These delays are a result of such items as propagation delays through devices and delays caused by length of physical interconnects.

This text only discusses issues with generating behavioral simulations. The general thinking here is that for a beginning digital design course, we're trying to design relatively simple circuits based on the underlying logic. As such, we are generally less interested in how fast that circuit operates.

16.3 Testbenches

Test benches range from quite simple to massively complex depending on their intended purpose; the testbench 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 to the DUT and 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 an 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. Once again, we do not intend to synthesize the testbench, so we can slightly adjust our notion of modeling. Note that in Figure 16.1 there are no signals touching the dotted line, therefore the testbench itself has no external interface (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. The DUT represents the top-level in your design, which means we only need to work at that top level, as we know that the verification software (simulator) will include any layers under the top level in the DUT[4]. This becomes clearer when you see it in an example.

---

[3] Keep in mind this could be an FPGA or an actually custom silicone implementation.

[4] This is of course assuming you're testing software can find all the underlying models.

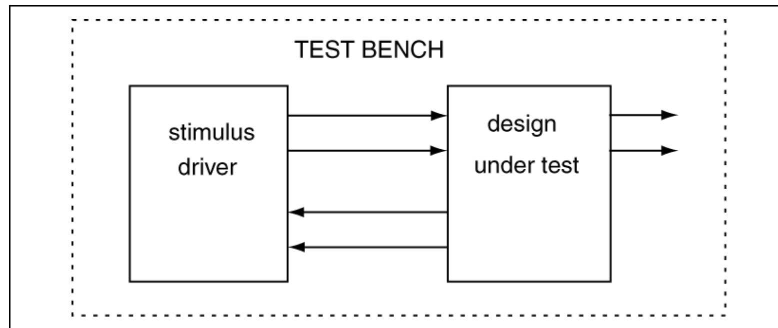


Figure 16.1: The general model of an HDL testbench.

Figure 16.2 shows a more complex testbench structure for future reference and inspiration. Here are some of the highlights of this diagram:

- The testbench can obtain test vectors from external text files. Verilog has constructs that allow for the reading of external data in a variety of formats.
- The stimulus driver can use data in external files for decision-making and/or for directly generating test vectors.
- We can view the data output from the DUT in a timing diagram or be used by other modules to determine if the DUT is working properly. You can model the testbench itself to determine if the DUT is working properly, or compare the generated outputs with expected outputs from an external file.
- The testbench can write any and all results to an external file, include wildly important data such as pass/fail information.

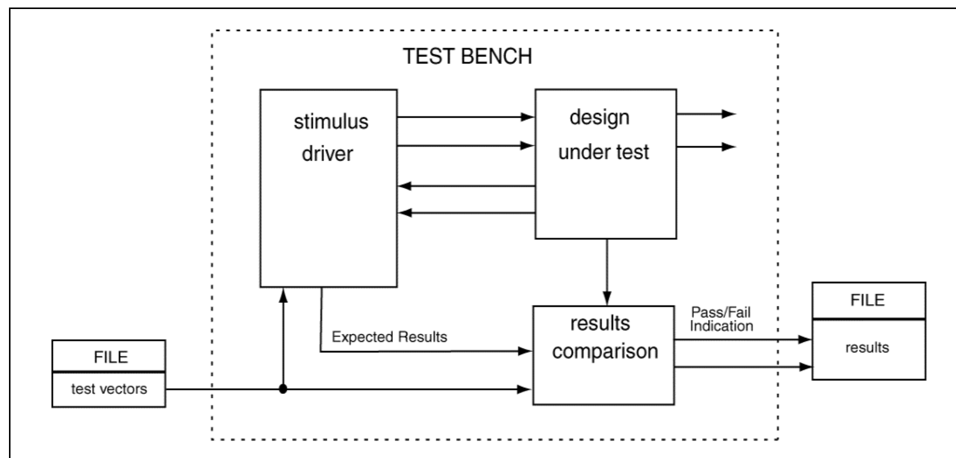


Figure 16.2: A more complex testing stucture.

16.3.1 Testbench Support Constructs

This section discusses a few new constructs associated with testbenches. Keep in mind that the testbench tests the operation of the DUT over a time span, which generally means that stimulus driver provides input the DUT at various time intervals.

16.3.1.1 `initial` Procedural Blocks

The **`initial`** block is one of two procedural blocks in Verilog. This text is most interested in synthesizable circuits, so we have up to this point only discussed **`always`** block, which is the other type of procedural block.

The **initial** block is not synthesizable and is thus primarily used for Verilog verification models. The primary use of initial blocks is to provide variables and port with values in simulation, which include both intial values and the changing of those values over time.

The **initial** block ”starts” at the beginning of a simulation, which Verilog defines as time zero. Because the **intial** block only evaluates at time zero the in the simulation, the simulator only evaluates the **intial** block once during the simulation; the simulator executes the statements it encounters in the **initial** block in sequential order. To say the **initial** block only evaluates at the start of the simulation is misleading because we can use the **initial** block to provide stimulus inputs at various times in the **intial** block before all the statements in the **initial** block are executed. Once all statements in the **initial** block execute, the **initial** block terminates.

Figure 16.3 shows the two forms of the **initial** block. The two forms differ by the number of statements included within the body of the block, where multple statements require delineation by using the **begin** and **end** keywords. Note that the **intial** blocks differ from the **always** blocks in that they can’t contain sensitivity lists.

```
initial
   signle statement


initial
begin
   multiple statements
end
```

Figure 16.3: The two forms of the initial block.

16.3.1.2 Time in SimulationLand

Your mission in the stimulus driver is to provide input to the DUT, which means that you will assign various values over a given amount of time. The general approach is to assign the inputs a value, then wait to see the affect those inputs have on the outputs. This means that there must be a way to instruct the simulator to both assign values to inputs and then delay before you change those input values to some other values.

- Verilog uses a standard delay element to indicate the passing of time in the simulator. The delay element is the “#” symbol followed by a numeric value. When the simulator encounters a delay element, the simulator waits until that amount of time passes before it changes any signal values. In this way, using delay elements allow modelers to specify the notion of testing a circuit over a given period. This provides the notion of changing stimulus signals, then waiting a period of time to verify the correct output, then changing stimulus signals to other values.
- **$finish** is a system task that instructs the simulator to end the current simulation after the last time delay in the given initial block expires. The **$finish** keyword provides a way for modelers to communicate with the simulator but is optional in testbenches. If you don’t include a **$finish**, the simulator determines the time when the simulation ends.

