Brock J. LaMeres

Quick Start Guide to Verilog

Second Edition

Springer

Quick Start Guide to Verilog

Quick Start Guide to Verilog

2nd Edition

Brock J. LaMeres

Springer

Brock J. LaMeres
Department of Electrical and Computer Engineering
Montana State University
Bozeman, MT, USA

ISBN 978-3-031-44103-5 ISBN 978-3-031-44104-2 (eBook)
https://doi.org/10.1007/978-3-031-44104-2

Cover Credit: © Carloscastilla | Dreamstime.com - Binary Code

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

The classical digital design approach (i.e., manual synthesis and minimization of logic) quickly becomes impractical as systems become more complex. This is the motivation for the modern digital design flow, which uses hardware description languages (HDL) and computer-aided synthesis/minimization to create the final circuitry. The purpose of this book is to provide a quick start guide to the Verilog language, which is one of the two most common languages used to describe logic in the modern digital design flow. This book is intended for anyone that has already learned the classical digital design approach and is ready to begin learning HDL-based design. This book is also suitable for practicing engineers that already know Verilog and need quick reference for syntax and examples of common circuits. This book assumes that the reader already understands digital logic (i.e., binary numbers, combinational and sequential logic design, finite state machines, memory, and binary arithmetic basics).

Since this book is designed to accommodate a designer that is new to Verilog, the language is presented in a manner that builds foundational knowledge first before moving into more complex topics. As such, Chaps. 1–6 provide a comprehensive explanation of the basic functionality in Verilog to model combinational and sequential logic. Chapters 7–12 focus on examples of common digital systems such as finite state machines, memory, arithmetic, and computers. For a reader that is using the book as a reference guide, it may be more practical to pull examples from Chaps. 7–12 as they use the full functionality of the language as it is assumed the reader has gained an understanding of it in Chaps. 1–6. For a Verilog novice, understanding the history and fundamentals of the language will help form a comprehensive understanding of the language; thus, it is recommended that the early chapters are covered in the sequence they are written.

The second edition of this book adds a chapter on floating-point systems. This new chapter provides a comprehensive background on the IEEE 754 standard for encoding floating-point numbers and then shows Verilog modeling approaches to implement floating-point arithmetic.

Bozeman, MT, USA — Brock J. LaMeres

Acknowledgments

For my incredible daughter Kylie. You are smart, beautiful, and creative. But even more important is that you have a sense of humor that brings joy to the world. You bring laughter to every situation and leave every room full of smiles. Never let the world make you think you should be anything other than who you are. Your strength will carry you through any hard times you encounter, and your humor will allow you to enjoy the ride. The world needs people like you more than ever. Your family will always be with you as you build the life of your dreams. With love, Dad.

Contents

Chapter 1: The Modern Digital Design Flow

The purpose of a hardware description languages (HDLs) is to describe digital circuitry using a text-based language. HDLs provide a means to describe large digital systems without the need for schematics, which can become impractical in very large designs. HDLs have evolved to support logic simulation at different levels of abstraction. This provides designers the ability to begin designing and verifying functionality of large systems at a high level of abstraction and postpone the details of the circuit implementation until later in the design cycle. This enables a top-down design approach that is scalable across different logic families. HDLs have also evolved to support automated *synthesis*, which allows the CAD tools to take a functional description of a system (e.g., a truth table) and automatically create the gate level circuitry to be implemented in real hardware. This allows designers to focus their attention on designing the behavior of a system and not spend as much time performing the formal logic synthesis steps as in the classical digital design approach.

There are two dominant hardware description languages in use today. They are VHDL and Verilog. VHDL stands for *very high-speed integrated circuit hardware description language*. Verilog is not an acronym but rather a trade name. The use of these two HDLs is split nearly equally within the digital design industry. Once one language is learned, it is simple to learn the other language, so the choice of the HDL to learn first is somewhat arbitrary. In this text, we will use Verilog to learn the concepts of an HDL. Verilog is a more lenient on its typecasting than VHDL, so it is a good platform for beginners as systems can be designed with less formality. The goal of this chapter is to provide the background and context of the modern digital design flow using an HDL-based approach.

Learning Outcomes—After completing this chapter, you will be able to:

1.1 Describe the role of hardware description languages in modern digital design
1.2 Describe the fundamentals of design abstraction in modern digital design
1.3 Describe the modern digital design flow based on hardware description languages

1.1 History of Hardware Description Languages

The invention of the integrated circuit is most commonly credited to two individuals who filed patents on different variations of the same basic concept within 6 months of each other in 1959. Jack Kilby filed the first patent on the integrated circuit in February of 1959 titled “Miniaturized Electronic Circuits” while working for *Texas Instruments*. Robert Noyce was the second to file a patent on the integrated circuit in July of 1959 titled “Semiconductor Device and Lead Structure” while at a company he cofounded called *Fairchild Semiconductor*. Kilby went on to win the Nobel Prize in Physics in 2000 for his invention, while Noyce went on to cofound *Intel Corporation* in 1968 with Gordon Moore. In 1971, Intel introduced the first single-chip microprocessor using integrated circuit technology, the *Intel 4004*. This microprocessor IC contained 2300 transistors. This series of inventions launched the semiconductor industry, which was the driving force behind the growth of Silicon Valley and led to 40 years of unprecedented advancement in technology that has impacted every aspect of the modern world.

Gordon Moore, cofounder of Intel, predicted in 1965 that the number of transistors on an integrated circuit would double every 2 years. This prediction, now known as *Moore’s Law*, has held true since the invention of the integrated circuit. As the number of transistors on an integrated circuit grew, so did the size of the design and the functionality that could be implemented. Once the first microprocessor was

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_1

invented in 1971, the capability of CAD tools increased rapidly enabling larger designs to be accomplished. These larger designs, including newer microprocessors, enabled the CAD tools to become even more sophisticated and, in turn, yield even larger designs. The rapid expansion of electronic systems based on digital integrated circuits required that different manufacturers needed to produce designs that were compatible with each other. The adoption of logic family standards helped manufacturers ensure their parts would be compatible with other manufacturers at the physical layer (e.g., voltage and current); however, one challenge that was encountered by the industry was a way to document the complex behavior of larger systems. The use of schematics to document large digital designs became too cumbersome and difficult to understand by anyone besides the designer. Word descriptions of the behavior were easier to understand, but even this form of documentation became too voluminous to be effective for the size of designs that were emerging.

In 1983, the US Department of Defense (DoD) sponsored a program to create a means to document the behavior of digital systems that could be used across all of its suppliers. This program was motivated by a lack of adequate documentation for the functionality of application-specific integrated circuits (ASICs) that were being supplied to the DoD. This lack of documentation was becoming a critical issue as ASICs would come to the end of their life cycle and need to be replaced. With the lack of a standardized documentation approach, suppliers had difficulty reproducing equivalent parts to those that had become obsolete. The DoD contracted three companies (Texas Instruments, IBM, and Intermetrics) to develop a standardized documentation tool that provided detailed information about both the interface (i.e., inputs and outputs) and the behavior of digital systems. The new tool was to be implemented in a format similar to a programming language. Due to the nature of this type of language-based tool, it was a natural extension of the original project scope to include the ability to *simulate* the behavior of a digital system. The simulation capability was desired to span multiple levels of abstraction to provide maximum flexibility. In 1985, the first version of this tool, called VHDL, was released. In order to gain widespread adoption and ensure consistency of use across the industry, VHDL was turned over to the *Institute of Electrical and Electronic Engineers* (IEEE) for standardization. IEEE is a professional association that defines a broad range of open technology standards. In 1987, IEEE released the first industry standard version of VHDL. The release was titled IEEE 1076-1987. Feedback from the initial version resulted in a major revision of the standard in 1993 titled IEEE 1076-1993. While many minor revisions have been made to the 1993 release, the 1076-1993 standard contains the vast majority of VHDL functionality in use today. The most recent VHDL standard is IEEE 1076-2008.

Also in 1983, the Verilog HDL was developed by *Automated Integrated Design Systems* as a logic simulation language. The development of Verilog took place completely independent from the VHDL project. Automated Integrated Design Systems (renamed *Gateway Design Automation* in 1985) was acquired by CAD tool vendor *Cadence Design Systems* in 1990. In response to the popularity of Verilog's intuitive programming and superior simulation support, and also to stay competitive with the emerging VHDL standard, Cadence made the Verilog HDL open to the public. IEEE once again developed the open standard for this HDL, and in 1995 released the Verilog standard titled IEEE 1364-1995. This release has undergone numerous revisions with the most significant occurring in 2001. It is common to refer to the major releases as "Verilog 1995" and "Verilog 2001" instead of their official standard numbers.

The development of CAD tools to accomplish automated logic synthesis can be dated back to the 1970s when IBM began developing a series of practical synthesis engines that were used in the design of their mainframe computers; however, the main advancement in logic synthesis came with the founding of a company called *Synopsis* in 1986. Synopsis was the first company to focus on logic synthesis directly from HDLs. This was a major contribution because designers were already using HDLs to describe and simulate their digital systems, and now logic synthesis became integrated in the same design flow. Due to the complexity of synthesizing highly abstract functional descriptions, only lower levels of abstraction that were thoroughly elaborated were initially able to be synthesized. As CAD tool

capability evolved, synthesis of higher levels of abstraction became possible, but even today not all functionality that can be described in an HDL can be synthesized.

The history of HDLs, their standardization, and the creation of the associated logic synthesis tools are key to understanding the use and limitations of HDLs. HDLs were originally designed for documentation and behavioral simulation. Logic synthesis tools were developed independently and modified later to work with HDLs. This history provides some background into the most common pitfalls that beginning digital designers encounter, that being that mostly any type of behavior can be described and simulated in an HDL, but only a subset of well-described functionality can be synthesized. Beginning digital designers are often plagued by issues related to designs that simulate perfectly but that will not synthesize correctly. In this book, an effort is made to introduce Verilog at a level that provides a reasonable amount of abstraction while preserving the ability to be synthesized. Figure 1.1 shows a timeline of some of the major technology milestones that have occurred in the past 150 years in the field of digital logic and HDLs.

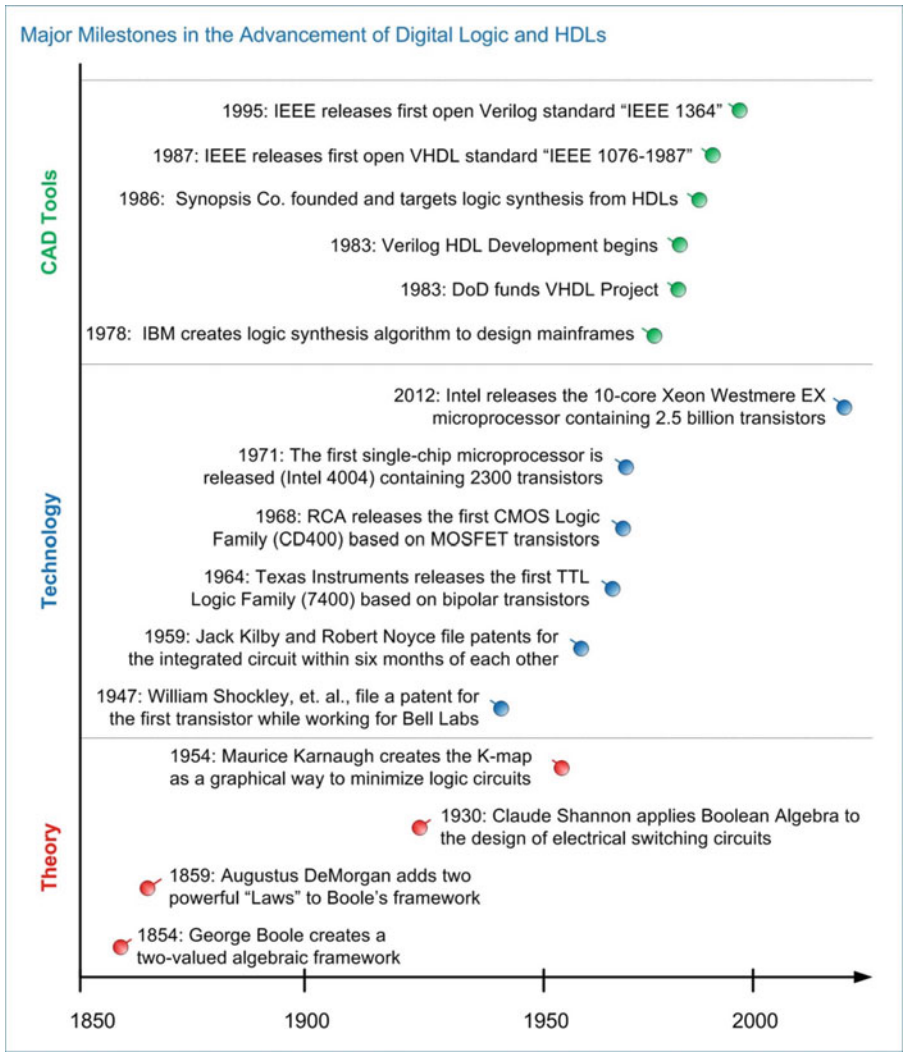


Fig. 1.1
Major milestones in the advancement of digital logic and HDLs

Concept Check

CC1.1 Why does Verilog support modeling techniques that *aren't* synthesizable?

A) There wasn't enough funding available to develop synthesis capability as it all went to the VHDL project.

B) At the time Verilog was created, synthesis was deemed too difficult to implement.

C) To allow Verilog to be used as a generic programming language.

D) Verilog needs to support all steps in the modern digital design flow, some of which are unsynthesizable such as test pattern generation and timing verification.

1.2 HDL Abstraction

HDLs were originally defined to be able to model behavior at multiple levels of abstraction. Abstraction is an important concept in engineering design because it allows us to specify how systems will operate without getting consumed prematurely with implementation details. Also, by removing the details of the lower-level implementation, simulations can be conducted in reasonable amounts of time to model the higher-level functionality. If a full computer system was simulated using detailed models for every MOSFET, it would take an impracticable amount of time to complete. Figure 1.2 shows a graphical depiction of the different layers of abstraction in digital system design.

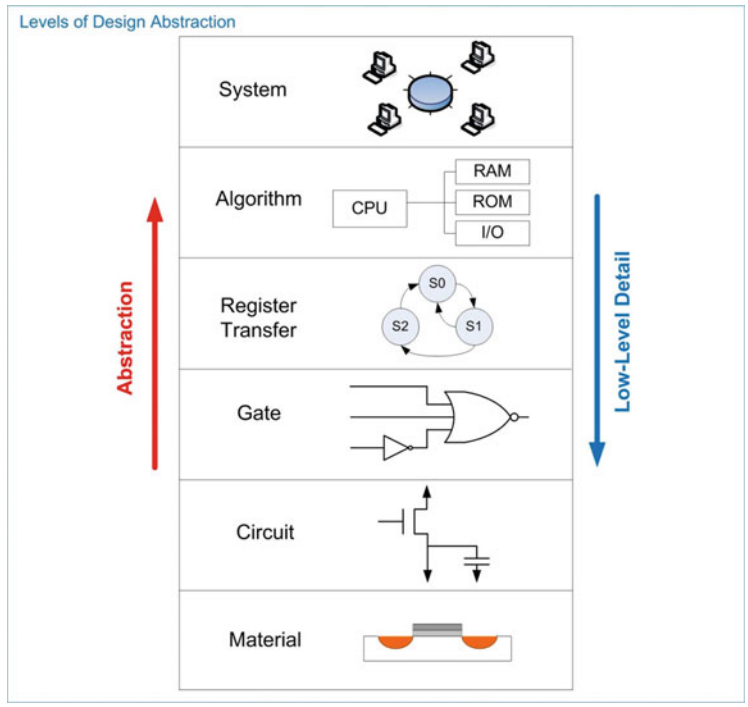


Fig. 1.2
Levels of design abstraction

The highest level of abstraction is the *system level*. At this level, behavior of a system is described by stating a set of broad specifications. An example of a design at this level is a specification such as "the computer system will perform 10 Tera Floating-Point Operations per Second (10 TFLOPS) on double-precision data and consume no more than 100 Watts of power." Notice that these specifications do not dictate the lower-level details such as the type of logic family or the type of computer architecture to use One level down from the system level is the *algorithmic level*. At this level, the specifications begin to be broken down into subsystems, each with an associated behavior that will accomplish a part of the primary task. At this level, the example computer specifications might be broken down into subsystems such as a central processing unit (CPU) to perform the computation and random access memory (RAM) to hold the inputs and outputs of the computation. One level down from the algorithmic level is the *register transfer level (RTL)*. At this level, the details of how data are moved between and within subsystems are described in addition to how the data are manipulated based on system inputs. One level down from the RTL level is the *gate level*. At this level, the design is described using basic gates and registers (or storage elements). The gate level is essentially a schematic (either graphically or text-based) that contains the components and connections that will implement the functionality from the above levels of abstraction. One level down from the gate level is the *circuit level*. The circuit level describes the operation of the basic gates and registers using transistors, wires, and other electrical components such as resistors and capacitors. Finally, the lowest level of design abstraction is the *material level*. This level describes how different materials are combined and shaped in order to implement the transistors, devices, and wires from the circuit level.

HDLs are designed to model behavior at all of these levels with the exception of the material level. While there is some capability to model circuit level behavior such as MOSFETs as ideal switches and pull-up/pull-down resistors, HDLs are not typically used at the circuit level. Another graphical depiction of design abstraction is known as the **Gajski and Kuhn's Y-chart**. A Y-chart depicts abstraction across three different design domains: behavioral, structural, and physical. Each of these design domains contains levels of abstraction (i.e., system, algorithm, RTL, gate, and circuit). An example Y-chart is shown in Fig. 1.3.

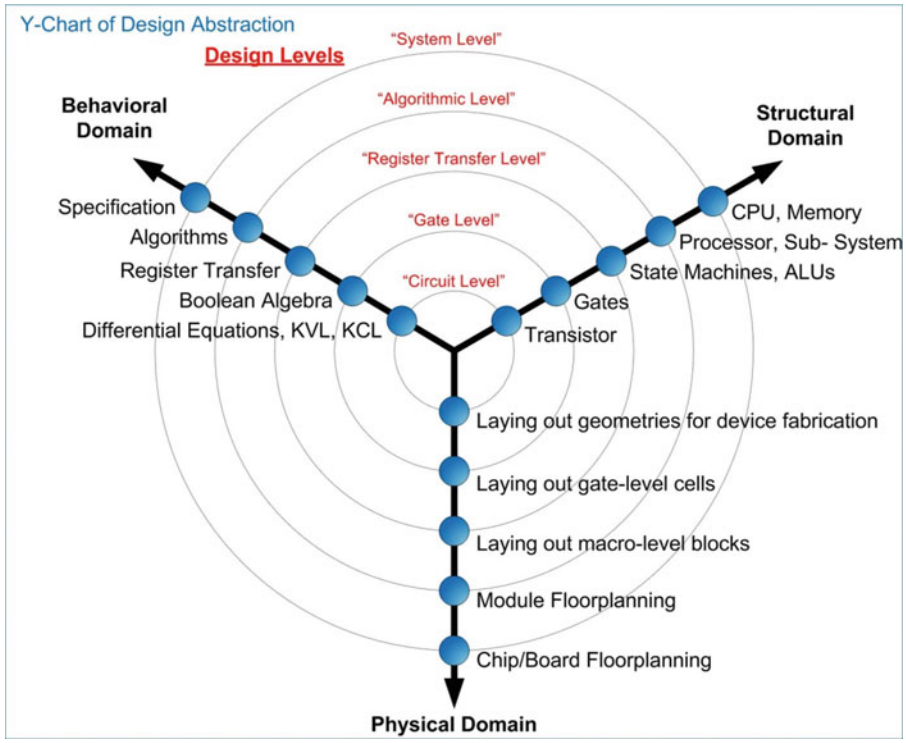


Fig. 1.3
Y-chart of design abstraction

A Y-chart also depicts how the abstraction levels of different design domains are related to each other. A top-down design flow can be visualized in a Y-chart by spiraling inward in a clockwise direction. Moving from the behavioral domain to the structural domain is the process of *synthesis*. Whenever synthesis is performed, the resulting system should be compared with the prior behavioral description. This checking is called *verification*. The process of creating the physical circuitry corresponding to the structural description is called *implementation.* The spiral continues down through the levels of abstraction until the design is implemented at a level that the geometries representing circuit elements (transistors, wires, etc.) are ready to be fabricated in silicon. Figure 1.4 shows the top-down design process depicted as an inward spiral on the Y-chart.

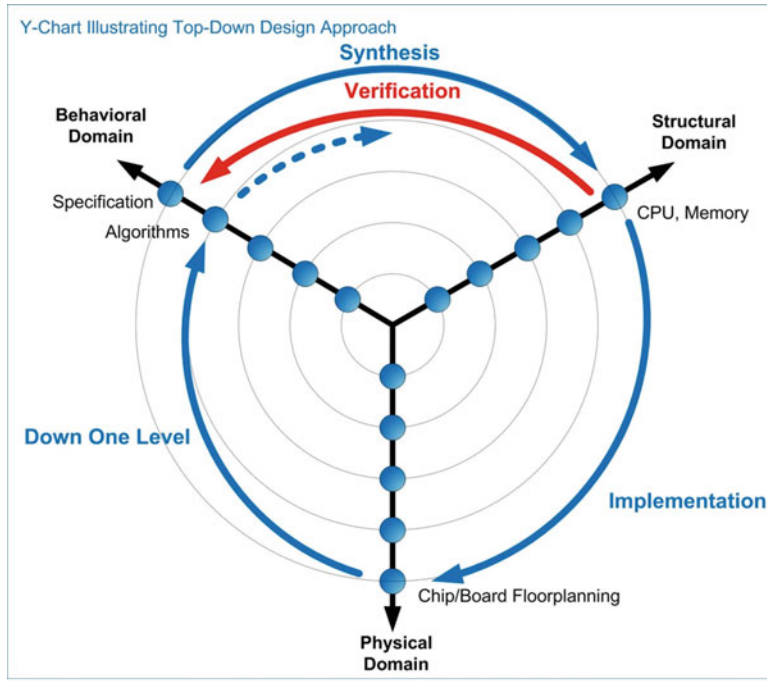


Fig. 1.4
Y-chart illustrating top-down design approach

The Y-chart represents a formal approach for large digital systems. For large systems that are designed by teams of engineers, it is critical that a formal, top-down design process is followed to eliminate potentially costly design errors as the implementation is carried out at lower levels of abstraction.

Concept Check

CC1.2 Why is abstraction an essential part of engineering design?

A) Without abstraction, all schematics would be drawn at the transistor level.

B) Abstraction allows computer programs to aid in the design process.

C) Abstraction allows the details of the implementation to be hidden while the higher-level systems are designed. Without abstraction, the details of the implementation would overwhelm the designer.

D) Abstraction allows analog circuit designers to include digital blocks in their systems.

1.3 The Modern Digital Design Flow

When performing a smaller design or the design of fully contained subsystems, the process can be broken down into individual steps. These steps are shown in Fig. 1.5. This process is given generically and applies to both *classical* and *modern* digital design. The distinction between classical and modern is that modern digital design uses HDLs and automated CAD tools for simulation, synthesis, place and route, and verification.

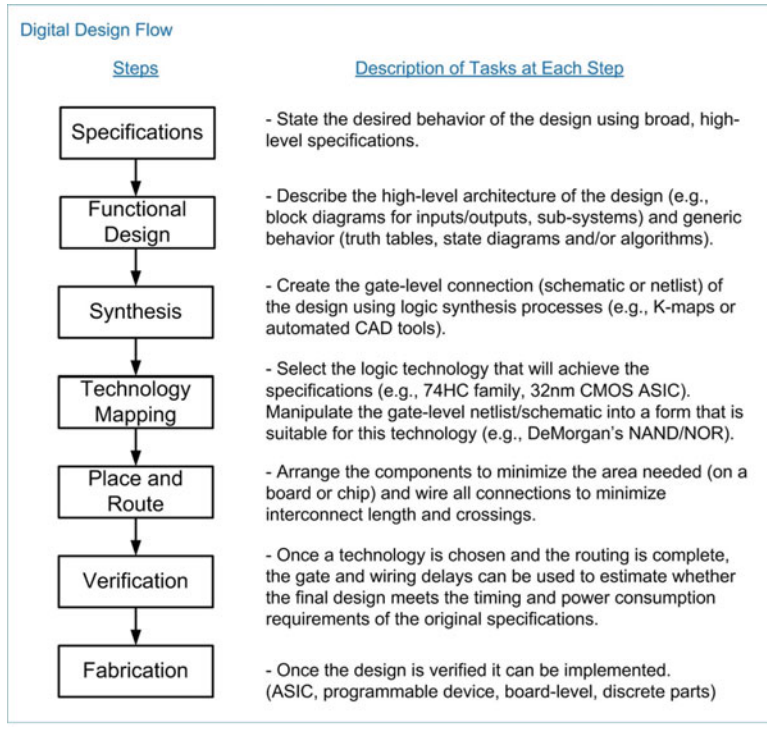


Fig. 1.5
Generic digital design flow

This generic design process flow can be used across classical and modern digital design, although modern digital design allows additional verification at each step using automated CAD tools. Figure 1.6 shows how this flow is used in the classical design approach of a combinational logic circuit.

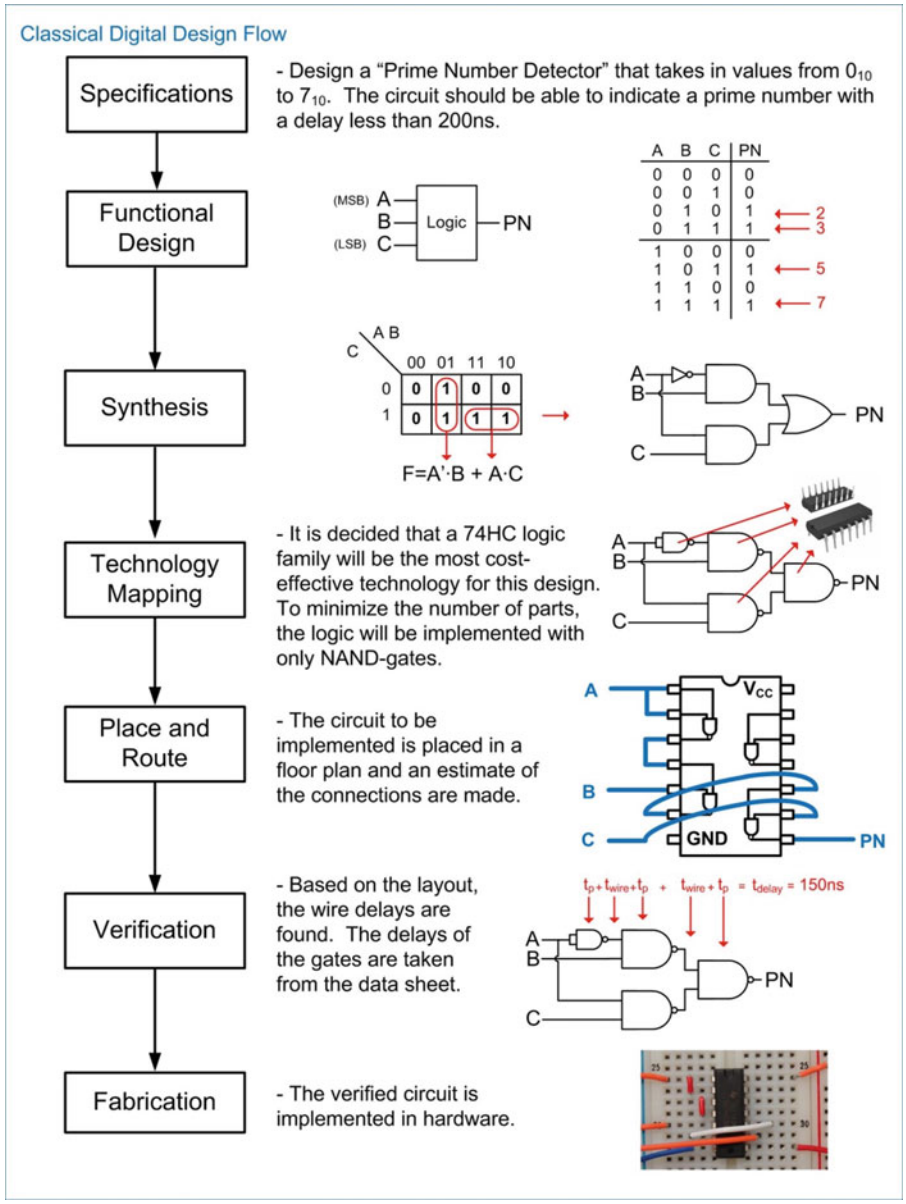


Fig. 1.6
Classical digital design flow

The modern design flow based on HDLs includes the ability to simulate functionality at each step of the process. Functional simulations can be performed on the initial behavioral description of the system. At each step of the design process, the functionality is described in more detail, ultimately moving toward the fabrication step. At each level, the detailed information can be included in the simulation to verify that the functionality is still correct and that the design is still meeting the original specifications. Figure 1.7 shows the modern digital design flow with the inclusion of simulation capability at each step.

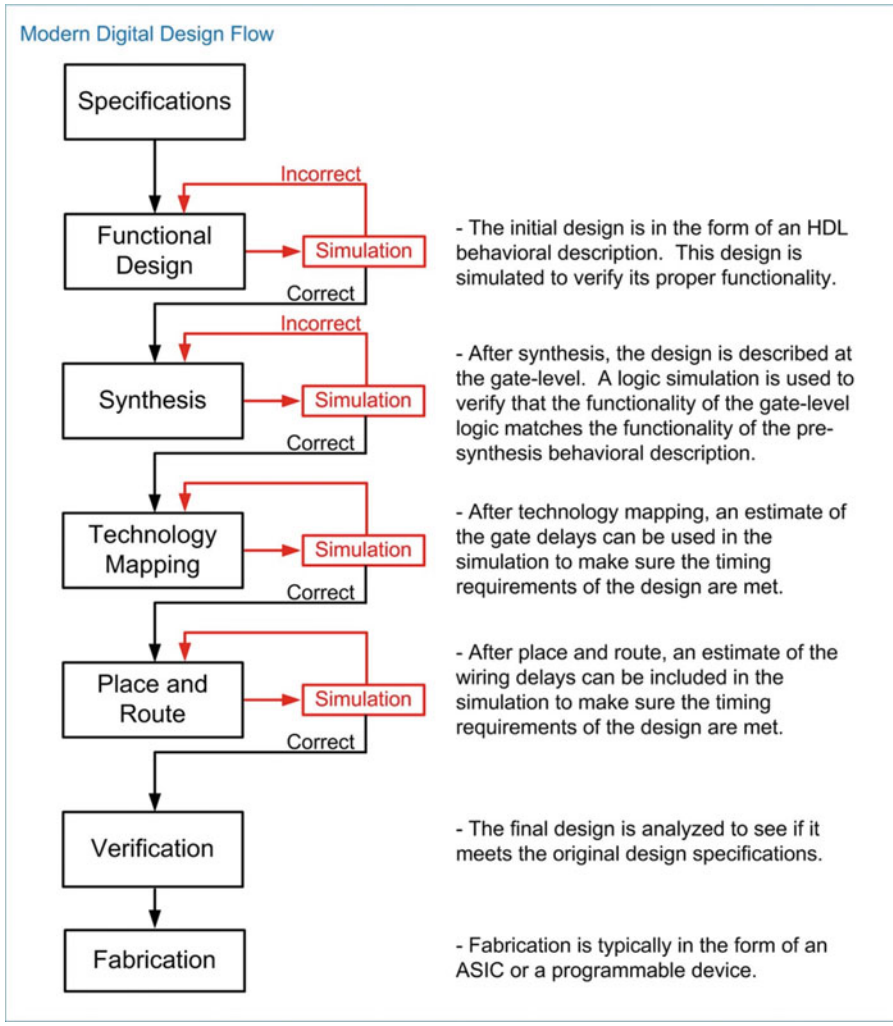


Fig. 1.7
Modern digital design flow

Concept Check

CC1.3 Why did digital designs move from schematic-entry to text-based HDLs?

A) HDL models could be much larger by describing functionality in text similar to traditional programming language.

B) Schematics required sophisticated graphics hardware to display correctly.

C) Schematics symbols became too small as designs became larger.

D) Text was easier to understand by a broader range of engineers.

Summary

- The modern digital design flow relies on computer-aided engineering (CAE) and computer-aided design (CAD) tools to manage the size and complexity of today's digital designs.
- Hardware description languages (HDLs) allow the functionality of digital systems to be entered using text. VHDL and Verilog are the two most common HDLs in use today.
- Verilog was originally created to support functional simulation of text-based designs.
- The ability to automatically synthesize a logic circuit from a Verilog behavioral description became possible approximately 10 years after the original definition of Verilog. As such, only a subset of the behavioral modeling techniques in Verilog can be automatically synthesized.
- HDLs can model digital systems at different levels of design abstraction. These include the *system*, *algorithmic*, *RTL*, *gate*, and *circuit* levels. Designing at a higher level of abstraction allows more complex systems to be modeled without worrying about the details of the implementation.

Exercise Problems

Section 1.1: History of HDLs

1.1.1 What was the original purpose of Verilog?

1.1.2 Can all of the functionality that can be described in Verilog be simulated?

1.1.3 Can all of the functionality that can be described in Verilog be synthesized?

Section 1.2: HDL Abstraction

1.2.1 Give the level of design abstraction that the following statement relates to: *if there is ever an error in the system, it should return to the reset state.*

1.2.2 Give the level of design abstraction that the following statement relates to: *once the design is implemented in a sum of products form, DeMorgan's Theorem will be used to convert it to a NAND-gate only implementation*.

1.2.3 Give the level of design abstraction that the following statement relates to: *the design will be broken down into two subsystems, one that will handle data collection and the other that will control data flow*.

1.2.4 Give the level of design abstraction that the following statement relates to: *the interconnect on the IC should be changed from aluminum to copper to achieve the performance needed in this design*.

1.2.5 Give the level of design abstraction that the following statement relates to: *the MOSFETs need to be able to drive at least eight other loads in this design.*

1.2.6 Give the level of design abstraction that the following statement relates to: *this system will contain one host computer and support up to 1000 client computers.*

1.2.7 Give the design domain that the following activity relates to: *drawing the physical layout of the CPU will require 6 months of engineering time*.

1.2.8 Give the design domain that the following activity relates to: *the CPU will be connected to four banks of memory.*

1.2.9 Give the design domain that the following activity relates to: *the fan-in specifications for this logic family require excessive logic circuitry to be used*.

1.2.10 Give the design domain that the following activity relates to: *the performance specifications for this system require 1 TFLOP at <5 W*.

Section 1.3: The Modern Digital Design Flow

1.3.1 Which step in the modern digital design flow does the following statement relate to: *a CAD tool will convert the behavioral model into a gate-level description of functionality.*

1.3.2 Which step in the modern digital design flow does the following statement relate to: *after realistic gate and wiring delays are determined, one last simulation should be performed to make sure the design meets the original timing requirements.*

1.3.3 Which step in the modern digital design flow does the following statement relate to: *if the memory is distributed around the perimeter of the CPU, the wiring density will be minimized.*

1.3.4 Which step in the modern digital design flow does the following statement relate to: *the design meets all requirements so now I'm building the hardware that will be shipped.*

1.3.5 Which step in the modern digital design flow does the following statement relate to: *the system will be broken down into three subsystems with the following behaviors.*

1.3.6 Which step in the modern digital design flow does the following statement relate to: *this system needs to have 10 Gbytes of memory.*

1.3.7 Which step in the modern digital design flow does the following statement relate to: *to meet the power requirements, the gates will be implemented in the 74HC logic family.*

Chapter 2: Verilog Constructs

This chapter begins looking at the basic construction of a Verilog module. This chapter begins by covering the built-in features of a Verilog module including the file structure, data types, operators, and declarations. This chapter provides a foundation of Verilog that will lead to modeling examples provided in Chap. 3. The original Verilog standard (IEEE 1364) has been updated numerous times since its creation in 1995. The most significant update occurred in 2001, which was titled IEEE 1394-2001. In 2005, minor improvements were added to the standard, which resulted in IEEE 1394-2005. The constructs described in this book reflect the functionality in the IEEE 1394-2005 standard. The functionality of Verilog (e.g., operators, signal types, functions, etc.) is defined within the Verilog standard, thus it is not necessary to explicitly state that a design is using the IEEE 1394 package because it is inherent in the use of Verilog.

Verilog is case sensitive. Also, each Verilog assignment, definition or declaration is terminated with a semicolon (;). As such, line wraps are allowed and do not signify the end of an assignment, definition or declaration. Line wraps can be used to make Verilog more readable. Comments in Verilog are supported in two ways. The first way is called a *line comment* and is preceded with two slashes (i.e., *//*). Everything after the slashes is considered a comment until the end of the line. The second comment approach is called a *block comment* and begins with */** and ends with a ***/*. Everything between /* and */ is considered a comment. A block comment can span multiple lines. All user-defined names in Verilog must start with an alphabetic letter, not a number. User-defined names are not allowed to be the same as any Verilog keyword. This chapter contains many definitions of syntax in Verilog. The following notations will be used throughout the chapter when introducing new constructs.

bold	= Verilog keyword, use as is, case sensitive.
italics	= User-defined name, case sensitive.
<>	= A required characteristic such as a data type, input/output, etc.

Learning Outcomes—After completing this chapter, you will be able to:

2.1. Describe the data types provided in Verilog
2.2. Describe the basic construction of a Verilog module

2.1 Data Types

In Verilog, every signal, constant, variable, and function must be assigned a *data type*. The IEEE 1394-2005 standard provides a variety of predefined data types. Some data types are synthesizable, while others are only for modeling abstract behavior. The following are the most commonly used data types in the Verilog language.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_2

2.1.1 Value Set

Verilog supports four basic values that a signal can take on: 0, 1, X, and Z. Most of the predefined data types in Verilog store these values. A description of each value supported is given below.

Value	Description
0	A logic zero, or false condition.
1	A logic one, or true condition.
x or **X**	Unknown or uninitialized.
z or **Z**	High impedance, tri-stated, or floating.

In Verilog, these values also have an associated *strength*. The strengths are used to resolve the value of a signal when it is driven by multiple sources. The names, syntax, and relative strengths are given below.

Strength	Description	Strength level
supply1	Supply drive for V_{CC}	7
supply0	Supply drive for V_{SS}, or GND	7
strong1	Strong drive to logic one	6
strong0	Strong drive to logic zero	6
pull1	Medium drive to logic one	5
pull0	Medium drive to logic zero	5
large	Large capacitive	4
weak1	Weak drive to logic one	3
weak0	Weak drive to logic zero	3
medium	Medium capacitive	2
small	Small capacitive	1
highz1	High impedance with weak pull-up to logic one	0
highz0	High impedance with weak pull-down to logic zero	0

When a signal is driven by multiple drivers, it will take on the value of the driver with the highest strength. If the two drivers have the same strength, the value will be *unknown*. If the strength is not specified, it will default to *strong drive*, or level 6.

2.1.2 Net Data Types

Every signal within Verilog must be associated with a data type. A *net data type* is one that models an interconnection (aka., a *net*) between components and can take on the values 0, 1, X, and Z. A signal with a net data type must be driven at all times and updates its value when the driver value changes. The most common synthesizable net data type in Verilog is the *wire*. The type wire will be used throughout this text. There are also a variety of other more advanced net data types that model complex digital systems with multiple drivers for the same net. The syntax and description for all Verilog net data types are given below.

Type	Description
wire	A simple connection between components.
wor	Wired-OR. If multiple drivers, their values are OR'd together.
wand	Wired-AND'd. If multiple drivers, their values are AND'd together.
supply0	Used to model the V_{SS}, (Ground or GND), power supply (supply strength inherent).
supply1	Used to model the V_{CC} power supply (supply strength inherent).
tri	Identical to **wire**. Used for readability for a net driven by multiple sources.
trior	Identical to **wor**. Used for readability for nets driven by multiple sources.
triand	Identical to **wand**. Used for readability for nets driven by multiple sources.
tri1	Pulls up to logic one when tri-stated.
tri0	Pulls down to logic zero when tri-stated.
trireg	Holds last value when tri-stated (capacitance strength inherent).

Each of these net types can also have an associated *drive strength*. The strength is used in determining the final value of the net when it is connected to multiple drivers.

2.1.3 Variable Data Types

Verilog also contains data types that model storage. These are called *variable data types*. A variable data type can take on the values 0, 1, X, and Z, but does not have an associated strength. Variable data types will hold the value assigned to them until their next assignment. The syntax and description for the Verilog variable data types are given below.

Type	Description
reg	A variable that models logic storage. Can take on values 0, 1, X, and Z.
integer	A 32-bit, 2s complement variable representing whole numbers between $-2{,}147{,}483{,}648_{10}$ to $+2{,}147{,}483{,}647$.
real	A 64-bit, floating-point variable representing real numbers between $-(2.2 \times 10^{-308})_{10}$ to $+(2.2 \times 10^{308})_{10}$.
time	An unsigned, 64-bit variable taking on values from 0_{10} to $+(9.2 \times 10^{18})$.
realtime	Same as **time**. Just used for readability.

2.1.4 Vectors

In Verilog, a *vector* is a one-dimensional array of elements. All of the net data types, in addition to the variable type reg, can be used to form vectors. The syntax for defining a vector is as follows:

```
<type> [<MSB_index>:<LSB_index>] vector_name
```

While any range of indices can be used, it is common practice to have the LSB index start at zero.

Example:

```
wire [7:0] Sum;     // This defines an 8-bit vector called "Sum" of type wire. The
                    // MSB is given the index 7 while the LSB is given the index 0.

reg [15:0] Q;       // This defines a 16-bit vector called "Q" of type reg.
```

Individual bits within the vector can be addressed using their index. Groups of bits can be accessed using an index range.

```
Sum[0];     // This is the least significant bit of the vector "Sum" defined above.
Q[15:8];    // This is the upper 8-bits of the 16-bit vector "Q" defined above.
```

2.1.5 Arrays

An *array* is a multidimensional array of elements. This can also be thought of as a "vector of vectors." Vectors within the array all have the same dimensions. To declare an array, the element type and dimensions are defined first followed by the array name and its dimensions. It is common practice to place the start index of the array on the left side of the ":" when defining its dimensions. The syntax for the creation of an array is shown below.

```
<element_type>  [<MSB_index>:<LSB_index>]  array_name  [<array_start_index>:
<array_end_index>];
```

Example:

```
reg[7:0] Mem[0:4095];  // Defines an array of 4096, 8-bit vectors of type reg.
integer A[1:100];      // Defines an array of 100 integers.
```

When accessing an array, the name of the array is given first, followed by the index of the element. It is also possible to access an individual bit within an array by adding appending the index of element.