Both of these items are straightforward to use in testbenches, as you will surely agree when we use them in a few examples. Keep in mind that the testbenches we present in this chapter are relatively simple in that they required the modeler to explicitly specify each change in stimulus, which we can do this way because the circuits we are dealing with are relatively simple. As circuit become more complex, you need to use other “software-like” constructs to test your circuits. Additionally, Verilog, and particularly SystemVerilog, provide many other “tools” that allow you to simplify and automate verification for circuits of any complexity.

Example 16.1: Testbench Example

Provide a Verilog model and testbench for a 2-input AND gate.

Solution: We designed this problem to be as basic as possible. The first step in this solution would be to model a 2-input AND gate; Figure 16.5(a) shows the resultant model. The next step in this solution is to understand what we're trying to do at a high level. Figure 16.4 shows the high-level block diagram for this problem including both the stimulus driver and the DUT. For this problem, the stimulus driver needs to drive the inputs of the AND gate, which it does with the A and B inputs. The outputs of the stimulus driver then become the inputs the DUT. The outputs of the DUT is the output of the AND gate. Here are a few things to note before we continue with this problem.

- Drawing a diagram helps, but we often omit this step for relatively simple testbenches.
- The signal names in the stimulus driver and the corresponding names in the DUT do not have to match. It's generally a good idea to match them as a way to simplify the model and to help avoid making mistakes.

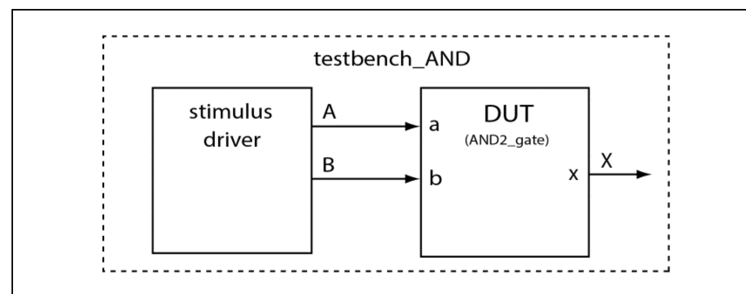


Figure 16.4: The high-level block diagram for this problem.

Figure 16.5(b) shows a simple testbench for the two input AND gate in this example. Here are the interesting things to note about this testbench.

- There is no need for an external interface for the testbench (testbench_AND), which we indicate with empty parenthesis after the module name.
- We declare intermediate signals to interact with the DUT. The DUT has two inputs and one output. We declare the inputs are reg-types because we will later assign them values from within a procedural block (the **`initial`** block). We declare the output X as a wires because the simulator is responsible for assigning X values based on the two inputs.
- We next instantiate the DUT, which is the AND2_gate. We map the devices inputs and outputs to our previously declared internal signals.
- We then define the stimulus driver using one **`initial`** block. Because there are multiple statements within the **`initial`** block, we delineate the body of the initial block using the **`begin`** and **`end`** keywords.
- The time associated with the initial block start at time zero. Note that at time zero in the initial block that we have not defined any stimulus output. The first delay element appears after the initial block begin; this time element directs the simulator to do nothing for 10 time units. After ten time units, the stimulus driver asserts the A input. Thus, the A output did not have a value until the code explicitly assigned one after 10 time units. Keep in mind that part of the simulator's job is to never make any assumption as to the value of signals; when the simulator does not know what a value is, it will indicate as much in the associated timing diagram.

- The second time delay element (#10) causes the simulator to make no change for another 10 time units. When this time expires, the code provides a value of ‘0’ to the B output.
- The third time delay element (#20) causes the simulator to make no changes for 20 time units. When this time expires, the testbench code set the B output to ‘1’.
- The fourth time delay element (#30) causes the simulator to make no changes for 30 time units. Not signals change after this time delay, but he code encounters #finish, which is a system task that the modeler can use to them the simulator to terminate the simulation.

```
module AND2_gate(
    input a,b,
    output x   );

    assign  x = a & b;

endmodule
```

```
module testbench_AND();
   //- internal interface signal
   reg  A,B;
   wire   X;

   //- instantiation of DUT
   AND2_gate  my_AND(
    .a  (A),
    .b  (B),
    .x  (X)    );

   //- stimulus driver
   initial
   begin
      #10       //- wait 10 time units from time 0
      A = 1;    //- A stimulus

      #10       //- wait 10 time units from previous delay
      B = 0;    //- B stimulus

      #20       //- wait 20 time units from previous delay
      B = 1;    // new B stimulus

      #30       //- wait 30 time units from previous delay
      $finish;
   end

endmodule
```

(a) **(b)**

Figure 16.5: A generic register BBD (a) and a fragment of it’s inline Verilog model (b).

Figure 16.6 shows an example timing diagram that could be associated with the testbench in this example. Here are the fantastically interesting things to notice about this timing diagram.

- The testbench provides the **A** and **B** inputs; the simulator provides the data for the **X** output.
- The simulation starts at time 0, which is the starting time for all **`initial`** blocks.
- The **A** signal does not have a value until 10 time units, at which time the stimulus driver sets the value to ‘1’. This timing diagram uses “??” to indicate that the simulator did not have a valid digital value and so did not know what to put in the timing diagram. The simulator you’re using will probably place a different value in the timing diagram to indicate the signal is undefined.
- The **B** signal does not have a value until 10 time units after the end of the previous delay element. Once the stimulus driver sets this input to ‘0’ after the delay, the AND gate then has two valid inputs. Once the AND gates has two valid inputs, the simulator can generate a valid out, which it does after the second 10 unit time delay. The AND gate output X at this point is ‘0’.

- The inputs remain unchanged as the code encounters a 20 time unit delay (#20), at which time the A signal transitions to '1'. At this point, the output of the AND gate (X) transitions to '1'.
- The final transition does not change during the final 30 time unit delay (#30); after that point, the simulator encounters the **`$finish`** in the code and the simulation terminates.