Example:

```
Mem[2];          // This is the 3rd element within the array named "Mem".
                 // This syntax represents an 8-bit vector of type reg.

Mem[2][7];       // This is the MSB of the 3rd element within the array named "Mem".
                 // This syntax represents a single bit of type reg.

A[2];            // This is the 2nd element within the array named "A". Recall
                 //  that A was declared with a starting index of 1.
                 // This syntax represents a 32-bit, signed integer.
```

2.1.6 Expressing Numbers Using Different Bases

If a number is simply entered into Verilog without identifying syntax, it is treated as an integer. However, Verilog supports defining numbers in other bases. Verilog also supports an optional bit size and sign of a number. When defining the value of arrays, the "_" can be inserted between numerals to improve readability. The "_" is ignored by the Verilog compiler. Values of numbers can be entered in either upper or lower case (i.e., b or B, f or F, etc.). The syntax for specifying the base of a number is as follows:

```
<size_in_bits>'<base><value>
```

Note that specifying the size is optional. If it is omitted, the number will default to a 32-bit vector with leading zeros added as necessary. The supported bases are as follows:

Syntax	Description
'b	Unsigned binary.
'o	Unsigned octal.
'd	Unsigned decimal.
'h	Unsigned hexadecimal.
'sb	Signed binary.
'so	Signed octal.
'sd	Signed decimal.
'sh	Signed hexadecimal.

Example:

```
10              // This is treated as decimal 10, which is a 32-bit signed vector.
4'b1111         // A 4-bit number with the value 1111₂.
8'b1011_0000    // An 8-bit number with the value 10110000₂.
8'hFF           // An 8-bit number with the value 11111111₂.
8'hff           // An 8-bit number with the value 11111111₂.
6'hA            // A 6-bit number with the value 001010₂. Note that leading zeros
                // were added to make the value 6-bits.
8'd7            // An 8-bit number with the value 00000111₂.
32'd0           // A 32-bit number with the value 0000_0000₁₆.
'b1111          // A 32-bit number with the value 0000_000F₁₆.
8'bZ            // An 8-bit number with the value ZZZZ_ZZZZ.
```

2.1.7 Assigning Between Different Types

Verilog is said to be a weakly typed (or loosely typed) language, meaning that it permits assignments between different data types. This is as opposed to a strongly typed language (such as VHDL) where signal assignments are only permitted between like types. The reason Verilog permits assignment between different types is because it treats all of its types as just groups of bits. When assigning between different types, Verilog will automatically truncate or add leading bits as necessary to make the assignment work. The following examples illustrate how Verilog handles a few assignments between different types. Assume that a variable called ABC_TB has been declared as type reg[2:0].

Example:

```
ABC_TB = 2'b00;   // ABC_TB will be assigned 3'b000. A leading bit is automatically
                     added.
ABC_TB = 5;       // ABC_TB will be assigned 3'b101. The integer is truncated to
                     3-bits.
ABC_TB = 8;       // ABC_TB will be assigned 3'b000. The integer is truncated to
                     3-bits.
```

CONCEPT CHECK

CC2.1 The two most commonly used data types in Verilog are *wire* and *reg*? What is the fundamental difference between these types?

A) They are the same because they can both take on 0, 1, X, or Z.

B) A *wire* is a net data type, meaning that it must be driven at all times. A *reg* is a variable data type, meaning that it will hold its value after it is assigned.

C) A *wire* can only take on values of 0 and 1 while a *reg* can take on 0, 1, X, or Z.

D) They cannot drive one other.

2.2 Verilog Module Construction

A Verilog design describes a single system in a single file. The file has the suffix *.v. Within the file, the system description is contained within a **module**. The module includes the interface to the system (i.e., the inputs and outputs) and the description of the behavior. Figure 2.1 shows a graphical depiction of a Verilog file.

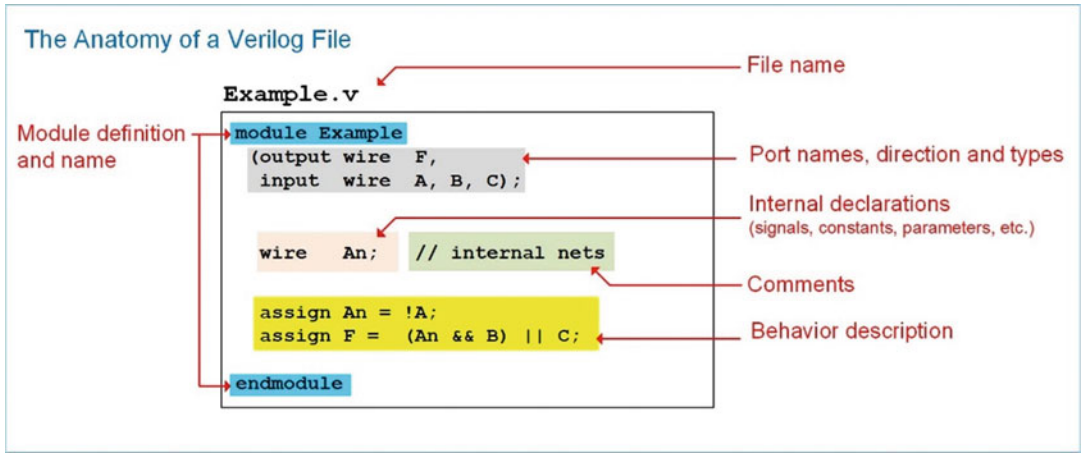


Fig. 2.1
The anatomy of a Verilog file

2.2.1 The Module

All systems in Verilog are encapsulated inside of a **module**. Modules can include instantiations of lower-level modules in order to support hierarchical designs. The keywords **module** and **endmodule** signify the beginning and end of the system description. When working on large designs, it is common practice to place each module in its own file with the same name.

```
module module_name (port_list);                    // Pre Verilog-2001
// port_definitions
// module_items
endmodule
```

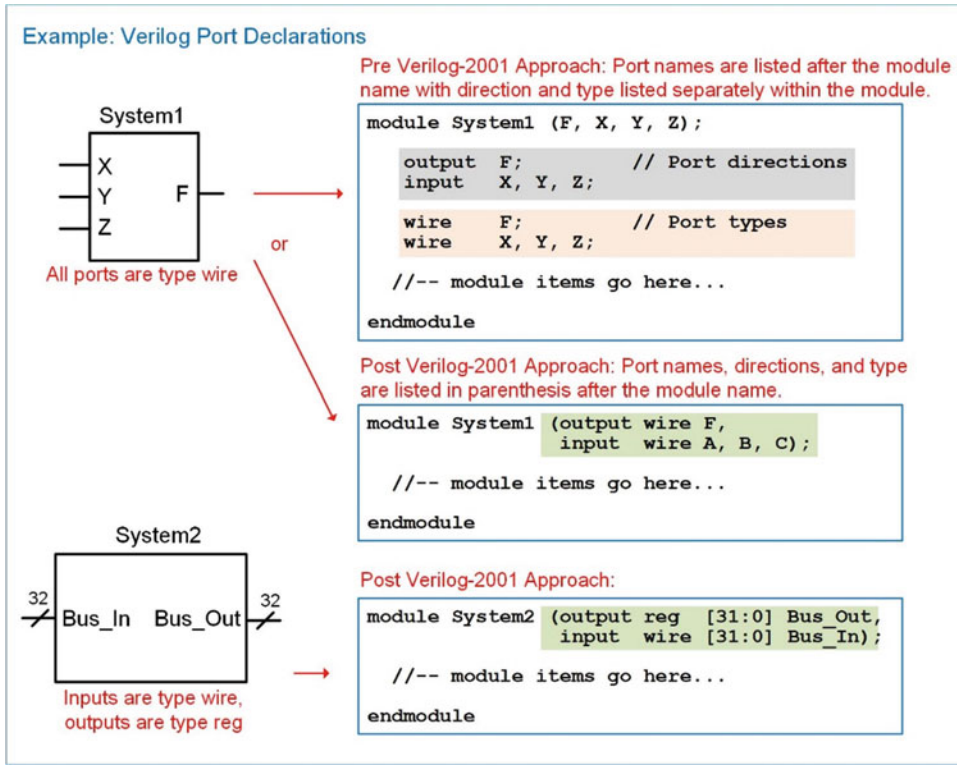
or

```
module module_name (port_list and port_definitions);    // Verilog-2001 and after
// module_items
endmodule
```

2.2.2 Port Definitions

The first item within a module is its definition of the inputs and outputs, or ports. Each port needs to have a user-defined name, a direction, and a type. The user-defined port names are case sensitive and must begin an alphabetic character. The port directions are declared to be one of the three types: **input**, **output**, and **inout**. A port can take on any of the previously described data types, but only wires, registers, and integers are synthesizable. Port names with the same type and direction can be listed on the same line separated by commas.

There are two different port definition styles supported in Verilog. Prior to the Verilog-2001 release, the port names were listed within parentheses after the module name. Then within the module, the directionality and type of the ports were listed. Starting with the Verilog-2001 release, the port directions and types could be included alongside the port names within the parenthesis after the module name. This approach mimicked more of an ANSCI-C approach to passing inputs/outputs to a system. In this text, the newer approach to port definition will be used. Example 2.1 shows multiple approaches for defining a module and its ports.



Example 2.1
Declaring Verilog module ports

2.2.3 Signal Declarations

A signal that is used for internal connections within a system is declared within the module before its first use. Each signal must be declared by listing its type followed by a user-defined name. Signal names of like type can be declared on the same line separated with a comma. All of the legal data types described above can be used for signals; however, only types net, reg, and integer will synthesize directly. The syntax for a signal declaration is as follows:

```
<type> name;
```

Example:

```
wire    node1;       // declare a signal named "node1" of type wire
reg     Q2, Q1, Q0;  // declare three signals named "Q2", "Q1", and "Q0", all of type reg
wire    [63:0] bus1; // declare a 64-bit vector named "bus1" with all bits of type wire
integer i,j;         // declare two integers called "i" and "j"
```

Verilog supports a hierarchical design approach, thus signal names can be the same within a subsystem as those at a higher level without conflict. Figure 2.2 shows an example of legal signal naming in a hierarchical design.

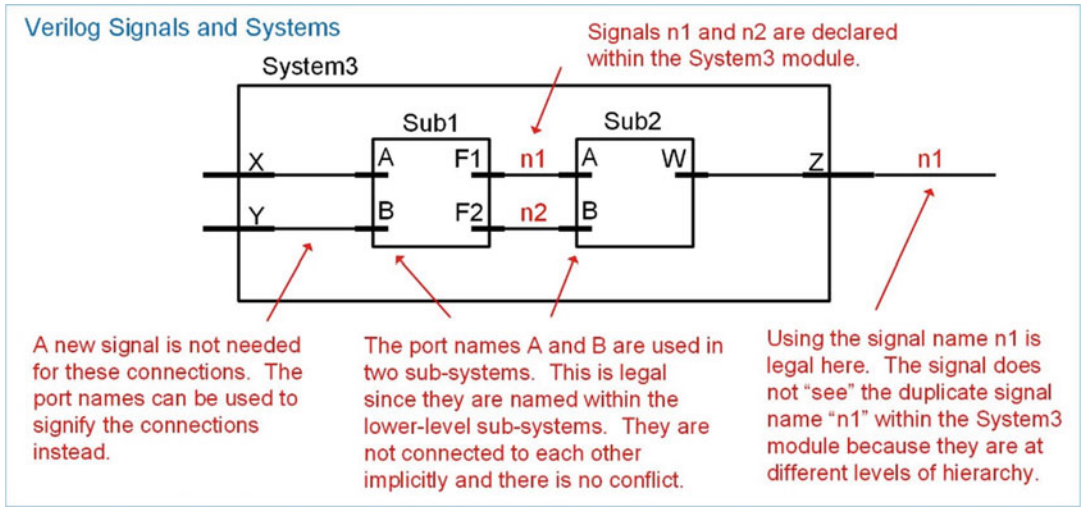


Fig. 2.2
Verilog signals and systems

2.2.4 Parameter Declarations

A parameter, or constant, is useful for representing a quantity that will be used multiple times in the architecture. The syntax for declaring a parameter is as follows:

```
parameter <type> constant_name = <value>;
```

Note that the type is optional and can only be **integer**, **time**, **real**, or **realtime**. If a type is provided, the parameter will have the same properties as a variable of the same time. If the type is excluded, the parameter will take on the type of the value assigned to it.

Example:

```
parameter BUS_WIDTH = 32;
parameter NICKEL   = 8'b0000_0101;
```

Once declared, the constant name can be used throughout the module. The following example illustrates how we can use a constant to define the size of a vector. Notice that since we defined the constant to be the actual width of the vector (i.e., 32-bits), we need to subtract one from its value when defining the indices (i.e., [31:0]).

Example:

```
wire [BUS_WIDTH-1:0] BUS_A;     // It is acceptable to add a "space" for readability
```

2.2.5 Compiler Directives

A compiler directive provides additional information to the simulation tool on how to interpret the Verilog model. A compiler directive is placed before the module definition and is preceded with a backtick (i.e., `). Note that this is not an apostrophe. A few of the most commonly used compiler directives are as follows:

Syntax	Description
`timescale <unit>,<precision>	Defines the timescale of the delay unit and its smallest precision.
`include <filename>	Includes additional files in the compilation.
`define <macroname> <value>	Declares a global constant.

Example:

```
`timescale 1ns/1ps     // Declares the unit of time is 1 ns with a precision of 1ps.
                       // The precision is the smallest amount that the time can
                       // take on. For example, with this directive the number
                       // 0.001 would be interpreted as 0.001 ns, or 1 ps.
                       // However, the number 0.0001 would be interpreted as 0 since
                       // it is smaller than the minimum precision value.
```

Concept Check

CC2.2 If a signal is declared within a module, can the same name be used in other modules within a hierarchical system?

A) Yes. To support hierarchy, Verilog signals are only seen within their respective module. That allows other modules to use the same names.

B) No. Once a signal name is defined, it cannot be used again.

Summary

- In a Verilog source file, all functionality is contained within a module. The first portion of the module is the port definition. The second portion contains declarations of internal signals / constants / parameters. The third portion contains the description of the behavior.
- A *port* is an input or output to a system that is defined as part of the initial module statement. A *signal*, or *net*, is an internal connection within the system that is declared inside of the module. A signal is not visible outside of the system.
- Instantiating other modules from within a higher-level module is how Verilog implements hierarchy. A lower-level module can be instantiated as many times as desired. An instance identifier is useful is keeping track of each instantiation. The ports of the component can be connected using either *explicit* or *positional port mapping*.

Exercise Problems

Section 2.1: Data Types

2.1.1 What is the name of the main design unit in Verilog?

2.1.2 What portion of the Verilog module describes the inputs and outputs?

2.1.3 What step is necessary if a system requires internal connections?

2.1.4 What are all the possible values that a Verilog net type can take on?

2.1.5 What is the highest strength that a value can take on in Verilog?

2.1.6 What is the range of decimal numbers that can be represented using the type *integer* in Verilog?

2.1.7 What is the width of the vector defined using the type *[63:0] wire*?

2.1.8 What is the syntax for indexing the most significant bit in the type *[31:0] wire*? Assume the vector is named *example*.

2.1.9 What is the syntax for indexing the least significant bit in the type *[31:0] wire*? Assume the vector is named *example*.

2.1.10 What is the difference between a *wire* and *reg* type?

2.1.11 How many bits is the type *integer* by default?

2.1.12 How many bits is the type *real* by default?

Section 2.2: Verilog Module Construction

2.2.1 What three directions can a *module port* take on?

2.2.2 What data types can a *signal* take on within a module?

2.2.3 What data types can a *parameter* take on within a module?

2.2.4 What is the purpose of a compiler directive?

Chapter 3: Modeling Concurrent Functionality in Verilog

This chapter presents a set of built in operators that will allow basic logic expressions to be modeled within a Verilog module. Specific focus is given to continuous signal assignments including the use of logical, conditional, and delay operators. This chapter then presents a series of combinational logic model examples.

Learning Outcomes—After completing this chapter, you will be able to:

3.1 Describe the various built-in operators within Verilog
3.2 Design a Verilog model for a combinational logic circuit using continuous assignment and logical operators
3.3 Design a Verilog model for a combinational logic circuit using continuous assignment and conditional operators
3.4 Design a Verilog model for a combinational logic circuit using continuous assignment with delay

3.1 Verilog Operators

There are a variety of predefined operators in the Verilog standard. It is important to note that operators are defined to work on specific data types and that not all operators are synthesizable.

3.1.1 Assignment Operator

Verilog uses the equal sign (=) to denote an assignment. The left-hand side (LHS) of the assignment is the target signal. The right-hand side (RHS) contains the input arguments and can contain both signals, constants, and operators.

Example:

```
F1 = A;       // F1 is assigned the signal A
F2 = 8'hAA;   // F2 is an 8-bit vector and is assigned the value 10101010₂
```

3.1.2 Continuous Assignment

Verilog uses the keyword **assign** to denote a continuous signal assignment. After this keyword, an assignment is made using the = symbol. The left-hand side (LHS) of the assignment is the target signal and must be a net type. The right-hand side (RHS) contains the input arguments and can contain nets, regs, constants, and operators. A continuous assignment models combinational logic. Any change to the RHS of the expression will result in an update to the LHS target net. The net being assigned to must be declared prior to the first continuous assignment. Multiple continuous assignments can be made to the same net. When this happens, the assignment containing signals with the highest drive strength will take priority.

Example:

```
assign F1 = A;       // F1 is updated anytime A changes, where A is a signal
assign F2 = 1'b0;    // F2 is assigned the value 0
assign F3 = 4'hAA;   // F3 is an 8-bit vector and is assigned the value 10101010₂
```


B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_3

Each individual assignment will be executed concurrently and synthesized as separate logic circuits. Consider the following example.

Example:

```
assign X = A;
assign Y = B;
assign Z = C;
```

When simulated, these three lines of Verilog will make three separate signal assignments at the exact same time. This is different from a programming language that will first assign A to X, then B to Y, and finally C to Z. In Verilog, this functionality is identical to three separate wires. This description will be directly synthesized into three separate wires.

Below is another example of how continuous signal assignments in Verilog differ from a sequentially executed programming language.

Example:

```
assign A = B;
assign B = C;
```

In a Verilog simulation, the signal assignments of C to B and B to A will take place at the same time. This means during synthesis, the signal B will be eliminated from the design since this functionality describes two wires in series. Automated synthesis tools will eliminate this unnecessary signal name. This is not the same functionality that would result if this example was implemented as a sequentially executed computer program. A computer program would execute the assignment of B to A first, then assign the value of C to B second. In this way, B represents a storage element that is passed to A before it is updated with C.

3.1.3 Bitwise Logical Operators

Bitwise operators perform logic functions on individual bits. The inputs to the operation are single bits and the output is a single bit. In the case where the inputs are vectors, each bit in the first vector is operated on by the bit in the same position from the second vector. If the vectors are not the same length, the shorter vector is padded with leading zeros to make both lengths equal. Verilog contains the following bitwise operators:

Syntax	Operation
~	Negation
&	AND
|	OR
^	XOR
~^ or ^~	XNOR
<<	Logical shift left (fill empty LSB location with zero)
>>	Logical shift right (fill empty MSB location with zero)

Example:

```
~X              // invert each bit in X
X & Y           // AND each bit of X with each bit of Y
X | Y           // OR each bit of X with each bit of Y
X ^ Y           // XOR each bit of X with each bit of Y
X ~^ Y          // XNOR each bit of X with each bit of Y
X << 3          // Shift X left 3 times and fill with zeros
Y >> 2          // Shift Y right 2 times and fill with zeros
```

3.1.4 Reduction Logic Operators

A *reduction* operator is one that uses each bit of a vector as individual inputs into a logic operation and produces a single bit output. Verilog contains the following reduction logic operators.

Syntax	Operation
&	AND all bits in the vector together (1-bit result)
~&	NAND all bits in the vector together (1-bit result)
|	OR all bits in the vector together (1-bit result)
~|	NOR all bits in the vector together (1-bit result)
^	XOR all bits in the vector together (1-bit result)
~^ or **^~**	XNOR all bits in the vector together (1-bit result)

Example:

```
&X       // AND all bits in vector X together
~&X      // NAND all bits in vector X together
|X       // OR all bits in vector X together
~|X      // NOR all bits in vector X together
^X       // XOR all bits in vector X together
~^X      // XNOR all bits in vector X together
```

3.1.5 Boolean Logic Operators

A Boolean logic operator is one that returns a value of TRUE (1) or FALSE (0) based on a logic operation of the input operations. These operations are used in decision statements.

Syntax	Operation
!	Negation
&&	AND
||	OR

Example:

```
!X       // TRUE if all values in X are 0, FALSE otherwise
X && Y   // TRUE if the bitwise AND of X and Y results in all ones, FALSE otherwise
X || Y   // TRUE if the bitwise OR of X and Y results in all ones, FALSE otherwise
```

3.1.6 Relational Operators

A relational operator is one that returns a value of TRUE (1) or FALSE (0) based on a comparison of two inputs.

Syntax	Description
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Example:

```
X == Y     // TRUE if X is equal to Y, FALSE otherwise
X != Y     // TRUE if X is not equal to Y, FALSE otherwise
X < Y      // TRUE if X is less than Y, FALSE otherwise
X > Y      // TRUE if X is greater than Y, FALSE otherwise
X <= Y     // TRUE if X is less than or equal to Y, FALSE otherwise
X >= Y     // TRUE if X is greater than or equal to Y, FALSE otherwise
```

3.1.7 Conditional Operators

Verilog contains a conditional operator that can be used to provide a more intuitive approach to modeling logic statements. The keyword for the conditional operator is **?** with the following syntax:

```
<target_net> = <Boolean_condition> ? <true_assignment> : <false_assignment>;
```

This operator specifies a Boolean condition in which if evaluated TRUE, the *true_assignment* will be assigned to the target. If the Boolean condition is evaluated FALSE, the *false_assignment* portion of the operator will be assigned to the target. The values in this assignment can be signals or logic values. The Boolean condition can be any combination of the Boolean operators described above. Nested conditional operators can also be implemented by inserting subsequent conditional operators in place of the *false_value*.

Example:

```
F = (A == 1'b0) ? 1'b1 : 1'b0;                  // If A is a zero, F=1, otherwise F=0.
                                                   This models an inverter.

F = (sel == 1'b0) ? A : B;                      // If sel is a zero, F=A, otherwise F=B.
                                                   This models a selectable switch.

F = ((A == 1'b0) && (B == 1'b0)) ? 1'b'0 :      // Nested conditional statements.
    ((A == 1'b0) && (B == 1'b1)) ? 1'b'1 :      // This models an XOR gate.
    ((A == 1'b1) && (B == 1'b0)) ? 1'b'1 :
    ((A == 1'b1) && (B == 1'b1)) ? 1'b'0;

F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;          // This models the logic expression
                                                // F = C'·(A'+B).
```

3.1.8 Concatenation Operator

In Verilog, the curly brackets (i.e., **{}**) are used to concatenate multiple signals. The target of this operation must be the same size of the sum of the sizes of the input arguments.

Example:

```
Bus1[7:0] = {Bus2[7:4], Bus3[3:0]};  // Assuming Bus1, Bus2, and Bus3 are all 8-bit
                                     // vectors, this operation takes the upper 4-bits of
                                     // Bus2, concatenates them with the lower 4-bits of
                                     // Bus3, and assigns the 8-bit combination to Bus1.

BusC = {BusA, BusB};                 // If BusA and BusB are 4-bits, then BusC
                                     // must be 8-bits.

BusC[7:0] = {4'b0000, BusA};         // This pads the 4-bit vector BusA with 4x leading
                                     //  zeros and assigns to the 8-bit vector BusC.
```

3.1.9 Replication Operator

Verilog provides the ability to concatenate a vector with itself through the *replication operator*. This operator uses double curly brackets (i.e., **{{}}**) and an integer indicating the number of replications to be performed. The replication syntax is as follows:

```
{<number_of_replications>{<vector_name_to_be_replicated>}}
```

Example:

```
BusX = {4{Bus1}};          // This is equivalent to: BusX = {Bus1, Bus1, Bus1, Bus1};
BusY = {2{A,B}};           // This is equivalent to: BusY = {A, B, A, B};
BusZ = {Bus1, {2{Bus2}}};  // This is equivalent to: BusZ = {Bus1, Bus2, Bus2};
```

3.1.10 Numerical Operators

Verilog also provides a set of numerical operators as follows:

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2s complement negation (when placed in front of an argument)
*****	Multiplication
/	Division
%	Modulus
******	Raise to the power
<<<	Shift to the left, fill with zeros
<<<	Shift to the right, fill with sign bit

Example:

```
X + Y      // Add X to Y
X - Y      // Subtract Y from X
-X         // Take the two's complement negation of X
X * Y      // Multiply X by Y
X / Y      // Divide X by Y
X % Y      // Modulus X/Y
X ** Y     // Raise X to the power of Y
X <<< 3    // Shift X left 3 times, fill with zeros
X >>> 2    // Shift X right 2 times, fill with sign bit
```

Verilog will allow the use of these operators on arguments of different sizes, types, and signs. The rules of the operations are as follows:

- If two vectors are of different sizes, the smaller vector is expanded to the size of the larger vector.
  - If the smaller vector is unsigned, it is padded with zeros.
  - If the smaller vector is signed, it is padded with the sign bit.
- If one of the arguments is real, then the arithmetic will take place using real numbers.
- If one of the arguments is unsigned, then all arguments will be treated as unsigned.

Example 3.1 shows the behavioral model for a 4-bit adder in Verilog using a combination of operators including continuous assignment, numerical addition, and concatenation. Note that when adding two *n*-bit arguments the sum produced will be *n* + 1 bits. This can be handled in Verilog by

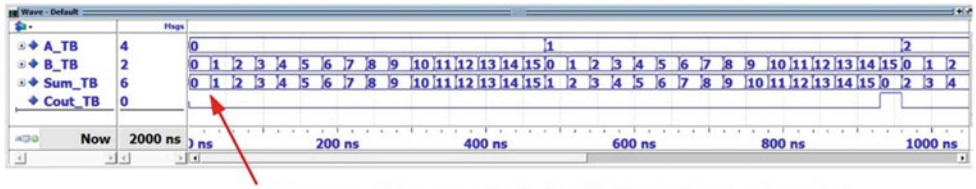
concatenating the Cout and Sum outputs on the LHS of the assignment. The entire add operation can be accomplished in a single continuous assignment that contains both the concatenation and addition operators. When using continuous assignment, the LHS must be a net data type. This means the outputs Cout and Sum need to be declared as type wire.

Example: Behavioral Model of a 4-Bit Adder in Verilog

```
module adder_4bit (output wire [3:0] Sum,
                   output wire       Cout,
                   input  wire [3:0] A, B);

   assign {Cout, Sum} = A + B;

endmodule
```

When using continuous assignment, the LHS needs to be a net data type.

The addition of two 4-bit numbers will result in a 5-bit sum. Cout and Sum are concatenated on the RHS of the assignment to accomidate 5-bits.



Since no delay was included in the behavioral model, the outputs are produced instantaneously.

Example 3.1
Behavioral model of a 4-bit adder in Verilog

3.1.11 Operator Precedence

The following is the order of precedence of the Verilog operators. If two operators of the same type appear in an expression without parenthesis to dictate the order of precedence, the precedence will be determined by executing from the operations from left to right.

Operators	Precedence	Notes
! ~ + –	Highest	Bitwise/Unary
{} {{}}		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ –	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
| ~|	↓	OR/NOR Operators
&&		Boolean AND
||		Boolean OR
?:	Lowest	Conditional Operator

Concept Check

CC3.1 For the expression: F = !A & (B I !C); What is the order of execution of the bitwise operations?

A) Negate → OR → AND

B) Negate → AND → OR

C) OR → Negate → AND

D) OR → AND → Negate

3.2 Continuous Assignment with Logical Operators

When modeling synthesizable logic, it is important to remember that Verilog is a hardware description language (HDL), not a programming language. In a programming language, the lines of code are executed sequentially as they appear in the source file. In Verilog, the lines of code represent the behavior of real hardware. Thus, the assignments are executed concurrently unless specifically noted otherwise. Each of the bitwise logical operators described in Sect. 3.1.3 can be used in conjunction with continuous signal assignments to create individual combinational logic circuits.

3.2.1 Logical Operator Example: SOP Circuit

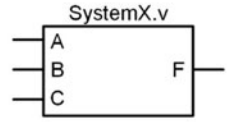
Example 3.2 shows how to design a Verilog model of a combinational logic circuit using continuous assignment and logical operators. Note that in this example, the logic expressions must first be determined by hand prior to modeling in Verilog.

Example: Modeling Combinational Logic using Continuous Assignment with Logical Operators

Implement the following truth table using continuous assignment with logical operators.

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

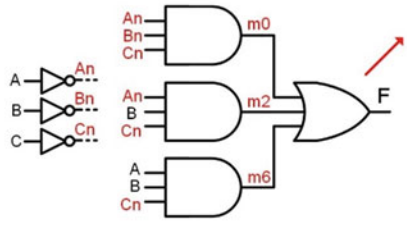
Let's call the module *SystemX*. First, let's declare the ports. The module will have three inputs (A, B, C) and one output (F). We'll use the type wire for all inputs/outputs so that this will synthesize directly into real circuitry.



Now we can design the behavior. We will create a canonical sum of products logic expression for this truth table using minterms.

$$F = \sum_{A,B,C}(0,2,6) = A'\cdot B'\cdot C' + A'\cdot B\cdot C' + A\cdot B\cdot C'$$

Drawing out the logic diagram will help us understand which internal signals need to be declared for the interim connections. Since there is a need for the complement of each of the inputs, the first set of logic will be three inverters. We'll need to create three wires to hold the inverted versions of the inputs. Let's call them An, Bn and Cn. We'll also need three wires to hold the outputs of the AND gates. Let's call them m0, m2 and m6. Using these internal wires, the port names, and logical operators, we can describe the behavior of the logic expression above.




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

   wire  An, Bn, Cn;   // internal nets
   wire  m0, m2, m6;

   assign An = ~A;              // Not's
   assign Bn = ~B;
   assign Cn = ~C;

   assign m0 = An & Bn & Cn;   // AND's
   assign m2 = An & B  & Cn;
   assign m6 = A  & B  & Cn;

   assign F  = m0 | m2 | m6;   // OR

endmodule
```

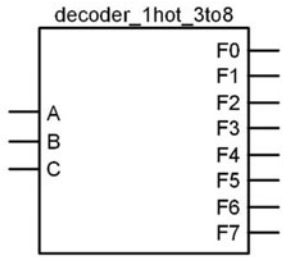
Example 3.2
Combinational logic using continuous assignment with logical operators

3.2.2 Logical Operator Example: One-Hot Decoder

A one-hot decoder is a circuit that has n inputs and 2^n outputs. Each output will assert for one and only one input code. Since there are 2^n outputs, there will always be one and only one output asserted at any given time. Example 3.3 shows how to model a 3-to-8 one-hot decoder in Verilog with continuous assignment and logic operators.

Example: 3-to-8 One-Hot Decoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



A	B	C	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

To implement this in Verilog using logical operators, we must first determine the logic that will be used in the continuous assignment. Again, since each logic function only has one input code corresponding to an output of '1', the minterm can be used to implement the logic.

$F0 = \sum_{A,B,C}(0) = A'\cdot B'\cdot C'$ $\quad F4 = \sum_{A,B,C}(4) = A\cdot B'\cdot C'$

$F1 = \sum_{A,B,C}(1) = A'\cdot B'\cdot C$ $\quad F5 = \sum_{A,B,C}(5) = A\cdot B'\cdot C$

$F2 = \sum_{A,B,C}(2) = A'\cdot B\cdot C'$ $\quad F6 = \sum_{A,B,C}(6) = A\cdot B\cdot C'$

$F3 = \sum_{A,B,C}(3) = A'\cdot B\cdot C$ $\quad F7 = \sum_{A,B,C}(7) = A\cdot B\cdot C$

In Verilog, each of the outputs requires a separate continuous assignment.

```
module decoder_1hot_3to8
  (output wire F0, F1, F2, F3, F4, F5, F6, F7,
   input  wire A, B, C);

  assign F0 = ~A & ~B & ~C;
  assign F1 = ~A & ~B &  C;
  assign F2 = ~A &  B & ~C;
  assign F3 = ~A &  B &  C;
  assign F4 =  A & ~B & ~C;
  assign F5 =  A & ~B &  C;
  assign F6 =  A &  B & ~C;
  assign F7 =  A &  B &  C;

endmodule
```

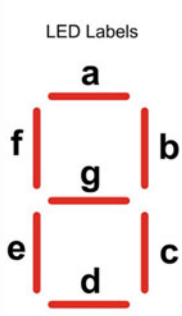
Example 3.3
3-to-8 one-hot decoder—Verilog modeling using logical operators

3.2.3 Logical Operator Example: 7-Segment Display Decoder

A 7-segment display decoder is a circuit used to drive character displays that are commonly found in applications such as digital clocks and household appliances. A character display is made up of seven individual LEDs, typically labeled a-g. The input to the decoder is the binary equivalent of the decimal or Hex character that is to be displayed. The output of the decoder is the arrangement of LEDs that will form the character. Decoders with 2-inputs can drive characters "0" to "3." Decoders with 3-inputs can drive characters "0" to "7." Decoders with 4-inputs can drive characters "0" to "F" with the case of the Hex characters being "A, b, c or C, d, E and F."

Let us look at an example of how to design a 3-input, 7-segment decoder in Verilog. The first step in the process is to create the truth table for the outputs that will drive the LEDs in the display. We'll call these outputs F_a, F_b, . . ., F_g. Example 3.4 shows how to construct the truth table for the 7-segment display decoder. In this table, a logic 1 corresponds to the LED being ON.

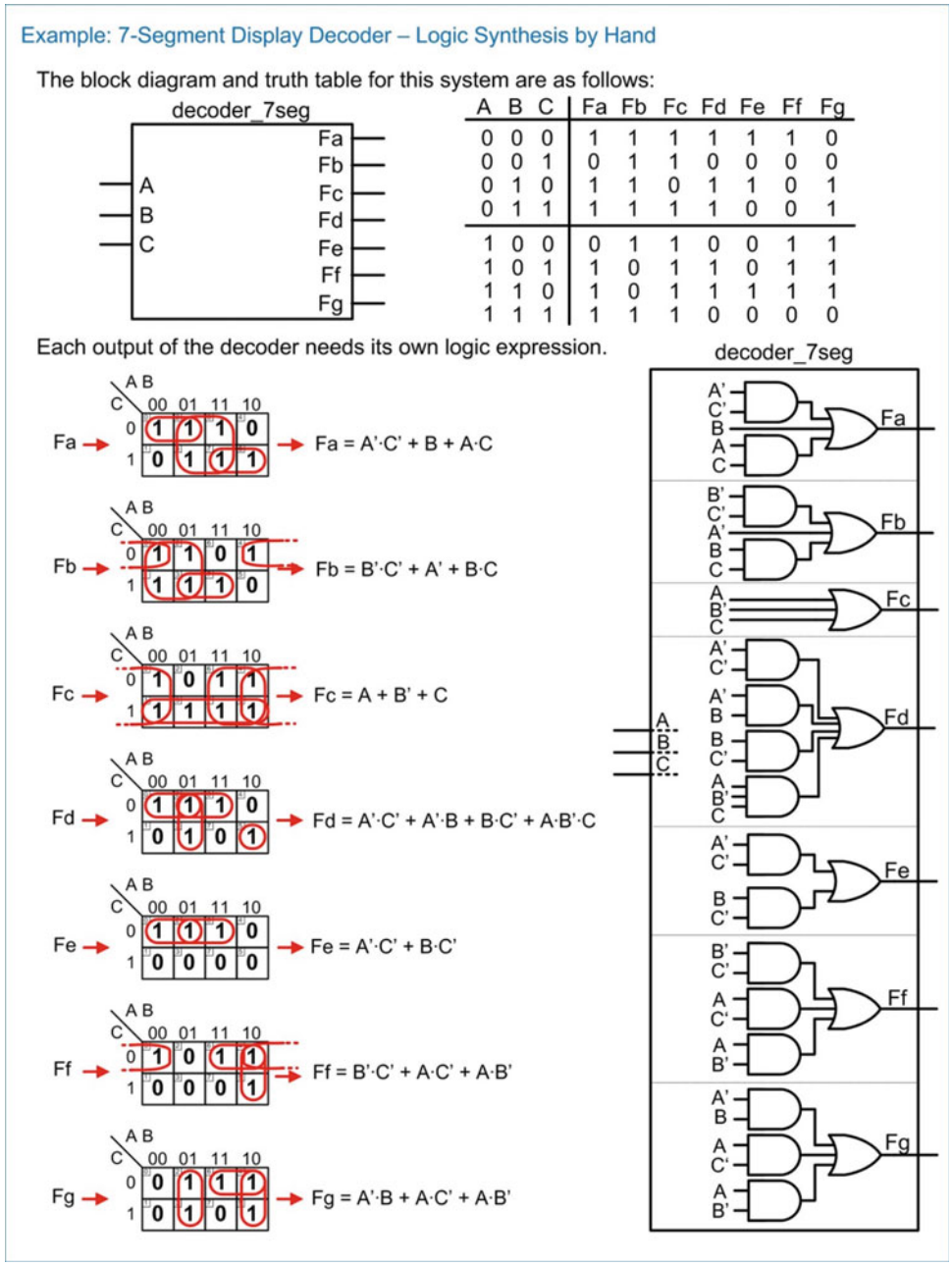
Example: 7-Segment Display Decoder - Truth Table



A	B	C		F_a	F_b	F_c	F_d	F_e	F_f	F_g
0	0	0		1	1	1	1	1	1	0
0	0	1		0	1	1	0	0	0	0
0	1	0		1	1	0	1	1	0	1
0	1	1		1	1	1	1	0	0	1
1	0	0		0	1	1	0	0	1	1
1	0	1		1	0	1	1	0	1	1
1	1	0		1	0	1	1	1	1	1
1	1	1		1	1	1	0	0	0	0

Example 3.4
7-segment display decoder—truth table

If we wish to model this decoder using logical operators, we need to first create the seven separate combinational logic expressions for each output. Each of the outputs (F_a–F_g) can be put into a 3-input K-map to find the minimized logic expression. Example 3.5 shows the derivation of the logic expressions for the decoder from the truth table in Example 3.4 using Karnaugh maps.

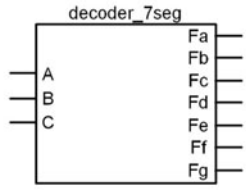


Example 3.5
7-segment display decoder—logic synthesis by hand

Now these seven logic expressions can be modeled in Verilog. Example 3.6 shows how to model the 7-segment decoder in Verilog using continuous assignment with logic operators.

Example: 7-Segment Display Decoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



A	B	C	Fa	Fb	Fc	Fd	Fe	Ff	Fg
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

```
module decoder_7seg (output wire Fa, Fb, Fc, Fd, Fe, Ff, Fg,
                     input  wire A, B, C);

   assign Fa = (~A & ~C) | (B) | (A & C);
   assign Fb = (~B & ~C) | (~A) | (B & C);
   assign Fc = (A) | (~B) | (C);
   assign Fd = (~A & ~C) | (~A & B) | (B & ~C) | (A & ~B & C);
   assign Fe = (~A & ~C) | (B & ~C);
   assign Ff = (~B & ~C) | (A & ~C) | (A & ~B);
   assign Fg = (~A & B) | (A & ~C) | (A & ~B);

endmodule
```

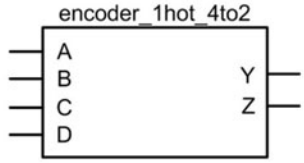
Example 3.6
7-segment display decoder—Verilog modeling using logical operators

3.2.4 Logical Operator Example: One-Hot Encoder

A one-hot binary encoder has *n* outputs and 2^n inputs. The output will be an n-bit, binary code which corresponds to an assertion on one and only one of the inputs. Example 3.7 shows the process of designing a 4-to-2 binary encoder by hand (i.e., using the classical digital design approach) in order to find the logic expression to model in Verilog using logical operators.

Example: 4-to-2 Binary Encoder – Logic Synthesis by Hand

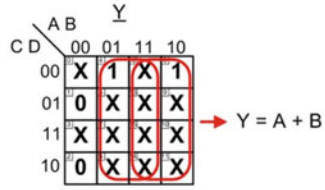
The block diagram and truth table for this system are as follows:

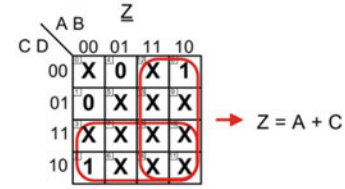


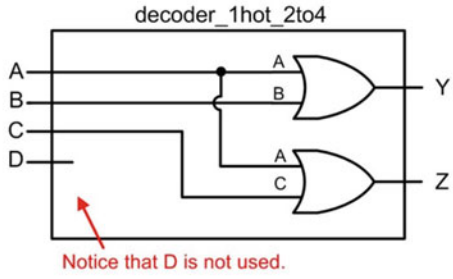
A	B	C	D	Y	Z
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

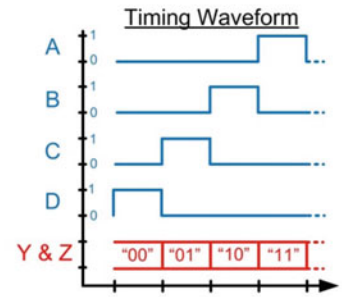
D => "00"
C => "01"
B => "10"
A => "11"

When designing this circuit, each output needs to have its own separate combinational logic circuit. When constructing the K-maps for Y and Z, each will have 4-inputs (A, B, C, D). The output values for many of the input codes are not specified in the above truth table. As such, we can use Don't Cares (X) to simplify the logic.







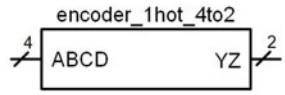


Example 3.7
4-to-2 binary encoder—logic synthesis by hand

Example 3.8 shows how to model the encoder with continuous assignments and logical operators using the logic expressions from Example 3.7.

Example: 4-to-2 Binary Encoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



ABCD	YZ
"0001"	"00"
"0010"	"01"
"0100"	"10"
"1000"	"11"

The following implements the behavior of the encoder with continuous assignment and logical operators.

```
module encoder_1hot_4to2 (output wire [1:0] YZ,
                          input  wire [3:0] ABCD);

   assign YZ[1] = ABCD[3] | ABCD[2];
   assign YZ[0] = ABCD[3] | ABCD[1];

endmodule
```

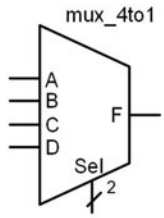
Example 3.8
4-to-2 binary encoder—Verilog modeling using logical operators

3.2.5 Logical Operator Example: Multiplexer

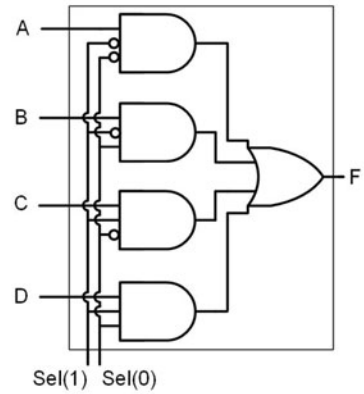
A multiplexer is a circuit that passes one of its multiple inputs to a single output based on a select input. This can be thought of as a digital switch. The multiplexer has *n* select lines, 2^n inputs, and *one* output. Example 3.9 shows the process of modeling a 4-to-1 multiplexer using continuous signal assignments and logical operators.

Example: 4-to-1 Multiplexer – Verilog Modeling using Logical Operators

The symbol and truth table for a 4-to-1 multiplexer are shown below. This can be implemented using a simple sum of products form based on the identity theorem and the appropriate inversions of the select line.



Sel	F
"00"	A
"01"	B
"10"	C
"11"	D



The following shows how to model the behavior of the mux using continuous assignment and logical operators.