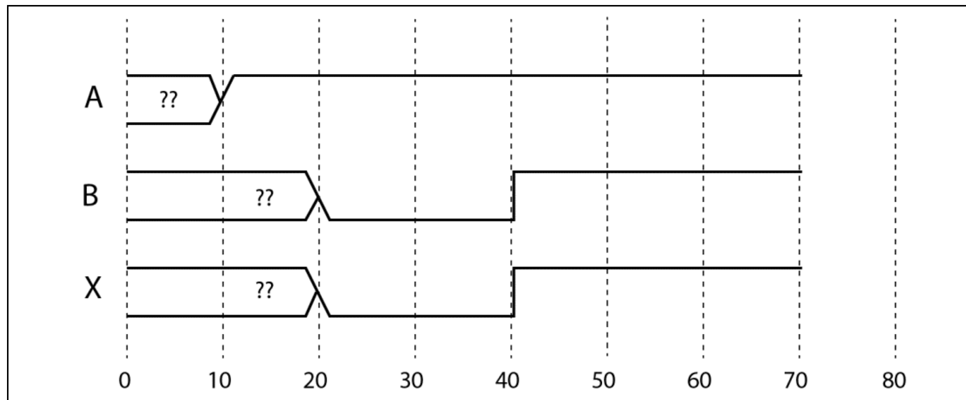


Figure 16.6: The associated timing diagram for this example.

Example 16.2: Testbench Example

Use the Verilog model of a 4-bit comparator in Figure 16.7 to generate a basic testbench model.

Solution: Figure 16.7 shows a model of a 4-bit comparator with **eq**, **lt**, & **gt** outputs. Figure 16.8 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 16.7: The Verilog model for a 4-bit comparator.

Here are the more pertinent features of the testbench model of Figure 16.8.

- 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 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 output declarations use **`wire`**-types as needed.
- The model uses an **`initial`** procedural block to generate stimulus to the DUT as a function of time.
- The initial block handling the simulation does not have a **`$finish`** statement, which means the testbench relies on the simulator to end the simulation.
- We use the "#XX delay notation 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 a time indicator, which means 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" occurs 40 time units after the initial block initially assigns the vectors.
- We assigned the first two sets of test vectors values using hexadecimal notation; we assign the final two sets of values using binary notation.
- 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 = 4'hA;
         b = 4'hB;

         //- a & b values 20 time units later
         #20 a = 4'hB;
             b = 4'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 16.8: The model of an HDL testbench for a 4-bit comparator

---

Example 16.3: Testbench Example

Use the Verilog model of an n-bit comparator in Figure 16.9(a) to generate a testbench model for a 4-bit comparator.

Solution: Figure 16.9 shows a model of a 4-bit comparator with **eq**, **lt**, & **gt** outputs. Figure 16.9 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. This examples show that you can override default parameters when you instantiate modules within testbenches.

```
module comp_nb #(parameter n=8)
               (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 = 4'hA;
      b = 4'hB;

      #20 a = 4'hB;    //- time=20
          b = 4'hB;

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

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

(b)

Figure 16.9: The Verilog model for an n-bit comparator (a) and an associated testbench (b).

16.4 Testbenches for Clocked Circuits

Testbenches for clocked circuits have one main difference from testbenches without clocked circuits: there needs to be a special stimulus block for generating a clock signal. You can do this in many different ways; this text only covers one way.

16.4.1 The `forever` Keyword

An iterative construct (a loop) is a typical feature associated with testbench. Verilog has a set of looping constructs that look very much like loops in a higher-level programming language. For the purpose of this text, we are only interested in one type of loop, which is the **`forever`** loop.

Using a **`forever`** loop in Verilog creates a block of code that runs continuously. The **`forever`** loop is not synthesizable so we use it exclusively in testbenches. The implication here is that “forever” in verification means that the code runs until the end of the simulation. Other loops in Verilog run conditionally, but the **`forever`** loop has no stopping condition.

```
forever   <statement>
```

Figure 16.10: The forever loop syntax.

The **`forever`** loop interests us in the realm of verification because it provides a simple method for generating a clock stimulus. Figure 16.11 shows an example of a forever loop as part of an initial block used as a clock-

generating signal; this code would appear as one initial block in a testbench. Here are a few things to note about Figure 16.11.

- The initial block requires **begin** and **end** keywords because there are two statements within the initial block.
- The first item in the initial block is to assign a default value to the clock. Since there are no delay statements prior to this assignment, the assignment occurs at time zero in the simulation. We typically always assign the clock a starting value because some simulators are quite fussy about not having a known starting value for signals.
- The **forever** loop contains a delay time. The notion here is that the **forever** block runs continually, but it also must implement the statement associated with the loop. The statement in the body of the forever loop effectively toggles the clock signal; the clock signal is thus toggled every five time units. Keep in mind that if the clock period is thus ten time units. Also, keep in mind that this code generates a signal with a 50% duty cycle.