```
module mux_4to1 (output wire F,
                 input  wire A, B, C, D,
                 input  wire [1:0] Sel);

   assign F = (A & ~Sel[1] & ~Sel[0]) |
              (B & ~Sel[1] &  Sel[0]) |
              (C &  Sel[1] & ~Sel[0]) |
              (D &  Sel[1] &  Sel[0]);

endmodule
```

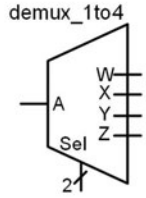
Example 3.9
4-to-1 multiplexer—Verilog modeling using logical operators

3.2.6 Logical Operator Example: Demultiplexer

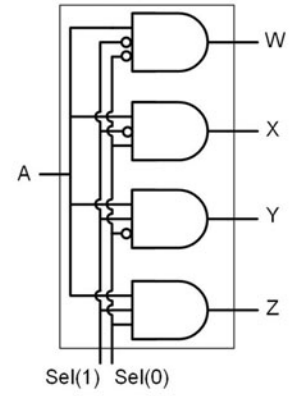
A demultiplexer works in a complementary fashion to a multiplexer. A demultiplexer has one input that is routed to one of its multiple outputs. The output that is active is dictated by a select input. A demux has *n* select lines that chooses to route the input to one of its 2^n outputs. When an output is not selected, it outputs a logic 0. Example 3.10 shows how to model the demultiplexer in Verilog using continuous assignments and logical operators.

Example: 1-to-4 Demultiplexer – Verilog Modeling using Logical Operators

The symbol and truth table for the 1-to-4 demultiplexer are shown below. This can be implemented using set of simple product terms based on the identity theorem and the appropriate inversions of the select line.



Sel	W	X	Y	Z
"00"	A	0	0	0
"01"	0	A	0	0
"10"	0	0	A	0
"11"	0	0	0	A



The following shows the behavior of the demux using continuous assignments with logical operators.

```
module demux_1to4 (output wire W, X, Y, Z,
                   input  wire A,
                   input  wire [1:0] Sel);

   assign W = (A & ~Sel[1] & ~Sel[0]);
   assign X = (A & ~Sel[1] &  Sel[0]);
   assign Y = (A &  Sel[1] & ~Sel[0]);
   assign Z = (A &  Sel[1] &  Sel[0]);

endmodule
```

Example 3.10
1-to-4 demultiplexer—Verilog modeling using logical operators

Concept Check

CC3.2 Why does modeling combinational logic in its canonical form with continuous assignment and logical operators defeat the purpose of the modern digital design flow?

A) It requires the designer to first create the circuit using the classical digital design approach and then enter it into the HDL in a form that is essentially a text-based netlist. This does not take advantage of the abstraction capabilities and automated synthesis in the modern flow.

B) It cannot be synthesized because the order of precedence of the logical operators in Verilog does not match the precedence defined in Boolean algebra.

C) The circuit is in its simplest form so there is no work for the synthesizer to do.

D) It does not allow an *else* clause to cover the outputs for any remaining input codes not explicitly listed.

3.3 Continuous Assignment with Conditional Operators

Logical operators are good for describing the behavior of small circuits; however, in the prior examples we still needed to create the canonical sum of products logic expression by hand before describing the functionality with logical operators. The true power of an HDL is when the behavior of the

system can be described fully without requiring any hand design. The conditional operator allows us to describe a continuous assignment using Boolean conditions that effect the values of the result. In this approach, we use the conditional operator (**?**) in conjunction with the continuous assignment keyword **assign**.

3.3.1 Conditional Operator Example: SOP Circuit

Example 3.11 shows how to design a Verilog model of a combinational logic circuit using continuous assignment with conditional operators. Note that this example uses the same truth table as in Example 3.2 to illustrate a comparison between approaches.

Example: Modeling Combinational Logic using Continuous Assignment with Conditional Operators (1)

Implement the following truth table using a continuous assignment with conditional operators.

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

We can implement the entire truth table in its current form by nesting conditional operators to explicitly list out each possible input code and its corresponding output as follows:

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

   assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b0) && (C == 1'b0)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
               1'b0;

endmodule
```

We can reduce the length of this model by only explicitly listing the input conditions for when the output is TRUE and allowing the final FALSE value to cover all other inputs.

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

   assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
               1'b0;

endmodule
```

Example 3.11
Combinational logic using continuous assignment with conditional operators (1)

In the prior example, the conditional operator was based on a truth table. Conditional operators can also be used to model logic expressions. Example 3.12 shows how to design a Verilog model of a combinational logic circuit when the logic expression is already known. Note that this example again uses the same truth table as in Examples 3.2 and 3.11 to illustrate a comparison between approaches.

Example: Modeling Combinational Logic using Continuous Assignment with Conditional Operators (2)

Implement the following truth table using a continuous assignment with conditional operators.

In this example, a K-map was used to find a minimized logic expression of:

$$F = C' \cdot (A' + B)$$

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

We can implement the conditional operator using input variables and Boolean operators to directly model the logic expression.

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

   assign F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;

endmodule
```

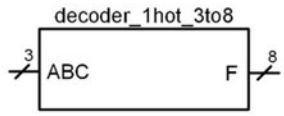
Example 3.12
Combinational logic using continuous assignment with conditional operators (2)

3.3.2 Conditional Operator Example: One-Hot Decoder

Example 3.13 shows how to model the 3-to-8 one-hot decoder in Verilog using continuous assignment with conditional operators. This description of a one-hot decoder can be simplified by using vector notation for the ports.

Example: 3-to-8 One-Hot Decoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows. Notice that the input and output ports now use vectors in order to create a more compact description.



ABC	F(7)	F(6)	F(5)	F(4)	F(3)	F(2)	F(1)	F(0)
"000"	0	0	0	0	0	0	0	1
"001"	0	0	0	0	0	0	1	0
"010"	0	0	0	0	0	1	0	0
"011"	0	0	0	0	1	0	0	0
"100"	0	0	0	1	0	0	0	0
"101"	0	0	1	0	0	0	0	0
"110"	0	1	0	0	0	0	0	0
"111"	1	0	0	0	0	0	0	0

The following shows a technique to model the decoder using continuous assignment with conditional operators. Note that the output will be "unknown" (X) if the input code is not one of the eight possible binary input values.

```
module decoder_1hot_3to8 (output wire [7:0] F,
                          input  wire [2:0] ABC);

   assign F = (ABC == 3'b000) ? 8'b0000_0001 :
              (ABC == 3'b001) ? 8'b0000_0010 :
              (ABC == 3'b010) ? 8'b0000_0100 :
              (ABC == 3'b011) ? 8'b0000_1000 :
              (ABC == 3'b100) ? 8'b0001_0000 :
              (ABC == 3'b101) ? 8'b0010_0000 :
              (ABC == 3'b110) ? 8'b0100_0000 :
              (ABC == 3'b111) ? 8'b1000_0000 :
             8'bXXXX_XXXX;

endmodule
```

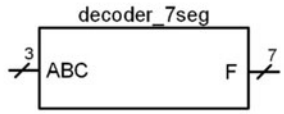
Example 3.13
3-to-8 one-hot decoder—Verilog modeling using conditional operators

3.3.3 Conditional Operator Example: 7-Segment Display Decoder

Example 3.14 shows how to model the 7-segment decoder in Verilog using continuous assignment with conditional operators. Again, a more compact description of the decoder can be accomplished if the ports are described as vectors.

Example: 7-Segment Decoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows:



ABC	a F(6)	b F(5)	c F(4)	d F(3)	e F(2)	f F(1)	g F(0)
"000"	1	1	1	1	1	1	0
"001"	0	1	1	0	0	0	0
"010"	1	1	0	1	1	0	1
"011"	1	1	1	1	0	0	1
"100"	0	1	1	0	0	1	1
"101"	1	0	1	1	0	1	1
"110"	1	0	1	1	1	1	1
"111"	1	1	1	0	0	0	0

The following shows a technique to model the decoder using continuous assignment with conditional operators.

```
module decoder_7seg (output wire [6:0] F,
                     input  wire [2:0] ABC);

   assign F = (ABC == 3'b000) ? 7'b111_1110 :
              (ABC == 3'b001) ? 7'b011_0000 :
              (ABC == 3'b010) ? 7'b110_1101 :
              (ABC == 3'b011) ? 7'b111_1001 :
              (ABC == 3'b100) ? 7'b011_0011 :
              (ABC == 3'b101) ? 7'b101_1011 :
              (ABC == 3'b110) ? 7'b101_1111 :
              (ABC == 3'b111) ? 7'b111_0000 :
              7'bXXXX_XXXX;

endmodule
```

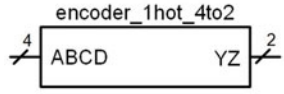
Example 3.14
7-segment display decoder—Verilog modeling using conditional operators

3.3.4 Conditional Operator Example: One-Hot Decoder

Example 3.15 shows how to model the encoder with continuous assignments and conditional operators. Notice that using this approach does not require synthesizing the logic expressions by hand but rather can model the functionality directly from the truth table.

Example: 4-to-2 Binary Encoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows:



ABCD	YZ
"0001"	"00"
"0010"	"01"
"0100"	"10"
"1000"	"11"

The following implements the behavior of the encoder with continuous assignment and conditional operators.

```
module encoder_1hot_4to2 (output wire [1:0] YZ,
                          input  wire [3:0] ABCD);

   assign YZ = (ABCD == 4'b0001) ? 2'b00 :
               (ABCD == 4'b0010) ? 2'b01 :
               (ABCD == 4'b0100) ? 2'b10 :
               (ABCD == 4'b1000) ? 2'b11 :
               2'bXX;
endmodule
```

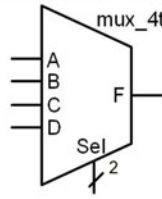
Example 3.15
4-to-2 binary encoder—Verilog modeling using conditional operators

3.3.5 Conditional Operator Example: Multiplexer

Example 3.16 shows the process of modeling a 4-to-1 multiplexer using continuous signal assignments and conditional operators. Notice that this approach can also be implemented directly from the truth table.

Example: 4-to-1 Multiplexer – Verilog Modeling using Conditional Operators

The symbol and truth table for the 4-to-1 multiplexer are as follows:



Sel	F
"00"	A
"01"	B
"10"	C
"11"	D

The following shows how to model the behavior of the mux using continuous assignment and conditional operators.

```
module mux_4to1 (output wire F,
                 input  wire A, B, C, D,
                 input  wire [1:0] Sel);

   assign F = (Sel == 2'b00) ? A :
              (Sel == 2'b01) ? B :
              (Sel == 2'b10) ? C :
              (Sel == 2'b11) ? D :
              1'bX;

endmodule
```

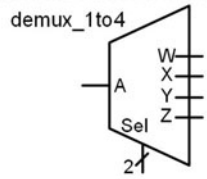
Example 3.16
4-to-1 multiplexer—Verilog modeling using conditional operators

3.3.6 Conditional Operator Example: Demultiplexer

Example 3.17 shows how to model the demultiplexer in Verilog using continuous assignments and conditional operators. Notice that this approach can be implemented directly from the truth table as well.

Example: 1-to-4 Demultiplexer – Verilog Modeling using Conditional Operators

The symbol and truth table for the 1-to-4 demultiplexer are as follows:



Sel	W	X	Y	Z
"00"	A	0	0	0
"01"	0	A	0	0
"10"	0	0	A	0
"11"	0	0	0	A

The following shows the behavior of the demux using continuous assignments with conditional operators.

```
module demux_1to4 (output wire W, X, Y, Z,
                   input  wire A,
                   input  wire [1:0] Sel);

   assign W = (Sel == 2'b00) ? A : 1'b0;
   assign X = (Sel == 2'b01) ? A : 1'b0;
   assign Y = (Sel == 2'b10) ? A : 1'b0;
   assign Z = (Sel == 2'b11) ? A : 1'b0;

endmodule
```

Example 3.17
1-to-4 demultiplexer—Verilog modeling using conditional operators

Concept Check

CC3.3 Why does a continuous signal assignment with conditional operators better reflect the modern digital design flow compared to using logical operators?

A) It allows the logic to be modeled directly from its functional description as opposed to from the final logic expressions, which must be determined prior to HDL modeling. This allows the continuous signal assignment approach to take advantage of automated synthesis and avoids any hand design.

B) A conditional operator has a final clause that covers any input cases not explicitly listed. This makes it more like a programming language operator.

C) A conditional operator has a final clause that covers any input cases not explicitly listed. This allows a final assignment of "X," which provides the ability to assign any outputs not explicitly listed to be treated as "unknowns."

D) The conditional operators can model the entire logic circuit in one assignment while the logical operator approach often takes multiple separate assignments.

3.4 Continuous Assignment with Delay

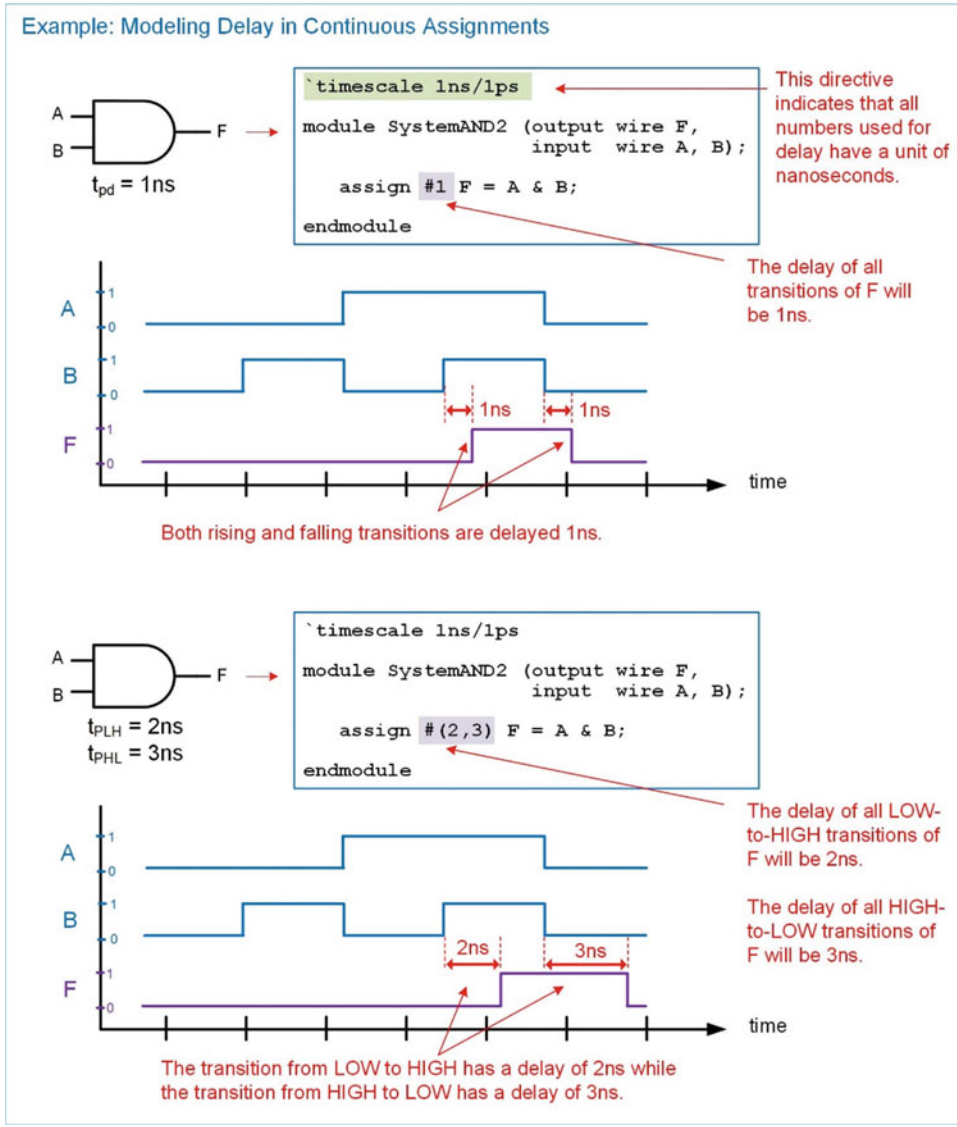
Verilog provides the ability to model gate delays when using a continuous assignment. The **#** is used to indicate a delayed assignment. For combinational logic circuits, the delay can be specified for all transitions, for rising and falling transitions separately, and for rising, falling, and transitions to the value *off* separately. A transition to *off* refers to a transition to Z. If only one delay parameter is specified, it is used to model all delays. If two delay parameters are specified, the first parameter is used for the rise time delay while the second is used to model the fall time delay. If three parameters are specified, the third parameter is used to model the transition to off. Parenthesis are optional but recommended when using multiple delay parameters.

```
assign #(<del_all>)                      <target_net> = <RHS_nets, operators, etc...>;
assign #(<del_rise, del_fall>)           <target_net> = <RHS_nets, operators, etc...>;
assign #(<del_rise, del_fall, del_off>)  <target_net> = <RHS_nets, operators, etc...>;
```

Example:

```
assign #1        F = A;   // Delay of 1 on all transitions.
assign #(2,3)    F = A;   // Delay of 2 for rising transitions and 3 for falling.
assign #(2,3,4)  F = A;   // Delay of 2 for rising, 3 for falling, and 4 for off transition.
```

When using delay, it is typical to include the **`timescale** directive to provide the units of the delay being specified. Example 3.18 shows a graphical depiction of using delay with continuous assignments when modeling combinational logic circuits.



Example 3.18
Modeling delay in continuous assignments

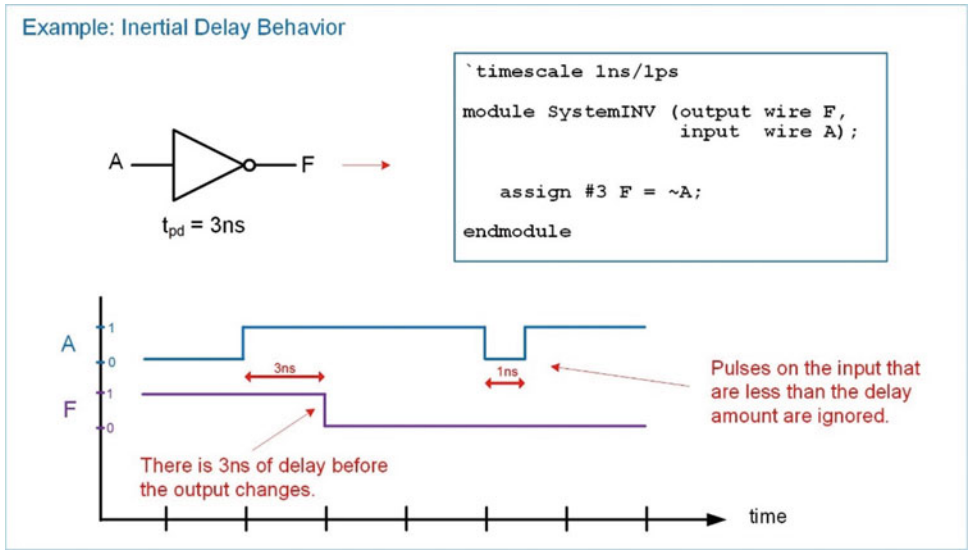
Verilog also provides a mechanism to model a range of delays that are selected by a switch set in the CAD compiler. There are three delays categories that can be specified: *minimum*, *typical*, and *maximum*. The delays are separated by a "**:**." The following is the syntax of how to use the delay range capability.

```
assign #(<min>:<typ>:<max>) <target_net> = <RHS_nets, operators, etc...>;
```

Example:

```
assign #(1:2:3)             F = A;  // Specifying a range of delays for all transitions.
assign #(1:1:2, 2:2:3)      F = A;  // Specifying a range of delays for rising/falling.
assign #(1:1:2, 2:2:3, 4:4:5) F = A;  // Specifying a range of delays for each transition.
```

The delay modeling capability in continuous assignment is designed to model the behavior of real combinational logic with respect to short duration pulses. When a pulse is shorter than the delay of the combinational logic gate, the pulse is ignored. Ignoring brief input pulses on the input accurately models the behavior of on-chip gates. When the input pulse is faster than the delay of the gate, the output of the gate does not have time to respond. As a result, there will not be a logic change on the output. This is called *inertial delay* modeling and is the default behavior when using continuous assignments. Example 3.19 shows a graphical depiction of inertial delay behavior in Verilog.



Example 3.19
Inertial delay modeling when using continuous assignment

Concept Check

CC3.4 Can a delayed signal assignment impact multiple continuous signal assignments?

A) Yes. If a signal assignment with delay is made to a signal that is also used as an input in a separate continuous signal assignment, then the delay will propagate through both assignments.

B) No. Only the assignment in which the delay is used will experience the delay.

Summary

- *Concurrency* is the term that describes operations being performed in parallel. This allows real-world system behavior to be modeled.
- Verilog provides the *continuous assignment* operator to support modeling concurrent combinational logic operations.
- Complex logic circuits can be implemented by using continuous assignment with *logical operators* or *conditional operators*.
- Delay can also be included in continuous assignments.
- Verilog supports a variety of delay models including delay for all transitions, separate delay for rising and falling transitions, separate delay for rising, falling, and transitions to off, and finally support for a min:typ:max delay that is selected by a compiler switch.

Exercise Problems

Section 3.1: Verilog Operators

3.1.1 What is the purpose of the continuous assignment operator?

3.1.2 If two continuous assignments are made to the same net, which one will take priority?

3.1.3 What is the difference between a bitwise logical AND (&) operation and a reduction AND (&) operation?

3.1.4 How is a conditional operator (?) similar to an if/then programming construct?

3.1.5 How many bits will the target vector F need to be if the following concatenation assignment is made?

F = {4'hA, 2'b00};

3.1.6 How many bits will the target vector F need to be if the following replication assignment is made?

F = {3{4'hA}};

3.1.7 When adding two unsigned vectors of different sizes using the + numerical operator, what happens to the smaller vector prior to the addition?