```
initial
begin
   clk = 0;    //- init signal
   forever  #5 clk = ~clk;
end;
```

Figure 16.11: An example of a forever loop.

Example 16.4: Testbench Example: Generic Up/Down Counter

Use the Verilog model of an n-bit counter in Figure 16.9 to generate a basic testbench model for an 8-bit counter.

Solution: Figure 16.12 shows a model of an n-bit counter; Figure 16.14 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 16.12: The Verilog model of an n-bit up/down counter.

Figure 16.13 shows a high-level model of the testbench for this solution. The important thing to note about the model in Figure 16.13 is that the stimulus driver consists of two **initial** blocks: one for the clock signal and the other one for the remainder of the input signal. This supports the notion that you can have as many sub-stimulus drivers as you need in the testbench[5]. This mechanism supports the notion of a digital being inherently parallel in that you can test the digital circuit in a parallel manner. Additionally, using multiple **initial** blocks in your testbench provides a way to keep your testbench code organized in that you can drive similar groups of signals from a single **initial** block that is separate from other **initial** blocks.

[5] There are some rules here, such as you don't want two different stimulus drivers to simultaneously drive the same signal.

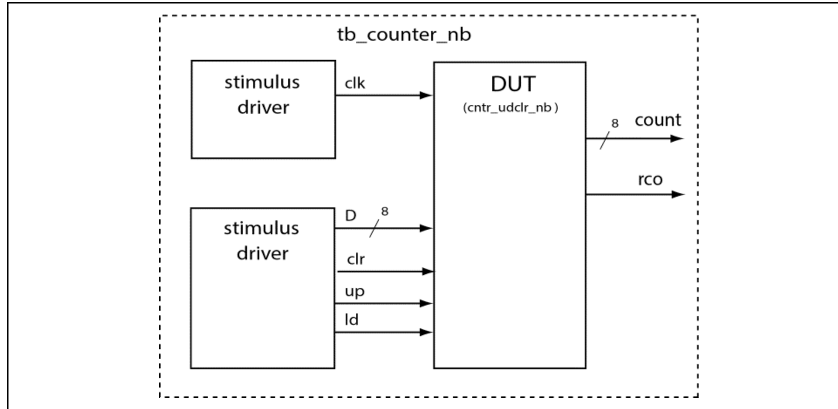


Figure 16.13: A high-level model of a Verilog testbench for the n-bit up/down counter.

Figure 16.14 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. 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_counter_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
         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 16.14: The testbench for the n-bit counter.

16.5 SystemVerilog Considerations

The main advantage that SystemVerilog has over Verilog is in the verification process. The good news is that SystemVerilog supports many constructs that are not synthesizable and are thus design to support the verification process. Many of these constructs operate similar to constructs in a higher-level programming language, which allows testbench writers to efficiently test circuits. The bad news is that this text does not cover any of these construct.

As if it's any consolation, you can use SystemVerilog to verify your circuits, which allows you to use **logic**-types rather than **wire** and **reg**-types in your design.

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

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

   initial
   begin
      a = 4'hA;            b = 4'hB;

      #20 a = 4'hB;        b = 4'hB;

      #20 a = 4'b1011;    b = 4'b0001;

      #20 a = 4'b0001;    b = 4'b0001;
      end
endmodule
```

Figure 16.15: The SystemVerilog testbench version of the testbench in Figure 16.9(b).

16.6 Chapter Summary

- Testbenches are HDL models designed to verify the correct operation of HDL models.
- The digital design process comprises of two distinct aspects: design and verification. The verification part is inherently the more challenging part, particularly in larger, more complex circuits.
- The verification process involves testing as much of a circuit as possible (coverage); the possibility of 100% coverage becomes less are circuits become more complex.
- The two main sources of errors that occur when modeling digital circuits with an HDL are that humans makes errors and that the synthesizer does things differently than modelers anticipate.
- For verification, the person writing the testbench must completely understand the circuit they are testing, and be able to write a testbench that "exercises" enough of the circuit to give a strong feeling that the circuit is working properly while not taking an excessive amount of time to test.
- The verification HDL models are significantly different from 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 are not synthesizable; 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.
- There are two main classes of verification: behavioral and functional. Behavior descriptions test whether the underlying logic is correct while ignoring all physical circuit implementation issues. Functional simulation verifies circuit after considering all physical implementation issues.

16.7 Chapter Exercises

1) What are the two main sources of errors in our HDL models?
2) Briefly explain why typical academic courses focus on design and attenuate the teaching of verification?
3) List and describe the two main types of verification.
4) Briefly explain why verification is more of an art form than a science.
5) Briefly explain why a 100% complete testing of a large and/or complex circuit is virtually impossible.
6) List and briefly describe the two main blocks in a testbench.
7) Briefly explain why it is that testbenches have no external interface.
8) Briefly explain why behavioral verification is only the first step in verifying your circuit is operating correctly.
9) List the two types Verilog's procedural blocks.
10) Briefly explain the main differences between the two types of procedural blocks in Verilog.
11) When using initial blocks in your testbench, briefly explain at what time the initial block executes.
12) If you used the following initial block to generate a clock signal, what would be the period of the clock in time units?

```
initial
begin
   clk = 0;    //- init signal
   forever  #20 clk = ~clk;
end;
```

13) What would the maximum number of initial blocks you could use in a testbench model?
14) Briefly describe one advantage to using multiple initial blocks for different signals in a single testbench model.

Appendix

Verilog Coding Guidelines

James Mealy & Paul Hummel

The guiding principle for HDL coding is to maximize the understandability of the code for other humans. Following an HDL's syntax rules ensures the synthesizer can interpret your code, but your code may not be readable by humans. Adhering to this set of guidelines enhances the readability of your code, which supports code maintainability, code reuse, and debugging.

Best Practices for Coding Verilog

Adopting an intelligent and structured approach to coding models helps you develop good overall models. You can create working models by not following these guidelines, but you'll complete your task faster if you follow them.

- Create a block level drawing first
  - Decompose the design into modules that perform a single a single function or task
  - Create buses for signals sharing a common purpose
- Write higher-level models based on pre-designed lower-level modules (when possible)
- Comment your code as you write it (see *Commenting Your Models*)
- Test your code as you develop it to expedite the debugging process
- Write models that don't use signal names indicating how higher-level modules use that module
- Design your code knowing that you'll need to later test that code (verify it works)
- Examine the elaborated design to ensure the synthesized hardware matches your expectations
  - Make sure no input signals are hanging or unconnected
  - Make sure the synthesize didn't optimize modules of signals out of the design

Naming Convention

All aspects of your code, including filenames, module names, instantiation labels, constants, and signal names should be self-commenting in nature.

- Use lowercase alpha, numeric and underscore only (except for external pins)
- Don't include direction information in signal names ("in" and "out").
- Number buses high order to low order (left to right) (ex: reg [31:0] bus_signal)
- Use all capitals letters for constants and parameter names

Code Format

In addition to following Verilog syntax rules, your code should also be readable for other humans.

- Use a courier new or similar font
- Include a standard header on all files that includes a brief but complete description of contents
- Fit code onto 72 column page (don't allow code to wrap)
- Use white space (blank lines) to separate distinct sections of code
- Use proper and consistent indentation to indicate nesting levels
- Use spaces for indentation, not tabs, to ensure printer maintains code formatting

Commenting Your Models

Comments are messages to humans who read your code. Good commenting is an indication of a good designer.