3.1.8 What operation has the highest precedence operation in Verilog?

Section 3.2: Continuous Assignment with Logical Operators

3.2.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

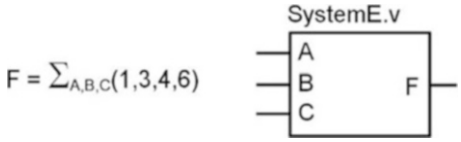


Fig. 3.1
System E functionality

3.2.2 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

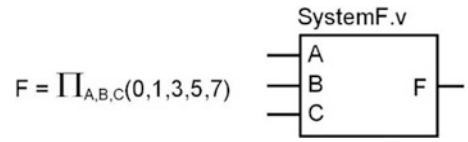


Fig. 3.2
System F functionality

3.2.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

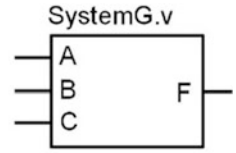


Fig. 3.3
System G functionality

3.2.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

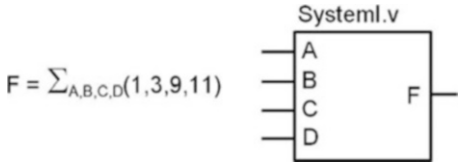


Fig. 3.4
System I functionality

3.2.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

Fig. 3.5
System J functionality

3.2.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

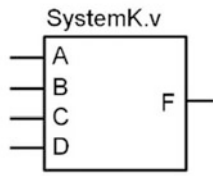
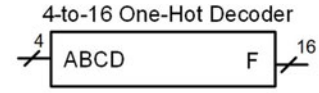


Fig. 3.6
System K functionality

3.2.7 Design a Verilog model for the 4-to-16 one-hot decoder shown in Fig. 3.7. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

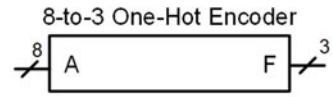


ABCD[3:0]	F[15:0]
0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 1 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 1 1 1	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1 0 0 1	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1 0 1 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 1 1	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
module decoder_1hot_4to16
  (output wire [15:0] F,
   input  wire [3:0]  ABCD);
                      :
```

Fig. 3.7
4-to-16 one-hot decoder functionality

3.2.8 Design a Verilog model for the 8-to-3 one-hot encoder shown in Fig. 3.8. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.



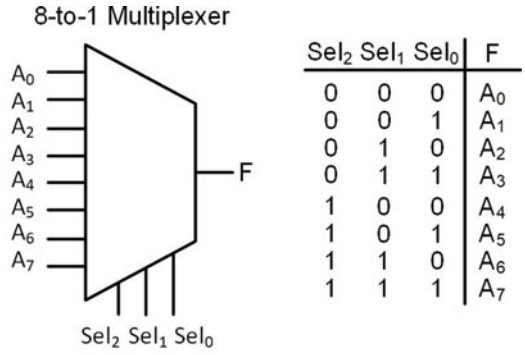
A[7:0]	F[2:0]
0 0 0 0 0 0 0 1	0 0 0
0 0 0 0 0 0 1 0	0 0 1
0 0 0 0 0 1 0 0	0 1 0
0 0 0 0 1 0 0 0	0 1 1
0 0 0 1 0 0 0 0	1 0 0
0 0 1 0 0 0 0 0	1 0 1
0 1 0 0 0 0 0 0	1 1 0
1 0 0 0 0 0 0 0	1 1 1

```
module encoder_8to3_binary
  (output wire [2:0] F,
   input  wire [7:0] A);
                    :
```

Fig. 3.8
8-to-3 one-hot encoder functionality

3.2.9 Design a Verilog model for the 8-to-1 multiplexer shown in Fig. 3.9. Use continuous

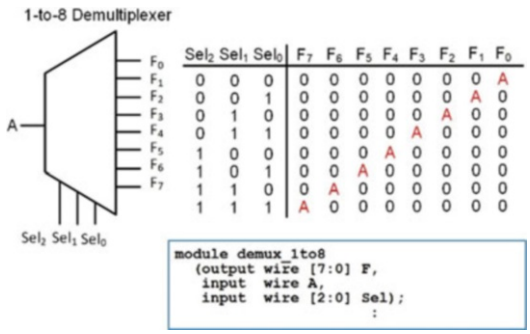
assignment and logical operators. Declare your module and ports to match the block diagram provided.




```
module mux_8to1
  (output wire F,
   input  wire [7:0] A,
   input  wire [2:0] Sel);
                   :
```

Fig. 3.9
8-to-1 multiplexer functionality

3.2.10 Design a Verilog model for the 1-to-8 demultiplexer shown in Fig. 3.10. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.




```
module demux_1to8
  (output wire [7:0] F,
   input  wire A,
   input  wire [2:0] Sel);
                   :
```

Fig. 3.10
1-to-8 demultiplexer functionality

Section 3.3: Continuous Assignment with Conditional Operators

3.3.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.2 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.7 Design a Verilog model for the 4-to-16 one-hot decoder shown in Fig. 3.7. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided.

3.3.8 Design a Verilog model for the 8-to-3 one-hot encoder shown in Fig. 3.8. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided.

3.3.9 Design a Verilog model for the 8-to-1 multiplexer shown in Fig. 3.9. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided.

3.3.10 Design a Verilog model for the 1-to-8 demultiplexer shown in Fig. 3.10. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided.

Section 3.4: Continuous Assignment with Delay

3.4.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with logical operators and give each logic operation 1 ns of delay. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.2 Design a Verilog model to implement the behavior described by the 3-input maxterm

list shown in Fig. 3.2. Use continuous assignment with logical operators and give each rising transition a delay of 1 ns and each falling transition a delay of 2 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with conditional operators and give the entire logic operation a delay of 3 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment with conditional operators and give rising transitions a delay of 3 ns and falling transitions a delay of 2 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and logical operators and give each logic operation a delay of 1 ns, 2 ns, and 3 ns respectively for the operation's min:typ:max behavior. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and conditional operators and give the entire operation a delay of 1 ns, 2 ns, and 3 ns respectively for the operation's min:typ:max behavior. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

Chapter 4: Structural Design and Hierarchy

This chapter describes how to accomplish hierarchy within Verilog using lower-level subsystems. Structural design in Verilog refers to including lower-level subsystems within a higher-level module in order to produce the desired functionality. This is called *hierarchy* and is a good design practice because it enables design partitioning. A purely structural design will not contain any behavioral constructs in the module such as signal assignments, but instead just contain the instantiation and interconnections of other subsystems. A subsystem in Verilog is simply another module that is called by a higher-level module. Each lower-level module that is called is executed concurrently by the calling module.

Learning Outcomes—After completing this chapter, you will be able to:

4.1 Instantiate and map the ports of a lower-level component in Verilog
4.2 Design a Verilog model for a system that uses hierarchy

4.1 Structural Design Constructs

4.1.1 Lower-Level Module Instantiation

The term *instantiation* refers to the *use* or *inclusion* of a lower-level module within a system. In Verilog, the syntax for instantiating a lower-level module is as follows.

```
module_name <instance_identifier> (port mapping...);
```

The first portion of the instantiation is the module name that is being called. This must match the lower level module name exactly, including case. The second portion of the instantiation is an optional instance identifier. Instance identifier is useful when instantiating multiple instances of the same lower-level module. The final portion of the instantiation is the port mapping. There are two techniques to connect signals to the ports of the lower-level module, *explicit* and *positional*.

4.1.2 Port Mapping

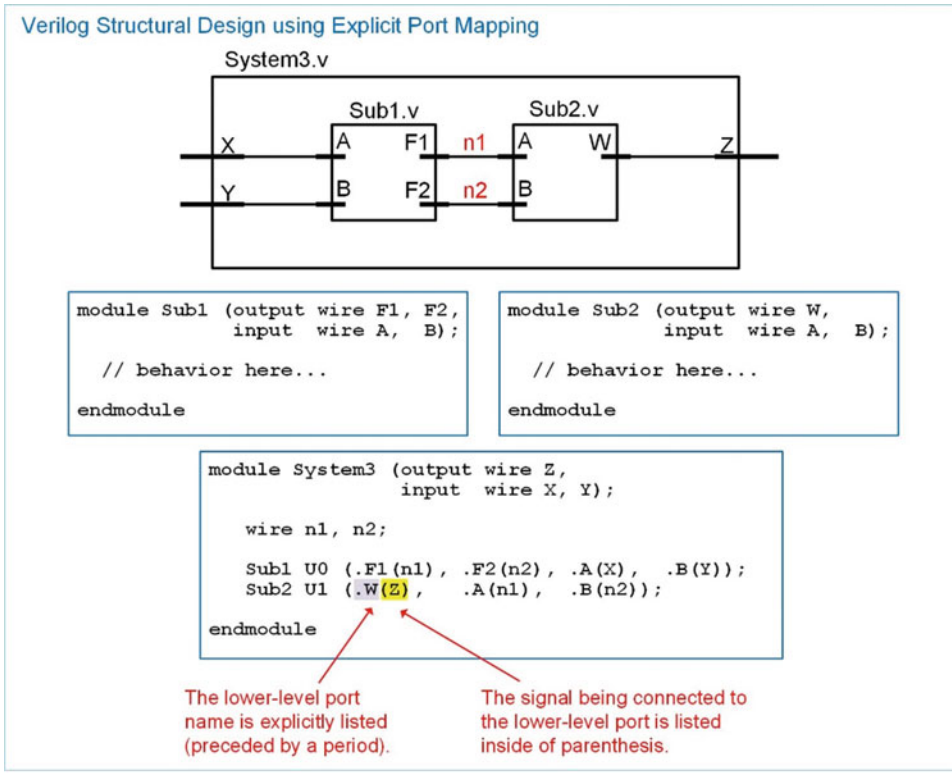
4.1.2.1 Explicit Port Mapping

In explicit port mapping, the names of the ports of the lower-level subsystem are provided along with the signals they are being connected to. The lower-level port name is preceded with a period (.) while the signal it is being connected is enclosed within parenthesis. The port connections can be listed in any order since the details of the connection (i.e., port name to signal name) are explicit. Each connection is separated by a comma. The syntax for explicit port mapping is as follows:

```
module_name <instance identifier> (.port_name1(signal1), .port_name2(signal2),
etc.);
```

Example 4.1 shows how to design a Verilog model of a hierarchical system that consists of two lower-level modules.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_4



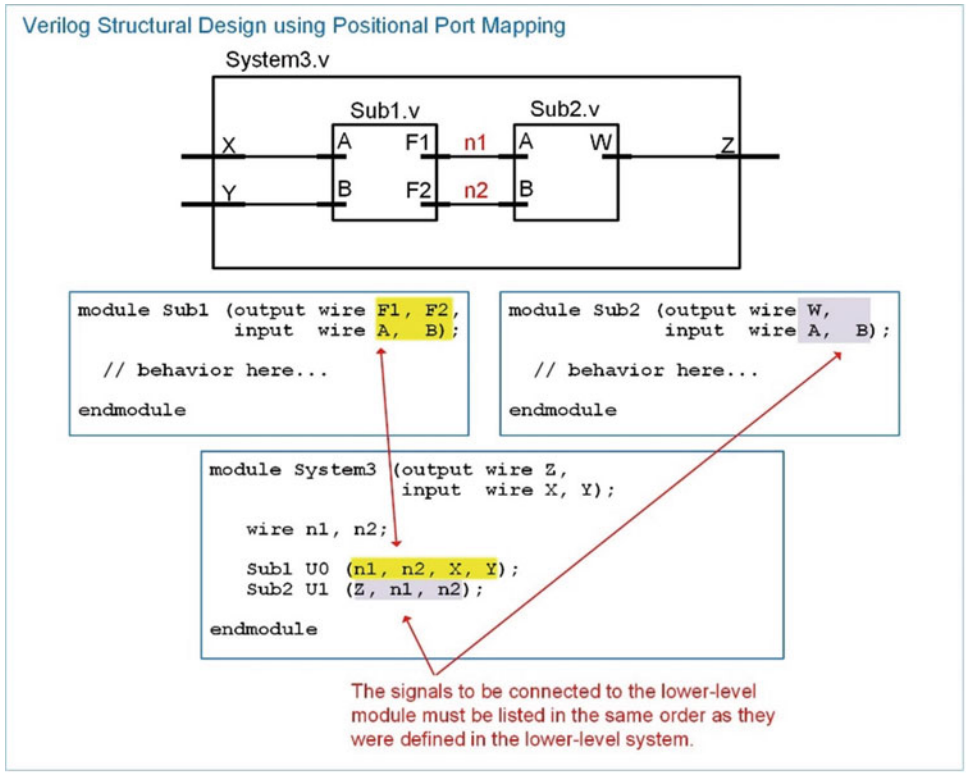
Example 4.1
Verilog structural design using explicit port mapping

4.1.2.2 Positional Port Mapping

In positional port mapping, the names of the ports of the lower-level modules are not explicitly listed. Instead, the signals to be connected to the lower-level system are listed in the same order in which the ports were defined in the subsystem. Each signal name is separated by a comma. This approach requires less text to describe the connection but can also lead to misconnections due to inadvertent mistakes in the signal order. The syntax for positional port mapping is as follows:

```
module_name <instance_identifier> (signal1, signal2, etc.);
```

Example 4.2 shows how to create the same structural Verilog model as in Example 4.1, but using positional port mapping instead.



Example 4.2
Verilog structural design using positional port mapping

4.1.3 Gate Level Primitives

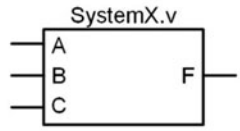
Verilog provides the ability to model basic logic functionality through the use of *primitives*. A primitive is a logic operation that is simple enough that it does not require explicit modeling. An example of this behavior can be a basic logic gate or even a truth table. Verilog provides a set of *gate level primitives* to model simple logic operations. These gate level primitives are **not()**, **and()**, **nand()**, **or()**, **nor()**, **xor()**, and **xnor()**. Each of these primitives are instantiated as lower-level subsystems with positional port mapping. The port order for each primitive has the output listed first followed by the input(s). The output and each of the inputs are scalars. Gate level primitives do not need to be explicitly created as they are provided as part of the Verilog standard. One of the benefits of using gate level primitives is that the number of inputs are easily scaled as each primitive can accommodate an increasing number of inputs automatically. Furthermore, modeling using this approach essentially provides a gate level netlist, so it represents a very low-level, detailed gate level implementation that is ready for technology mapping. Example 4.3 shows how to use gate level primitives to model the behavior of a combinational logic circuit.

Example: Modeling Combinational Logic using Gate Level Primitives

Implement the following truth table using gate level primitives.

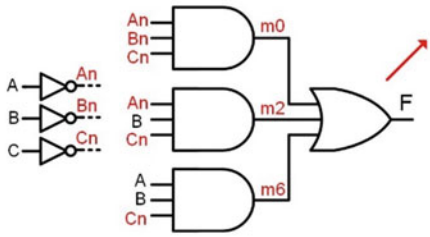
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Let's call the design SystemX and implement its logic as a canonical SOP logic expression.



$F = \sum_{A,B,C}(0,2,6) = A'\cdot B'\cdot C' + A'\cdot B\cdot C' + A\cdot B\cdot C'$

The corresponding logic diagram is as follows. From this, we can create the Verilog model directly with gate level primitives.



The output is always listed first in the port mapping when using gate level primites.

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

   wire  An, Bn, Cn;  // internal nets
   wire  m0, m2, m6;

   not U0 (An, A);            // Not's
   not U1 (Bn, B);
   not U2 (Cn, C);

   and U3 (m0, An, Bn, Cn);  // AND's
   and U4 (m2, An, B,  Cn);
   and U5 (m6, A,  B,  Cn);

   or  U6 (F, m0, m2, m6);   // OR

endmodule
```

Example 4.3
Modeling combinational logic circuits using gate level primitives

4.1.4 User-Defined Primitives

A **user-defined primitive** (UDP) is a system that describes the behavior of a low-level module using a logic table. This is very useful for creating combinational logic functionality that will be used numerous times. UDPs are also useful for large truth tables where it is more convenient to list the functionality in table form. UDPs are lower-level subsystems that are intended to be instantiated in higher-level modules just like gate level primitives, with the exception that the UDP needs to be created in its own file. The syntax for a UDP is as follows:

```
primitive primitive_name (output output_name,
                          input input_name1, input_name2, ...);
   table
     in1_val in2_val ... : out_val;
     in1_val in2_val ... : out_val;
           :
   endtable
endprimitive
```

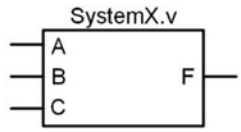
A UDP must list its output(s) first in the port definition. It also does not require types to be defined for the ports. For combinational logic UDPs, all ports are assumed to be of type wire. Example 4.4 shows how to design a user-defined primitive to implement a combinational logic circuit.

Example: Modeling Combinational Logic with a User-Defined Level Primitives

Implement the following truth table with a user-defined primitives.

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

Let's call the design SystemX. We will create a simple module for SystemX that defines the ports and then calls the UDP.



The user-defined primitive will be called SystemX_UDP and will contain the table describing the desired functionality.

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

  SystemX_UDP U0 (F, A, B, C);

endmodule
```

The top-level module simply instantiates the UDP.

```
primitive SystemX_UDP (output F,
                       input  A, B, C);

   table
    // A B C : F
       0 0 0 : 1;
       0 0 1 : 0;
       0 1 0 : 1;
       0 1 1 : 0;
       1 0 0 : 0;
       1 0 1 : 0;
       1 1 0 : 1;
       1 1 1 : 0;
   endtable

endprimitive
```

UDPs require that the output be listed first in the port definition.

It is helpful to insert a comment above the table values to list the location of the port names within the table.

Notice that the inputs are listed first, in the order they appear in the port declaration, followed by a ":" and the output.

Example 4.4
Modeling combinational logic circuits with a user-defined primitive

4.1.5 Adding Delay to Primitives

Delay can be added to primitives using the same approach as described in Sect. 3.4. The delay is inserted after the primitive name but before the instance name.

Example:

```
not #2 U0 (An, A);                // Gate level primitive for an inverter with delay of 2.
and #3 U3 (m0, An, Bn, Cn);       // Gate level primitive for an AND gate with delay of 3.
SystemX_UDP #1 U0 (F, A, B, C);   // UDP with a delay of 1.
```

Concept Check

CC4.1 Does the use of lower-level submodules model concurrent functionality? Why?

A) No. Since the lower-level behavior of the module being instantiated may contain nonconcurrent behavior, it is not known what functionality will be modeled.

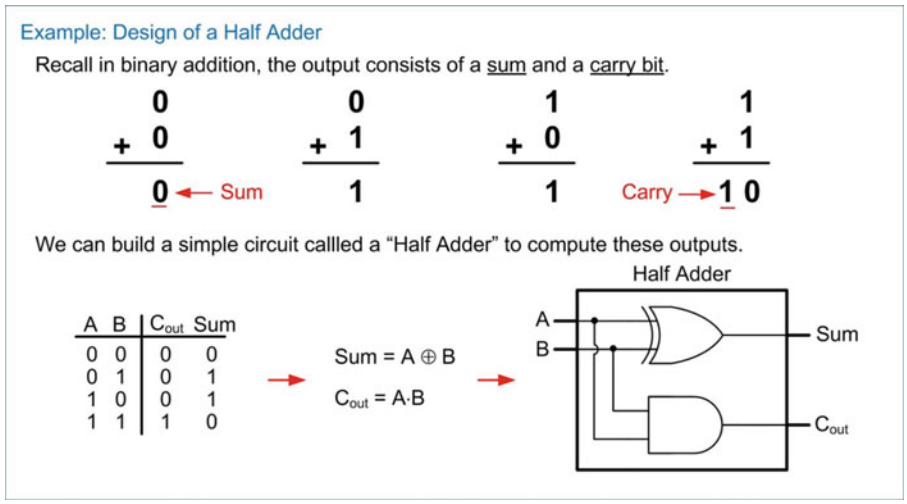
B) Yes. The modules are treated like independent subsystems whose behavior runs in parallel just as if separate parts were placed in a design.

4.2 Structural Design Example: Ripple Carry Adder

This section gives an example of a structural design that implements a simple binary adder.

4.2.1 Half Adders

When creating an adder, it is desirable to design incremental subsystems that can be reused. This reduces design effort and minimizes troubleshooting complexity. The most basic component in the adder is called a *half adder*. This circuit computes the sum and carry out on two input arguments. The reason it is called a half adder instead of a full adder is because it does not accommodate a *carry in* during the computation, thus it does not provide all of the necessary functionality required for a positional adder. Example 4.5 shows the design of a half adder. Notice that two combinational logic circuits are required in order to produce the sum (the XOR gate) and the carry out (the AND gate). These two gates are in parallel to each other, thus the delay through the half adder is due to only one level of logic.

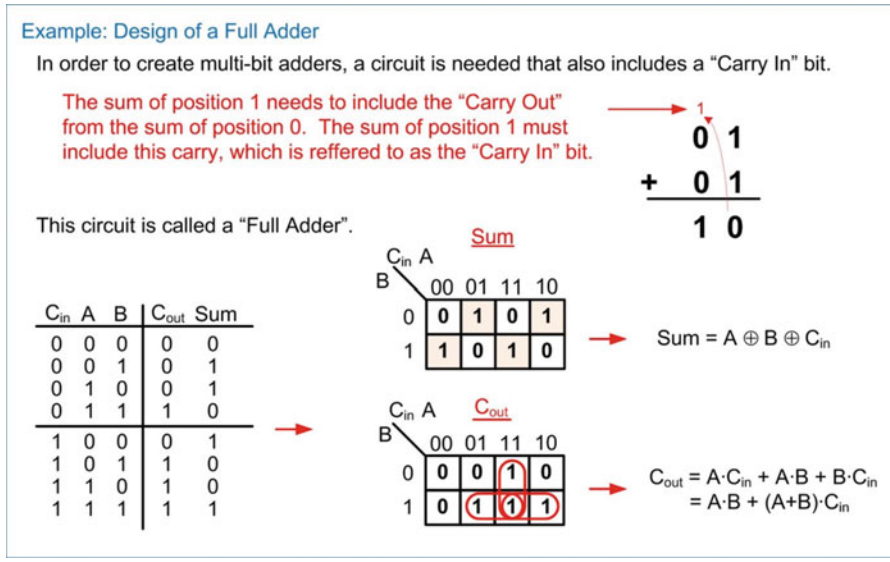


Example 4.5
Design of a half adder

4.2.2 Full Adders

A full adder is a circuit that still produces a sum and carry out, but considers three inputs in the computations (A, B, and C_{in}). Example 4.6 shows the design of a full adder using the classical design

approach. This step is shown to illustrate why it is possible to reuse half adders to create the full adder. In order to do this, it is necessary to have the minimal sum of products logic expression.



Example 4.6
Design of a full adder

As mentioned before, it is desirable to reuse design components as we construct more complex systems. One such design reuse approach is to create a full adder using two half adders. This is straightforward for the sum output since the logic is simply two cascaded XOR gates (Sum = A ⊕ B ⊕ Cin). The carry out is not as straightforward. Notice that the expression for C_{out} derived in Example 4.6 contains the term (A + B). If this term could be manipulated to use an XOR gate instead, it would allow the full adder to take advantage of existing circuitry in the system. Figure 4.1 shows a derivation of an equivalency that allows (A + B) to be replaced with (A ⊕ B) in the Cout logic expression.

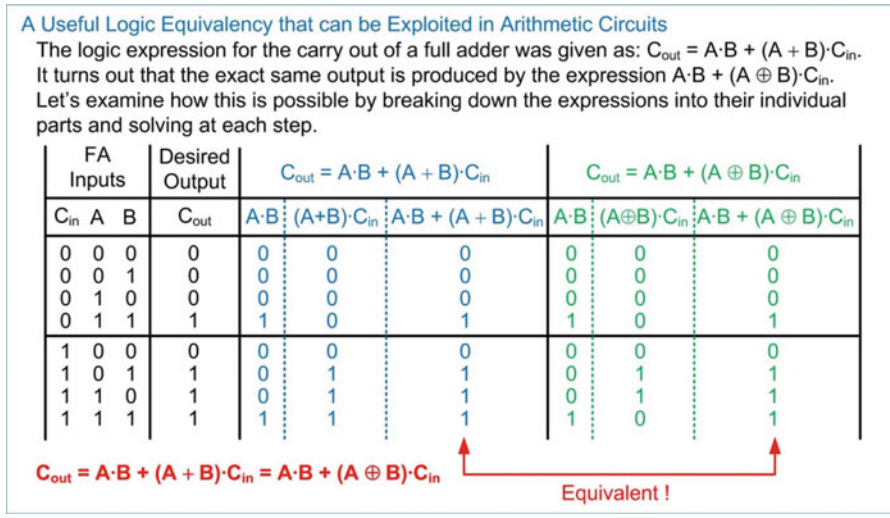
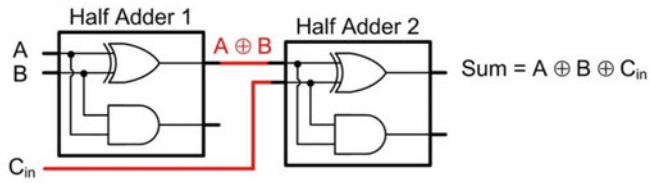


Fig. 4.1
A useful logic equivalency that can be exploited in arithmetic circuits

The ability to implement the carry out logic using the expression $C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$ allows us to implement a full adder with two half adders and the addition of a single OR gate. Example 4.7 shows this approach. In this new configuration, the sum is produced in two levels of logic while the carry out is produced in three levels of logic.

Example – Design of a Full Adder Out of Two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



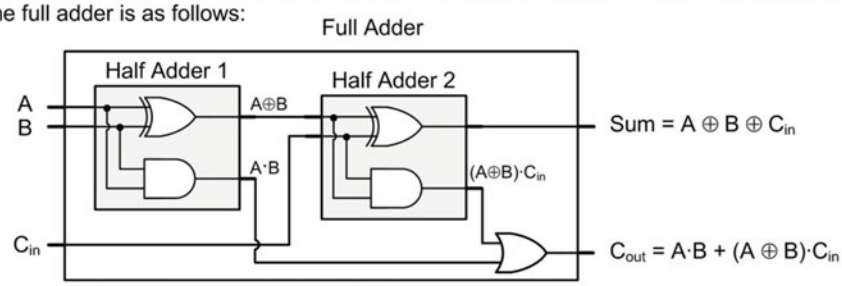
The expression for the "Carry Out" of the full adder is:

$$C_{out} = A \cdot B + (A + B) \cdot C_{in}$$

or

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

Notice that the carry out of Half Adder 1 produces the $A \cdot B$ term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B) \cdot C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:



Example 4.7
Design of a full adder out of half adders

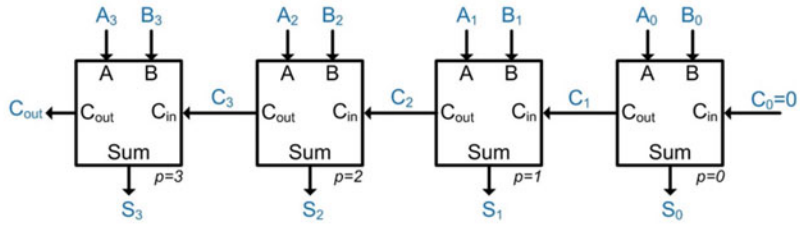
4.2.3 Ripple Carry Adder (RCA)

The full adder can now be used in the creation of multibit adders. The simplest topology exploiting the full adder is called a *ripple carry adder* (RCA). In this approach, full adders are used to create the sum and carry out of each bit position. The carry out of each full adder is used as the carry in for the next higher position. Since each subsequent full adder needs to wait for the carry to be produced by the preceding stage, the carry is said to *ripple* through the circuit, thus giving this approach its name.

Example 4.8 shows how to design a 4-bit ripple carry adder using a chain of full adders. Notice that the carry in for the full adder in position 0 is tied to a logic 0. The 0 input has no impact on the result of the sum but enables a full adder to be used in the 0th position.

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.

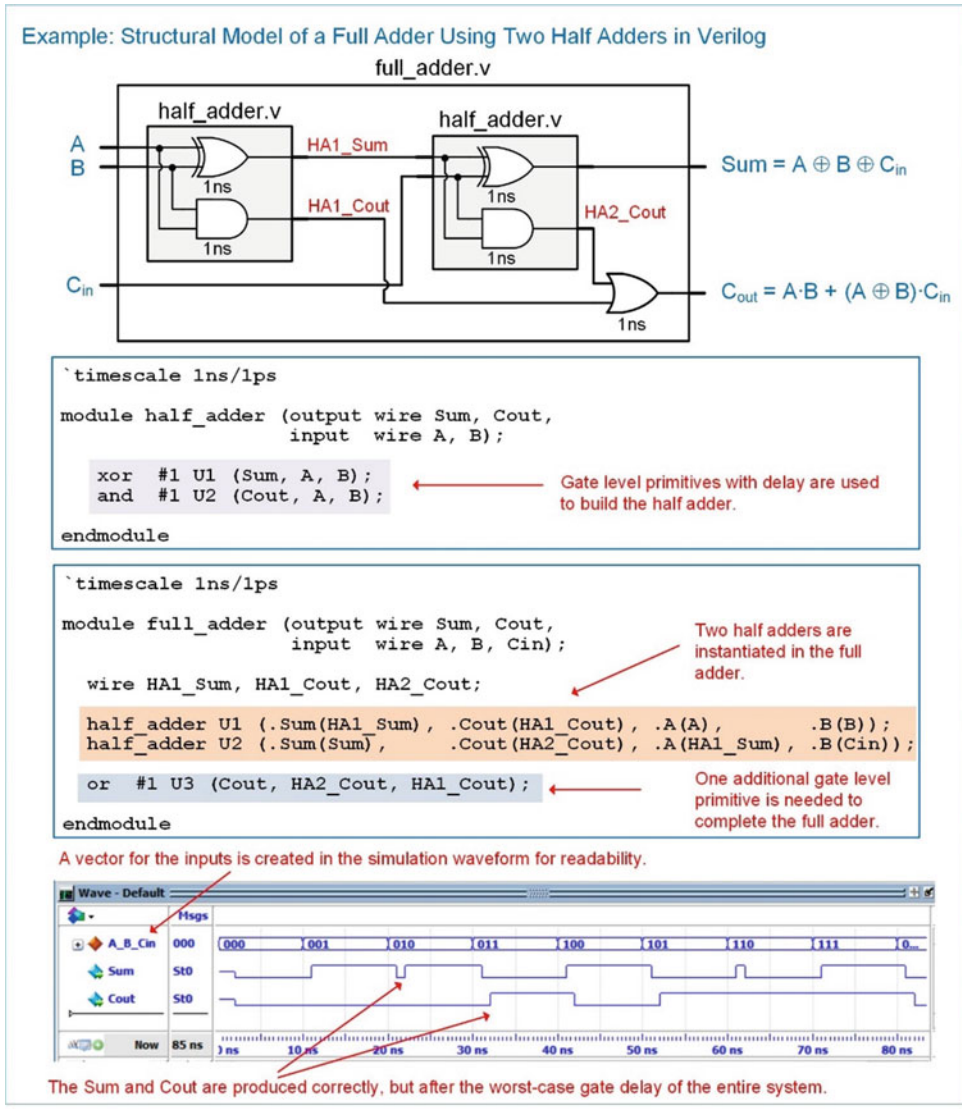


The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

Example 4.8
Design of a 4-bit ripple carry adder (RCA)

4.2.4 Structural Model of a Ripple Carry Adder in Verilog

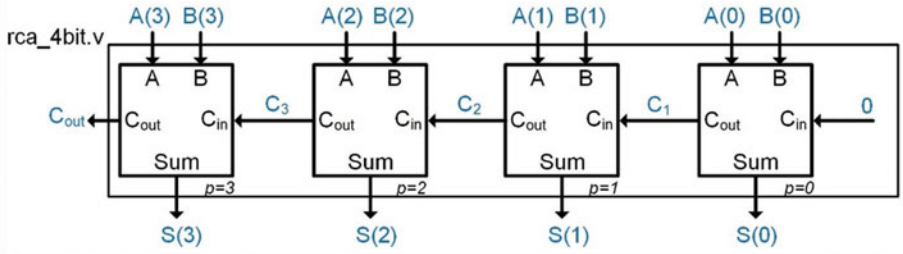
Now that the hierarchical design of the RCA is complete, we can now model it in Verilog as a system of lower-level modules. Example 4.9 shows the structural model for a full adder in Verilog consisting of two half adders. The full adder is created by instantiating two versions of the half adder as subsystems. The half adder in this example is implemented using gate level primitives. In this example, all gates are modeled with a delay of 1 ns.



Example 4.9
Structural model of a full adder using two half adders

Example 4.10 shows the structural model of a 4-bit ripple carry adder in Verilog. The RCA is created by instantiating four full adders. Notice that a logic 1'b0 can be directly inserted into the port map of the first full adder to model the behavior of $C_0 = 0$.

Example: Structural Model of a 4-Bit Ripple Carry Adder in Verilog




```
`timescale 1ns/1ps

module rca_4bit (output wire [3:0] Sum,
                 output wire       Cout,
                 input  wire [3:0] A, B);

  wire C1, C2, C3;

  full_adder U1 (.Sum(Sum[0]), .Cout(C1),   .A(A[0]), .B(B[0]), .Cin(1'b0));
  full_adder U2 (.Sum(Sum[1]), .Cout(C2),   .A(A[1]), .B(B[1]), .Cin(C1));
  full_adder U3 (.Sum(Sum[2]), .Cout(C3),   .A(A[2]), .B(B[2]), .Cin(C2));
  full_adder U4 (.Sum(Sum[3]), .Cout(Cout), .A(A[3]), .B(B[3]), .Cin(C3));

endmodule
```

A fixed value can be inserted into the port map of a sub-system. This handles the Cin port for the first full_adder.

Example 4.10
Structural model of a 4-bit ripple carry adder in Verilog

Concept Check

CC4.2 Why is the use of hierarchy considered a good design practice?

A) Hierarchy allows the design to be broken into smaller pieces, each with simpler functionality that can be verified independently prior to being used in a higher-level system.

B) Hierarchy allows a large system to be broken into smaller subsystems that can be designed by multiple engineers, thus decreasing the overall development time.

C) Hierarchy allows a large system to be broken down into smaller subsystems that can be more easily understood so that debugging is more manageable.

D) All of the above.

Summary

- Instantiating other modules from within a higher-level module is how Verilog implements hierarchy. A lower-level module can be instantiated as many times as desired. An instance identifier is useful is keeping track of each instantiation.
- The ports of a lower-level module can be connected using either *explicit* or *positional port mapping*.
- Verilog sub-systems are also treated as concurrent subsystems.
- Gate level primitives are provided in Verilog to implement basic logic functions (not, and, nand, or, nor, xor, xnor). These primitives are instantiated just like any other lower-level subsystem.
- User-Defined Primitives are supported in Verilog that allow the functionality of a circuit to be described in table form.

Exercise Problems

Section 4.1: Structural Design Constructs

4.1.1 How many times can a lower-level module be instantiated?

4.1.2 Which port mapping technique is more compact, explicit or positional?

4.1.3 Which port mapping technique is less prone to connection errors because the names of the lower-level ports are listed within the mapping?

4.1.4 Would it make sense to design a lower-level module to implement an AND gate in Verilog?

4.1.5 When would it make more sense to build a user-defined primitive instead of modeling the logic using continuous assignments?

Section 4.2: Structural Design Examples

4.2.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 4.2. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

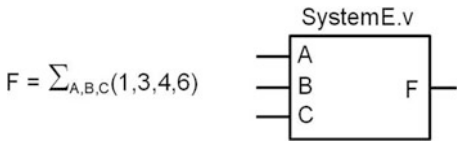


Fig. 4.2
System E functionality

4.2.2 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 4.2. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.3 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 4.3. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

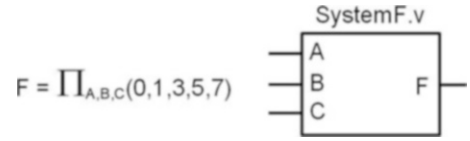


Fig. 4.3
System F functionality

4.2.4 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 4.3. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.5 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 4.4. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

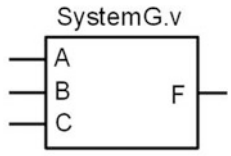


Fig. 4.4
System G functionality

4.2.6 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 4.4. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.7 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 4.5. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

$F = \sum_{A,B,C,D}(1,3,9,11)$

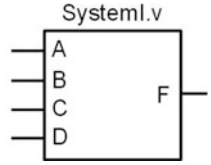


Fig. 4.5
System I functionality

4.2.8 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 4.5. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.9 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 4.6. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

$F = \prod_{A,B,C,D}(0,1,2,3,6,8,9,10,11,14)$

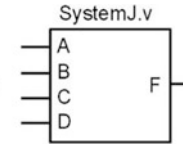


Fig. 4.6
System J functionality

4.2.10 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 4.6. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.11 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 4.7. Use a structural design approach based on gate level primitives. This is considered *structural* because you will need to instantiate the gate level primitives just like a traditional subsystem; however, you do not need to create the gate level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate level primitives. You can use whatever approach you prefer to create the logic expression (i.e.,

canonical SOP/POS, minimized SOP/POS, etc.). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

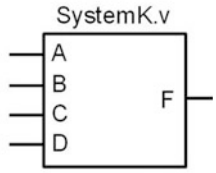


Fig. 4.7
System K functionality

4.2.12 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 4.7. Use a structural design approach based on a user- defined primitive. This is considered *structural* because you will need to instantiate the user- defined primitive just like a traditional subsystem. You will need to create both the upper-level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

Chapter 5: Modeling Sequential Functionality

In Chap. 3, techniques were presented to describe the behavior of concurrent systems. The modeling techniques presented were appropriate for combinational logic because these types of circuits have outputs dependent only on the current values of their inputs. This means a model that continuously performs signal assignments is an accurate model of this circuit behavior. When we start looking at sequential circuits (i.e., D-Flip-Flops, registers, finite state machine, and counters), these devices only update their outputs based upon an event, most often the edge of a clock signal. The modeling techniques presented in Chap. 3 are unable to accurately describe this type of behavior. In this chapter, we describe the Verilog constructs to model signal assignments that are triggered by an event to accurately model sequential logic. We can then use these techniques to describe more complex sequential logic circuits such as finite state machines and register transfer level systems.

Learning Outcomes—After completing this chapter, you will be able to:

5.1 Describe the behavior of a Verilog procedural block and how it is used to model sequential logic circuits
5.2 Model combinational logic circuits using a procedural block and conditional programming constructs
5.3 Use Verilog system tasks to provide additional functionality to a simulation model

5.1 Procedural Assignment

Verilog uses *procedural assignment* to model signal assignments that are based on an event. An *event* is most commonly a transition of a signal. This provides the ability to model sequential logic circuits such as D-flip-flops and finite state machines by triggering assignments off of a clock edge. Procedural assignments can only drive variable data types (i.e., reg, integer, real, and time), thus they are ideal for modeling storage devices. Procedural signal assignments can be evaluated in the order they are listed; thus they are able to model sequential assignments.

A procedural assignment can also be used to model combinational logic circuits by making signal assignments when any of the inputs to the model change. Despite the left-hand side of the assignment not being able to be of type wire in procedural assignment, modern synthesizers will recognize properly designed combinational logic models and produce the correct circuit implementation. Procedural assignment also supports standard programming constructs such as if-else decisions, case statements, and loops. This makes procedural assignment a powerful modeling approach in Verilog and is the most common technique for designing digital systems and creating test benches.

5.1.1 Procedural Blocks

All procedural signal assignments must be enclosed within a procedural *block*. Verilog has two types of procedural blocks, *initial* and *always*.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_5

5.1.1.1 Initial Blocks

An initial block will execute all of the statements embedded within it one time at the beginning of the simulation. An initial block is not used to model synthesizable behavior. It is instead used within test benches to either set the initial values of repetitive signals or to model the behavior of a signal that only has a single set of transitions. The following is the syntax for an initial block.

```
initial
 begin                // an optional ": name" can be added after the begin keyword
  signal_assignment_1
  signal_assignment_2
            :
 end
```

Let us look at a simple model of how an initial block is used to model the reset line in a test bench. In the following example, the signal "Reset_TB" is being driven into a DUT. At the beginning of the simulation, the initial value of Reset_TB is set to a logic zero. The second assignment will take place after a delay of 15 time units. The second assignment statement sets Reset_TB to a logic one. The assignments in this example are evaluated in sequence in the order they are listed due to the delay operator. Since the initial block executes only once, Reset_TB will stay at the value of its last assignment for the remainder of the simulation.

Example:

```
initial
 begin
      Reset_TB = 1'b0;
  #15 Reset_TB = 1'b1;
 end
```

5.1.1.2 Always Blocks

An *always* block will execute forever, or for the duration of the simulation. An always block can be used to model synthesizable circuits in addition to nonsynthesizable behavior in test benches. The following is the syntax for an always block.

```
always
 begin
  signal_assignment_1
  signal_assignment_2
    :
 end
```

Let us look at a simple model of how an always block can be used to model a clock line in a test bench. In the following example, the value of the signal Clock_TB will continuously change its logic value every 10 time units.

Example:

```
always
  begin
   #10 Clock_TB = ~Clock_TB;
end
```

By itself, the above always block will not work because when the simulation begins, Clock_TB does not have an initial value so the simulator will not know what the value of Clock_TB is at time zero. It will also not know what the output of the negation operation (~) will be at time unit 10. The following example

shows the correct way of modeling a clock signal using a combination of initial and always blocks. Verilog allows assignments to the same variable from multiple procedural blocks, so the following example is valid. Note that when the simulation begins, Clock_TB is assigned a logic zero. This provides a known value for the signal at time zero and also allows the always block negation to have a deterministic value. The example below will create a clock signal that will toggle every 10 time units.

Example:

```
initial
  begin
    Clock_TB = 1'b0;
  end

always
  begin
    #10 Clock_TB = ~Clock_TB;
  end
```

5.1.1.3 Sensitivity Lists

A *sensitivity list* is used in conjunction with a procedural block to trigger when the assignments within the block are executed. The symbol **@** is used to indicate a sensitivity list. Signals can then be listed within parenthesis after the @ symbol that will trigger the procedural block. The following is the base syntax for a sensitivity list.

```
always @ (signal1, signal2)
  begin
    signal_assignment_1
    signal_assignment_2
        :
  end
```

In this syntax, any transition on any of the signals listed within the parenthesis will cause the always block to trigger and all of its assignments to take place one time. After the always block ends, it will await the next signal transition in the sensitivity list to trigger again. The following example shows how to model a simple 3-input AND gate. In this example, any transition on inputs A, B, or C will cause the block to trigger and the assignment F to occur.

Example:

```
always @ (A, B, C)
  begin
    F = A & B & C;
  end
```

Verilog also supports keywords to limit triggering of the block to only rising edge or falling edge transitions. The keywords are **posedge** and **negedge**. The following is the base syntax for an edge-sensitive block. In this syntax, only rising edge transitions on signal1 or falling edge transitions on signal2 will cause the block to trigger.