- Use comments to describe the intent or functionality of the code, not its behavior. Provide comments on the code's behavior for more complex code.
- Preface each major section with a comment describing what it does, why it exists, how it works, and assumptions the code makes on the environment.
- Use block comments for comments requiring more than one line (ex: "`/*        */`")

Model Coding Specifics

- Sequential Modules
  - Make sensitive to signal edges only (use **`posedge`** or **`negedge`** arguments in sensitivity list)
  - Use only use non-blocking assignment statements
- Combinatorial Modules
  - Make sensitive to signal levels (no posedge or negedge arguments in sensitivity list)
  - Use only blocking assignment statements
  - Use catch-all statements in procedural programming blocks (elses & defaults)
- Use one assignment statement per line in case and else statements
- Explicitly specify all value cases in code; don't rely on default for valid cases
- Use **`localparam`** for defining constants
- Use uppercase letters for all constant and parameter names

Structural Modeling Specifics

Structural modeling supports the inherent hierarchicalness of digital designs. Because hierarchical designs are easier for humans to understand, you should strive to keep your structural models as readable as possible.

- Use a vertical dot form for instantiations (not positional); use one signal name per line
- Map all inputs to a signal or a constant value
- Leave unused instantiation outputs empty (don't remove from model)

Simulation & Debugging

You'll typically need to verify your design works and/or debug it using a simulator. The debugging process is a matter of finding out what is not working, which is typically done by not considering areas of your design that are known to be working. The best approach is thus to ensure all modules work properly before including them in your design, which means that you should ideally simulate all modules first.

- The synthesizer and simulator use different Verilog "compilers", so code that "works" in one may not work in the other. Using both compilers to test your code is a good debugging approach.
- Use simulation to test and verify each sub module as you code them.
- Don't change input values on clock edges (rising or falling)
- Changes in code require relaunching the simulator; adding signals to timing diagram output does not require relaunching.

Verilog Style File

The main goal of your Verilog source code is to model a digital circuit, which means that your only requirement is to satisfy the Verilog synthesizer. However, good Verilog models must both work properly and be readable by humans; poorly written Verilog models may work properly but are not maintainable or reusable if humans can’t easily read and understand the code. Digital designers can generate good Verilog models by following a few relatively simple guidelines. The following code describes how digital designers can create nicely formatted Verilog models; you should use these guidelines in conjunction with the Verilog Coding Guidelines to help you create excellent Verilog models. Strive to make your Verilog source code neat, organized, and readable; any specific items not listed in this style file should adhere to these principles.

```
// The file contains a header describing the important features of the file.

//////////////////////////////////////////////////////////////////////////////////
// Company:  Ratner Surf Designs
// Engineer:  James Ratner & Myron Bucketts
//
// Create Date: 07/07/2018 08:05:03 AM
// Design Name:
// Module Name: prime_gen_fsm
// Project Name:
// Target Devices:
// Tool Versions:
// Description: Model contains the names of the model creator, name of
//    the module, and a description at the very least. You should also track
//    revisions of the model also. This is the standard Xilinx header which
//    contains other items we choose to remain blank.
//
// Dependencies:
//
// Revisions:
// Revision 1.00 - (07-07-2018) File Created
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

// The module name describes the module’s purpose, separates inputs and outputs,
//   and only places one item per line. We opted to include types (wire or reg),
//   but this is not necessary.
module prime_gen_fsm(
    input wire PRIME,
    input wire DONE,
    input wire RCO,
    input wire btn,
    input wire clk,

    output reg START,
    output reg WE,
    output reg UP1,
    output reg UP2,
    output reg CLR,
    output reg SEL   );

    /* Longer comments are more easily modified if you delineate them using
       block comments.
     */

    /* Note how we separate each “item” with whitespace (blink lines). We
       Do this in the entire file. While this approach makes the code model
       Longer, it does not affect the size of the synthesized hardware.
     */

    /* Note that we place all declarations at the top of the model and do
       Not inter-mingle declarations throughout the code.
     */
```

```
    // next state & present state variables
    reg [1:0] NS, PS;

    wire s_clk;   // divided (slowed) clock signal

    // bit-level state representations
    parameter [1:0] st_wait=2'b00, st_start=2'b01, st_work=2'b11;


    /* We use the vertical "dot" form for instantiations. There is one
       mapping per line, everything is nicely aligned, and we use self-
       commenting names.
     */

    // divide the FSM clock down
    clk_2n_div_test #(.n(25)) fsm_clk_divider (
        .clockin    (clk),
        .fclk_only  (1'b0),
        .clockout   (s_clk)   );

    // the state registers
    always @ (posedge s_clk)
          PS <= NS;

    // the next-state and output decoders
    always @ (*)
    begin

       // we place many assignments on the same line because they serve
       //  a similar purpose.
       START=0; WE=0; UP1=0; UP2=0; CLR=0; SEL=0; // assign all outputs

       /* All the cases in the case statement are nicely delineated.
          The if clauses in the final cases are also separated using
          whitespace (blank lines).

          All cases are represented so the case statement does not rely
          on the default clause to work properly.

          Longer blocks use comments for "ends" to indicate what they
          are ending.

          If statements with compound requirements are delineated using
          parenthesis.

          This is a combinatorial block (a decoder) so all if statements
          contain else statements, all case statements contain default
          statements, and we use block assignment statements.
        */

       case(PS)

          st_wait:  // waiting for button press
          begin
             if (btn == 0)
             begin
                CLR = 0;
                NS = st_wait;
             end

             else
             begin
                CLR = 1;
                NS = st_start;
             end
          end


          st_start:  // prepare FSM to start calculation
          begin
             START = 1; SEL = 1;
```

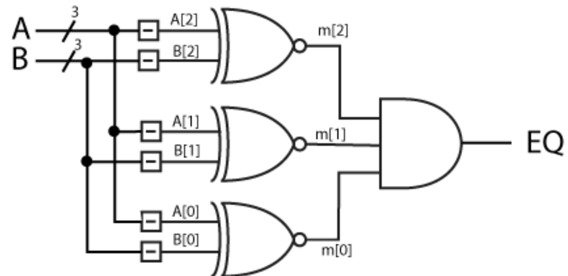
```
            NS = st_work;
         end


         st_work:  // state doing the main work
         begin
            START = 0; SEL = 1;
            if ( (RCO==1) && (PRIME==1) && (DONE==1) )
            begin
               WE = 1;
               NS = st_wait;
            end

            else if ( (DONE==1) && (PRIME==0) )
            begin
               UP1=1; UP2=0; WE=0;
               NS = st_start;
            end

            else if ( (RCO==0) && (DONE==1) && (PRIME==1) )
            begin
               UP1=1; UP2=1; WE=1;
               NS = st_start;
            end

            else if (DONE==0)
               NS = st_work;

            else
               NS = st_work;

         end   // ends current case

         default: NS = st_wait;

         endcase
      end   // ends always block
endmodule
```

Figure 0.1: Verilog style file.

CheatSheets

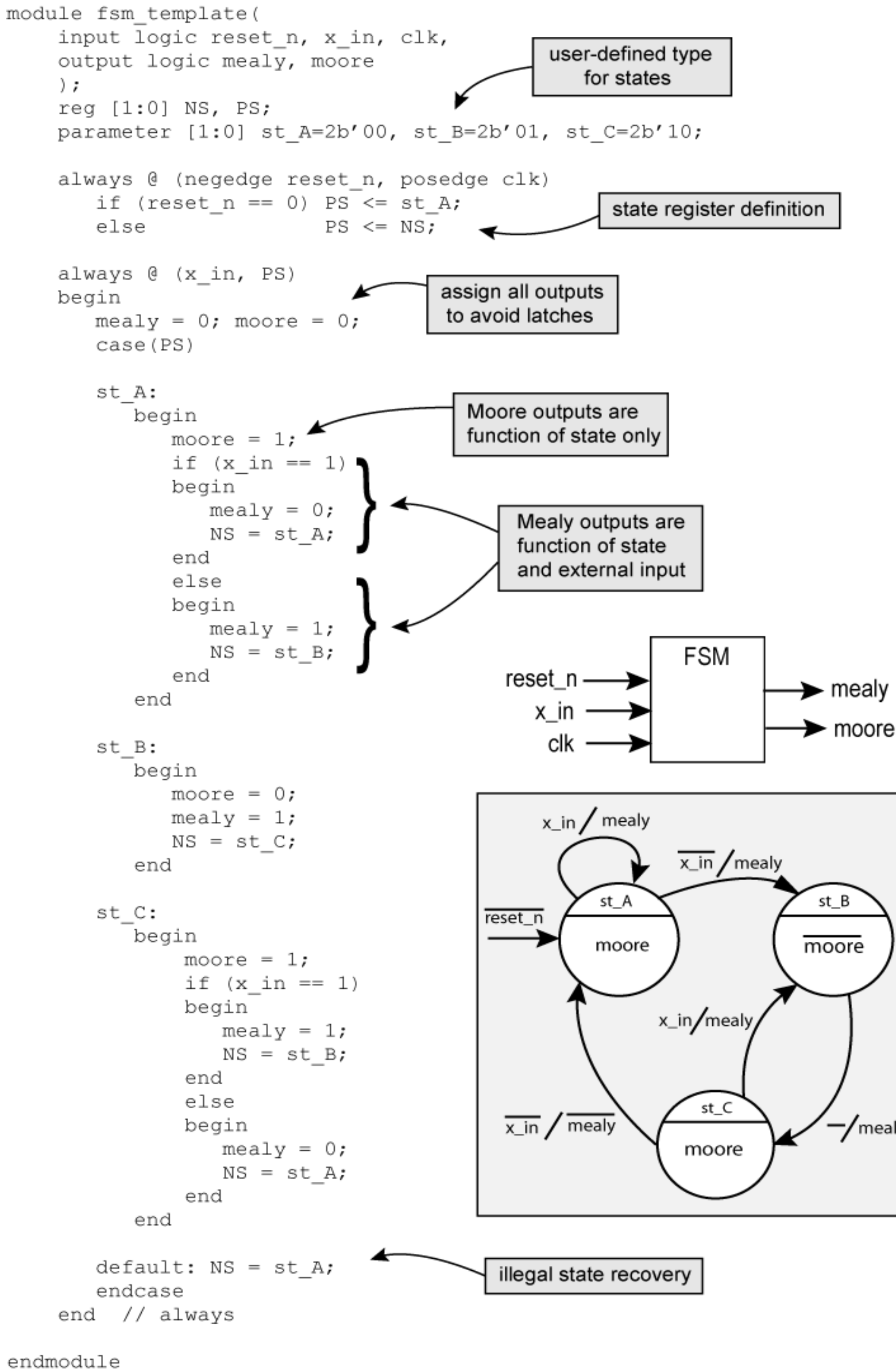
Structural Modeling CheatSheet

```
//- 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
```

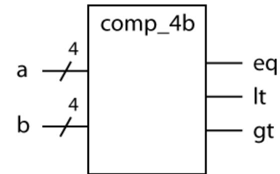
Verilog Behavioral Model Finite State Machine CheatSheet




```
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;

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

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

endmodule
```

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
```

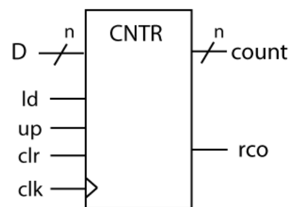



```
module tb_comp_4b(   );
   reg [3:0] a, b;       //- stimulus outputs
   wire eq, lt, gt;      //- DUT outputs

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

   initial
      begin
         //- initial values of a & b
         a = 4'hA;
         b = 4'hB;

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

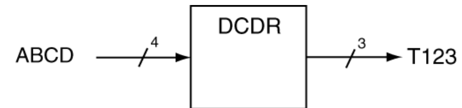
```
module tb_comp_nb(   );
   reg [7:0] D;
   reg clk, clr, up, ld;
   wire [7:0] count;
   wire rco;

   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
```



Digital Design Foundation Module Templates

Generic Decoder

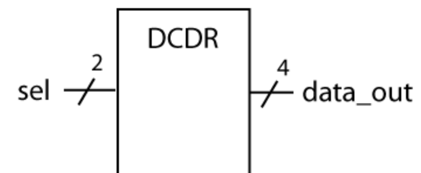
```
module dcdr_ex2a(ABCD, T123);
    input  [3:0] ABCD;
    output reg [2:0] T123;

    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

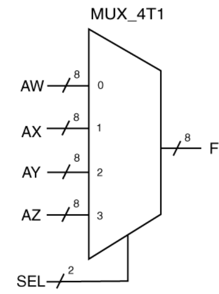
2: 4 Standard Decoder

```
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
```

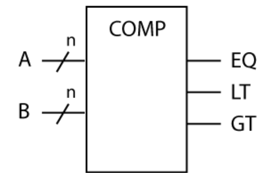


Multiplexor (MUX)

```
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
```

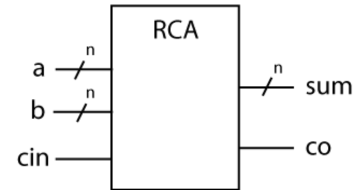


N-Bit Comparator



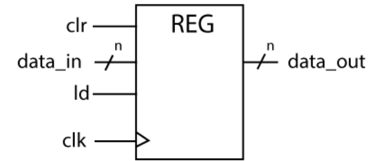

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////
// Company: Ratner Engineering
// Engineer: James Ratner
//
// Create Date: 07/04/2018 02:13:56 PM
// Design Name:
// Module Name: comp_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 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)



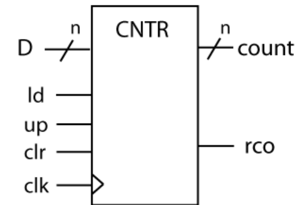

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////
// Company: Ratner Engineering
// Engineer: James Ratner
//
// Create Date: 07/04/2018 02:13:56 PM
// Design Name:
// Module Name: rca_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: n-bit RCA model
//
//   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



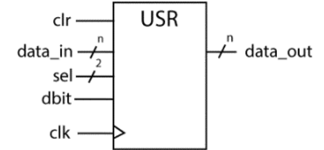

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
// Company:   Ratner Surf Designs
// Engineer:  James Ratner
//
// Create Date: 09/08/2018 07:17:37 PM
// Design Name:
// Module Name: reg_nb
// Project Name:
// Target Devices:
// Tool Versions:
// Description: 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)

See the FSM Cheatsheet

Index

A

abstracting · - 131 -
active edges · - 117 -
addition operator · - 149 -
`always` blocks · - 52 -
arithmetic operators · - 106 -
arithmetic shift operators · - 160 -
arithmetic shifts · - 155 -
assembler · - 9 -
asynchronous clear · - 124 -
asynchronous inputs · - 133 -

B

backward compatible · - 23 -
bark · - 111 -
barrel shifts · - 155 -
behavioral model · - 132 -
behavioral modeling · - 30 -, - 52 -, - 131 -
behavioral models · - 24 -, - 42 -
Behavioral Simulation · - 164 -
BFD · *See* Brute Force Design
bi-directional · - 30 -
bit-select operator · - 36 -
bitwise logic operators · - 32 -
bitwise operators · - 32 -
blank lines · - 26 -
Block Comments · - 26 -
blocked · - 54 -
blocking · - 66 -, - 119 -
blocking procedural assignment · - 54 -
blocks · - 119 -
boxes · - 30 -
BRUTE FORCE DESIGN · - 19 -
bummer · - 12 -
bundle access operators · - 97 -
bundled signal · - 70 -
bundled signals · - 35 -
busses · - 35 -

C

C programming · - 26 -, - 65 -
cascading · - 105 -
`case` · - 53 -
case sensitivity · - 25 -
cast · - 99 -
catch-all · - 67 -, - 70 -, - 72 -, - 82 -, - 115 -, - 134 -
catch-all statements · - 55 -
CIRCUIT CONTROL · - 19 -
circuit verification · - 163 -
code reuse · - 95 -
code space · - 138 -
combinatorial · - 16 -, - 24 -, - 54 -
combinatorial process · - 133 -
comparator · - 43 -, - 92 -, - 95 -
compiler · - 9 -
complete tables · - 59 -
computer aided design · - 131 -
computer peripherals. · - 20 -
concatenation operator · - 63 -, - 109 -
concurrency · - 23 -
concurrent statement · - 53 -
continuous assignment · - 31 -, - 52 -, - 70 -
control · - 17 -
controlled circuits · - 19 -
controller circuits · - 19 -
cookbook · - 132 -
count direction · - 150 -
coverage · - 163 -

D

D flip-flop · - 115 -, - 117 -
data · - 17 -, - 20 -
dataflow · - 52 -
dataflow descriptions · - 24 -
dataflow modeling · - 30 -
dataflow models · - 42 -
default statement · - 134 -
delay element · - 166 -
design under test · - 164 -
deterministic · - 134 -
Digital design · - 16 -
Digital Design Foundation Modeling · - 19 -
digital integrated circuit · - 8 -
discomfort · - 16 -
divide by two · - 155 -
down counters · - 148 -
dumbtarted · - 136 -
DUT · - 164 -, *See* design under test
duty cycle · - 173 -

E

edge-triggering · - 24 -
engineering step · - 131 -
enlightening · - 30 -
equality operators · - 69 -
explicitly mapped · - 44 -

EXTERNAL CONTROL · - 19 -
External Interface · - 30 -
extraterrestrials · - 44 -

F

field programmable gate array · - 8 -
finite state machine · - 20 -
finite state machines · - 17 -
flat design · - 11 -, - 48 -
flexibility · - 25 -
flow of data · - 16 -
following rules · - 66 -
forever · - 172 -
forever statement · - 175 -
foundation modules · - 19 -
FPGA · *See* Field Programmable Gate Array
FRDDFM · - 5 -, - 16 -, *See* FreeRange Digital Design Foundation Modeling, *See* FreeRange Digital Design Foundation Modeling, *See* FreeRange Digital Design Foundation Modeling
FreeRange Digital Design Foundation Modeling · - 5 -, - 16 -
FSMs · - 17 -, *See* finite state machines
full adder · - 105 -
Functional Simulation · - 164 -
functionally equivalent · - 23 -

G

generic decoder · - 70 -
generic decoders · - 59 -, - 60 -
generic modules · - 84 -
genericity · - 47 -
gentle reader · - 85 -
grunt work · - 9 -
gruntwork · - 132 -

H

half adder · - 105 -
hang state · - 133 -
happy ending · - 10 -
Hardware Description Language · - 8 -
Hardware Description Languages · - 5 -
HDL · - 8 -, - 23 -, *See* Hardware Description Language, *See* Hardware Description Language
hierarchical · - 10 -, - 12 -, - 13 -
Hierarchicality · - 11 -
higher-level languages · - 9 -
historical purposes · - 131 -

I

identifier · - 25 -
IEEE 1364-1995 · - 23 -
IEEE 1364-2001 · - 23 -
if-else · - 53 -
illegal state recovery · - 138 -
IMD · *See* Iterative Modular Design
implicitly mapped · - 44 -
incomplete tables · - 59 -
incompletely specified · - 93 -
initial block · - 166 -
initial blocks · - 52 -
initial procedural block · - 170 -
inline · - 126 -
instances · - 44 -
instantiate · - 44 -
instruction set architecture · - 9 -
Internal Circuitry · - 30 -
INTERNAL CONTROL · - 19 -
Internal Interface · - 30 -
ISA · *See* Instruction Set Architecture
iteration · - 52 -
iterative construct · - 172 -
ITERATIVE MODULAR DESIGN · - 19 -

J

JK flip-flop · - 115 -

K

keywords · - 26 -
knob · - 24 -, - 25 -
knobs · - 10 -, - 24 -, - 52 -, - 66 -, - 163 -
knob-tweaking · - 10 -

L

labels · - 12 -
latch · - 115 -, - 134 -
left-extending · - 99 -
level sensitive · - 122 -
look-up-table · - 59 -
looping constructs · - 172 -
LUT · - 59 -, *See* look-up-table

M

machine code · - 9 -
magic · - 10 -
magic tools · - 13 -
map · - 44 -

MD · *See* Modular Design
Mealy-type outputs · - 131 -, - 134 -
microcontroller · - 20 -
modeling hardware · - 164 -
modeling journey · - 59 -
modular · - 10 -, - 12 -, - 13 -
MODULAR DESIGN · - 19 -
modularity · - 42 -
Modularity · - 10 -
module · - 31 -, - 33 -
module parameters · - 85 -
modules · - 30 -
Moore-type outputs · - 131 -, - 134 -
multiplexor · - 81 -
multiply by two · - 155 -
MUX · - 81 -

N

name clash · - 44 -
n-bit counter · - 175 -
Neatness · - 13 -
negative edge · - 118 -
negative edge sensitivity · - 117 -
negedge · - 117 -
nets · - 33 -
newbie digital designers · - 44 -
next state decoder · - 131 -
NO CONTROL · - 19 -
non-blocking · - 66 -, - 119 -
non-blocking procedural assignment · - 54 -
noobs · - 10 -
not synthesizable · - 163 -
not-blocked · - 54 -

O

off-the-shelf · - 20 -
one-cold code · - 61 -
one-hot code · - 61 -
operator precedence rules · - 26 -
output decoder · - 131 -, - 132 -

P

parallel · - 23 -
parameter · - 85 -, - 133 -
parameterized · - 125 -, - 149 -
parenthesis · - 26 -
plagiarize yourself · - 16 -
PLD · *See* Programmable Logic Device
portability · - 9 -
posedge · - 117 -
positive edge sensitivity · - 117 -
prayer · - 10 -
precedence · - 31 -
precedence rules · - 26 -
prefix · - 126 -
procedural assignment statements · - 53 -, - 92 -
procedural blocks · - 52 -
procedural programming assignment · - 82 -
procedural programming statements · - 53 -, - 67 -
programmable logic device · - 8 -

R

RCA · - 105 -, *See* ripple carry adder
readability · - 10 -, - 140 -
redesigning the wheel · - 42 -
reduction operator · - 46 -, - 150 -
reg · - 33 -
register with features · - 155 -
registers with features · - 124 -
re-invent the wheel · - 84 -
relational operators · - 54 -, - 92 -
reserved words · - 26 -, - 27 -
ripple carry adder · - 105 -
ripple carry out · - 149 -
rotates · - 155 -

S

scalars · - 35 -
scheduled · - 54 -, - 119 -
seamless · - 11 -
selector circuit · - 81 -
self-commenting · - 25 -, - 133 -
self-commenting label · - 117 -
sensitivity list · - 65 -, - 66 -, - 82 -, - 93 -, - 125 -
sequential · - 16 -, - 24 -, - 54 -
shift operators · - 158 -
shift register · - 155 -
sign extension · - 99 -
signed shift operators · - 160 -
simple assignments · - 53 -
simultaneously · - 23 -
Single-Line Comments · - 26 -
space bar · - 26 -
specify parameters · - 85 -
specifying numbers · - 64 -
standard decoders · - 59 -
state diagram · - 131 -, - 140 -
state registers · - 131 -, - 132 -, - 133 -
statements · - 65 -
status · - 20 -
stimulus driver · - 164 -
structural modeling · - 42 -, - 52 -, - 105 -
structural models · - 42 -
structured · - 19 -
style file · - 13 -

subtraction operator · - 149 -
super-shero · - 107 -
swagger · - 12 -
synchronous load · - 124 -
syntactic constraints · - 9 -
syntactically structured · - 9 -
synthesis · - 52 -
Synthesis · - 8 -
synthesizability · - 8 -
synthesizable · - 165 -
synthesize · - 23 -
synthesizer · - 25 -
SystemVerilog · - 5 -, - 11 -, - 23 -

T

T flip-flop · - 115 -
tab key · - 26 -
terminal count · - 150 -
test vectors · - 164 -, - 170 -
testability · - 10 -
testable · - 13 -
testbench · - 164 -, - 170 -
ton of effort · - 25 -
top-level module · - 44 -
triggers · - 125 -
tweak · - 24 -
two's complement notation · - 107 -

U

unary reduction operators · - 47 -
unconditionally · - 134 -
understandability · - 10 -
universal shift register · - 155 -
unstated goal · - 13 -
up counter · - 148 -
up/down counters · - 148 -

V

variables · - 33 -
vectors · - 35 -
Venn diagram · - 59 -
verification · - 11 -, - 13 -, - 52 -, - 163 -
Verification · - 8 -
verify · - 23 -
Verilog · - 5 -
VHDL · - 5 -

W

white space · - 26 -, - 70 -
wire · - 33 -
writing software · - 163 -