```
always @ (posedge signal1, negedge signal2)
  begin
    signal_assignment_1
    signal_assignment_2
          :
  end
```

Sensitivity lists can also contain Boolean operators to more explicitly describe behavior. The following syntax is identical to the syntax above.

```
always @ (posedge signal1 or negedge signal2)
  begin
    signal_assignment_1
    signal_assignment_2
            :
  end
```

The ability to model edge sensitivity allows us to model sequential circuits. The following example shows how to model a simple D-flip-flop.

Example:

```
always @ (posedge Clock)
  begin
    Q = D;   // Note: This model does not include a reset.
  end
```

In Verilog-2001, the syntax to support sensitivity lists that will trigger based on any signal listed on the right-hand side of any assignment within the block was added. This syntax is **@***. The following example is how to use this modeling approach to model a 3-input AND gate.

Example:

```
always @*
  begin
    F = A & B & C;
  end
```

5.1.2 Procedural Statements

There are two kinds of signal assignments that can be used within a procedural block, **blocking** and **nonblocking**.

5.1.2.1 Blocking Assignments

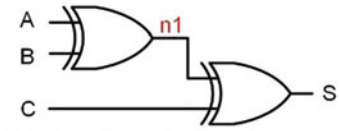
A *blocking assignment* is denoted with the = symbol and the evaluation and assignment of each statement takes place immediately. Each assignment within the block is executed in parallel. When this behavior is coupled with a sensitivity list that contains all of the inputs to the system, this approach can model synthesizable combinational logic circuits. This approach provides the same functionality as continuous assignments outside of a procedural block. The reason that designers use blocking assignments instead of continuous assignment is that more advanced programming constructs are supported within Verilog procedural blocks. These will be covered in the next section. Example 5.1 shows how to use blocking assignments within a procedural block to model a combinational logic circuit.

Example: Using Blocking Assignments to Model Combinational Logic

In this model, each of the inputs A, B, and C are listed in the sensitivity list so that the procedural block is triggered on any input transition. When using blocking assignments, the assignments inside of the block are evaluated and executed immediately. These two behaviors allow us to model combinational logic.

```
module BlockingEx1 (output reg  S,
                    input  wire A, B, C);

  reg  n1;

  always @ (A, B, C)
   begin
     n1 = A ^ B;     // statement 1
     S  = n1 ^ C;    // statement 2
   end

endmodule
```

Resulting Circuit



Both statement 1 and statement 2 are treated as separate circuits that execute concurrently when using blocking assignments.

Example 5.1
Using blocking assignments to model combinational logic

5.1.2.2 Nonblocking Assignments

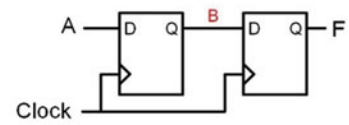
A *nonblocking assignment* is denoted with the **<=** symbol. When using nonblocking assignments, the assignment to the target signal is deferred until the end of the procedural block. This allows the assignments to be executed in the order they are listed in the block without cascading interim assignments through the list. When this behavior is coupled with triggering the block off of a clock signal, this approach can model synthesizable sequential logic circuits. Example 5.2 shows an example of using nonblocking assignments to model a sequential logic circuit.

Example: Using Non-Blocking Assignments to Model Sequential Logic

In this model, the always block will only trigger on the rising edge of a clock. When using non-blocking assignments, the assignments inside of the block are only executed at the end of the block. These two behaviors allow us to model sequential logic.

```
module NonBlockingEx1 (output reg  F,
                       input  wire A,
                       input  wire Clock);
  reg  B;

  always @ (posedge Clock)
    begin
      B <= A;    // statement 1
      F <= B;    // statement 2
    end

endmodule
```

Resulting Circuit



Notice that the value of B in statement 2 is not immediately updated with the assignment made in statement 1 due to the nature of non-blocking assignments.

Example 5.2
Using nonblocking assignments to model sequential logic

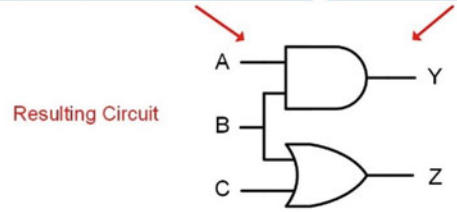
The difference between blocking and nonblocking assignments is subtle and is often one of the most difficult concepts to grasp when first learning Verilog. One source of confusion comes from the fact that blocking and nonblocking assignments *can* produce the same results when they either contain a single assignment or a list of assignments that do not have any signal interdependencies. A *signal interdependency* refers to when a signal that is the target of an assignment (i.e., on the LHS of an assignment) is used as an argument (i.e., on the RHS of an assignment) in subsequent statements. Example 5.3 shows two models that produce the same results regardless of whether a blocking or nonblocking assignment is used.

Example: Identical Behavior when using Blocking vs. Non-Blocking Assignments

In these models, there are no signal interdependencies between statement 1 and statement 2. This means regardless of whether the assignments are made instantaneously (left) or at the end of the always block (right), the results are the same.

```
module BlockingEx2
 (output reg  Y, Z,
  input  wire A, B, C);

  always @ (A, B, C)
   begin
     Y = A & B;     // statement 1
     Z = B | C;     // statement 2
   end

endmodule
```

```
module NonBlockingEx2
 (output reg  Y, Z,
  input  wire A, B, C);

  always @ (A, B, C)
   begin
     Y <= A & B;    // statement 1
     Z <= B | C;    // statement 2
   end

endmodule
```



Resulting Circuit

Both modeling approaches yield the same result because there are no signal interdependences between the statements.

Example 5.3
Identical behavior when using blocking versus nonblocking assignments

When a list of statements within a procedural block *does* have signal interdependencies, blocking and nonblocking assignments will have different behavior. Example 5.4 shows how signal interdependencies will cause different behavior between blocking and nonblocking assignments. In this example, all inputs are listed in the sensitivity list with the intent of modeling combinational logic.

Example: Different Behavior when using Blocking vs. Non-Blocking Assignments (1)

In these examples, there is a signal interdependency and all of the inputs are listed in the sensitivity list. By listing all of the inputs in the sensitivity list, the intent is to model combinational logic such that the outputs update anytime there is a change on the inputs.

CORRECT

```
module BlockingEx3
 (output reg  S,
  input  wire A, B, C);

 reg  n1;

 always @ (A, B, C)
  begin
    n1 = A ^ B;   // statement 1
    S  = n1 ^ C;  // statement 2
  end
```

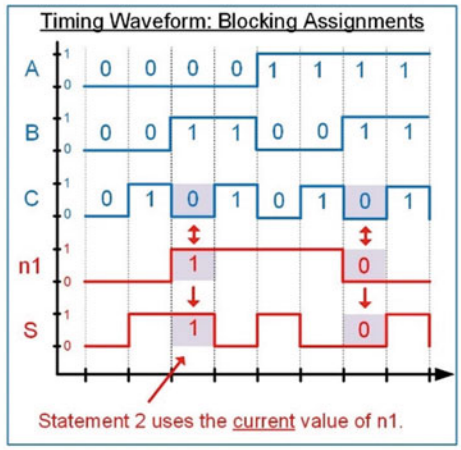
```
module NonBlockingEx3
 (output reg  S,
  input  wire A, B, C);

 reg  n1;

 always @ (A, B, C)
  begin
    n1 <= A ^ B;  // statement 1
    S  <= n1 ^ C; // statement 2
  end
```

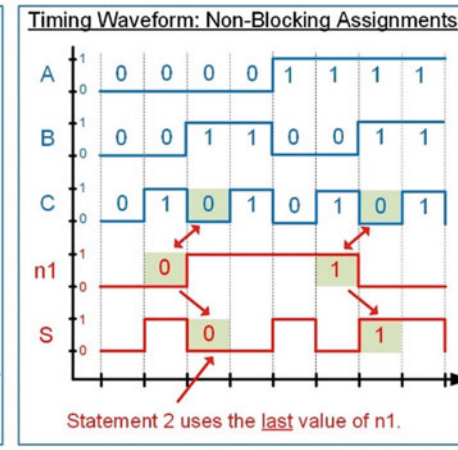
In both cases, statement 1 (the assignment to **n1**) will produce the same result. This is because the assignment only depends on the inputs A and B. Since A and B are listed in the sensitivity list, any change on them will trigger the block and their current value will be used in the assignment to n1.

However, statement 2 (the assignment to **S**) contains a signal interdependency and will NOT produce the same result in both cases.

Blocking Assignment Case (=): Blocking assignments take place immediately. This means that the assignment to n1 in statement 1 takes place immediately and the updated value of n1 is used in statement 2. When used in conjunction with listing all of the inputs in the sensitivity list, this approach successfully models **combinational logic**.

Non-Blocking Assignment Case (<=): Non-blocking assignments take place at the end of the procedural block. This means that the assignment to n1 in statement 1 does not take place before the assignment in statement 2. The value of n1 used in statement 2 will be the value of n1 when the block is triggered, not the new value of n1 assigned within the block. Said another way, the value of n1 used in statement 2 will be the value of n1 from the "prior" time the block triggered, or the "last" value of n1. This does **not model combinational logic**.





Example 5.4
Different behavior when using blocking versus nonblocking assignments (1)

Example 5.5 shows another case where signal interdependencies will cause different behavior between blocking and nonblocking assignments. In this example, the procedural block is triggered by the rising edge of a clock signal with the intent of modeling two stages of sequential logic.

Example: Different Behavior when using Blocking vs. Non-Blocking Assignments (2)

In these examples, there is a signal interdependency and the sensitivity list is triggered by the edge of a clock. The intent of this model is to create a 2-stage sequential logic circuit such that the outputs only update when there is a rising edge of the clock.

```
module BlockingEx4
 (output reg  F,
  input  wire A,
  input  wire Clock);

  reg  B;

  always @ (posedge Clock)
    begin
      B = A;    // statement 1
      F = B;    // statement 2
    end

endmodule
```

CORRECT

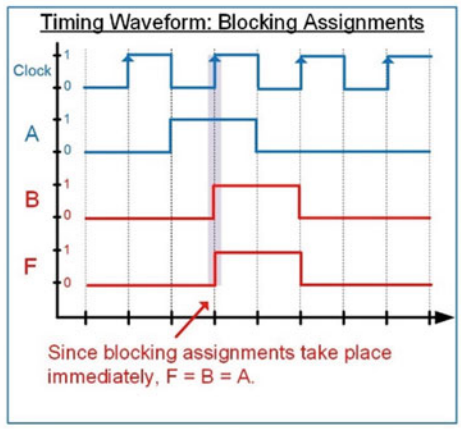
```
module NonBlockingEx4
 (output reg  F,
  input  wire A,
  input  wire Clock);

  reg  B;

  always @ (posedge Clock)
    begin
      B <= A;    // statement 1
      F <= B;    // statement 2
    end

endmodule
```

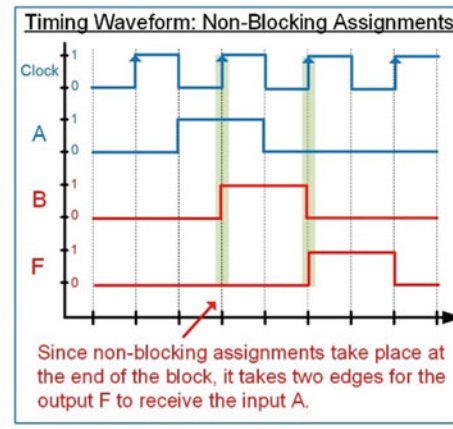
In both cases, the procedural block will trigger on the rising edge of a clock. Also in each case, the assignment in statement 1 will produce the same result.

However, statement 2 has a signal dependency and will NOT produce the same result in both cases.

Blocking Assignment Case (=): Blocking assignments take place immediately. This means that the assignment of A to B and then B to F will result in simply F = A. This will still result in sequential logic, but the output F will be updated with the input A on every rising edge of clock.

Non-Blocking Assignment Case (<=): Non-blocking assignments take place at the end of the procedural block. This means that an input on A will be stored to B on rising edge of clock, but not to F on the same edge. Instead, the subsequent clock edge will cause the result stored in B to be stored to F. This will result in sequential logic that has two edge triggered storage elements instead of just one as in the blocking example.





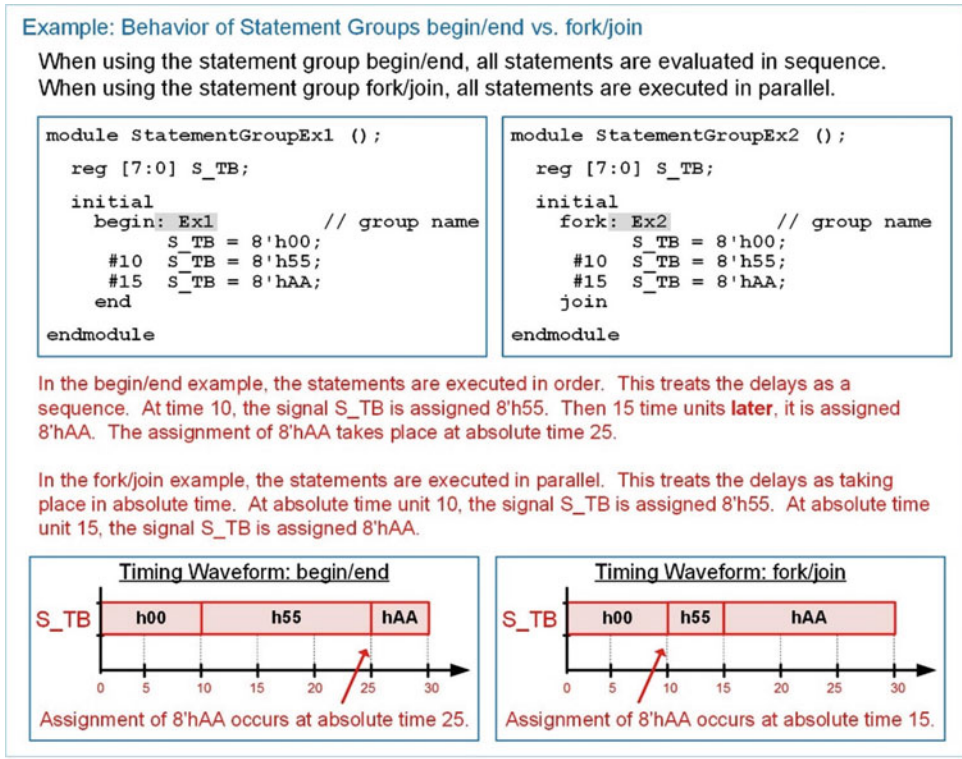
Example 5.5
Different behavior when using blocking versus nonblocking assignments (2)

While the behavior of these procedural assignments can be confusing, there are two design guidelines that can make creating accurate, synthesizable models straightforward. They are:

1. When modeling **combinational logic**, use *blocking assignments* and *list every input in the sensitivity list*.
2. When modeling **sequential logic**, use *nonblocking assignments* and *only list the clock and reset lines* (if applicable) in the sensitivity list.

5.1.3 Statement Groups

A statement group refers to how the statements in a block are processed. Verilog supports two types of statement groups: **begin/end** and **fork/join**. When using begin/end, all statements enclosed within the group will be evaluated in the order they are listed. When using a fork/join, all statements enclosed within the group will be evaluated in parallel. When there is only one statement within procedural block, a statement group is not needed. For multiple statements in a procedural block, a statement group is required. Statement groups can contain an optional name that is appended after the first keyword preceded by a ":." Example 5.6 shows a graphical depiction of the difference between begin/end and fork/join groups. Note that this example also shows the syntax for naming the statement groups.



Example 5.6
Behavior of statement groups begin/end versus fork/join

5.1.4 Local Variables

Local variables can be declared within a procedural block. The statement group must be named, and the variables will not be visible outside of the block. Variables can only be of variable type.

Example:

```
initial
 begin: stim_block  // it is required to name the block when declaring local variables
   integer i;       // local variables can only be of variable type
   i=2;
 end
```

CONCEPT CHECK

CC5.1 If a model of a combinational logic circuit excludes one of its inputs from the sensitivity list, what is the implied behavior?

A) A storage element because the output will be held at its last value when the unlisted input transitions.

B) An infinite loop.

C) A don't care will be used to form the minimal logic expression.

D) Not applicable because this syntax will not compile.

5.2 Conditional Programming Constructs

One of the more powerful features that procedural blocks provide in Verilog is the ability to use conditional programming constructs such as if-else decisions, case statements, and loops. These constructs are only available within a procedural block and can be used to model both combinational and sequential logic.

5.2.1 if-else Statements

An *if-else* statement provides a way to make conditional signal assignments based on Boolean conditions. The **if** portion of statement is followed by a Boolean condition that if evaluated TRUE will cause the signal assignment listed after it to be performed. If the Boolean condition is evaluated FALSE, the statements listed after the **else** portion are executed. If multiple statements are to be executed in either the if or else portion, then the statement group keywords begin/end need to be used. If only one statement is to be executed, then the statement group keywords are not needed. The else portion of the statement is not required and if omitted, no assignment will take place when the Boolean condition is evaluated FALSE. The syntax for an if-else statement is as follows:

```
if (<boolean_condition>)
   true_statement
else
   false_statement
```

The syntax for an if-else statement with multiple true/false statements is as follows:

```
if (<boolean_condition>)
  begin
   true_statement_1
   true_statement_2
  end
 else
  begin
   false_statement_1
   false_statement_2
  end
```

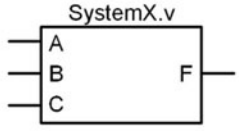
If more than one Boolean condition is required, additional if-else statements can be embedded within the *else* clause of the preceding if statement. The following shows an example of if-else statements implementing two Boolean conditions.

```
if (<boolean_condition_1>)
   true_statement_1
else if (<boolean_condition_2>)
   true_statement_2
else
   false_statement
```

Let us look at using an if-else statement to describe the behavior of a combinational logic circuit. Recall that a combinational logic circuit is one in which the output depends on the instantaneous values of the inputs. This behavior can be modeled by placing all of the inputs to the circuit in the sensitivity list of an always block and using blocking assignments. Using this approach, a change on any of the inputs in the sensitivity list will trigger the block and the assignments will take place immediately. Example 5.7 shows how to model a 3-input combinational logic circuit using if-else statements within a procedural always block.

Example: Using If-Else Statements to Model Combinational Logic

Implement the following truth table using an if-else statement within a procedural block.



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

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

  always @ (A, B, C)
    begin
      if       (A==1'b0 && B==1'b0 && C==1'b0)
        F = 1'b1;
      else if (A==1'b0 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else if (A==1'b1 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else
        F = 1'b0;
    end

endmodule
```

When modeling combinational logic using a procedural assignment, all of the inputs to the circuit must be listed in the sensitivity list and blocking assignments are used.

In this model, three nested if-else statements are used to explicitly describe the input conditions corresponding to an output of a one. For all other input codes, the else clause is used to drive the output to a zero.

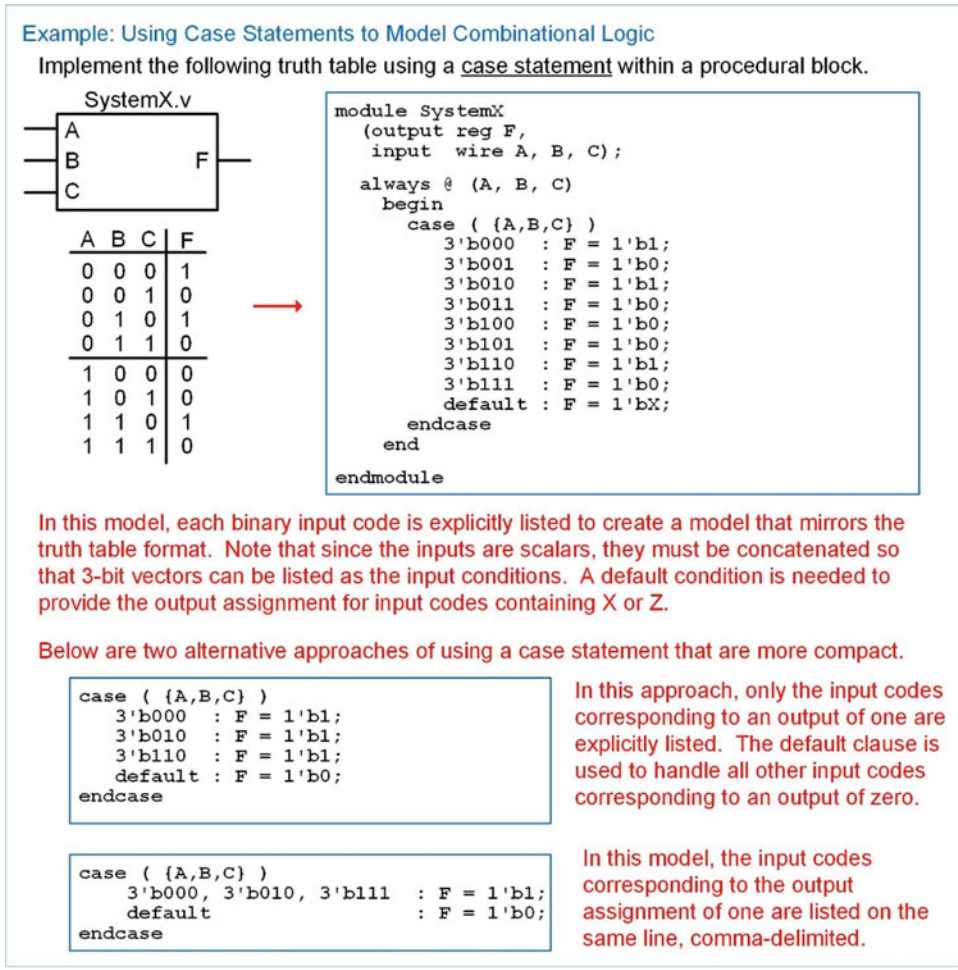
Example 5.7
Using if-else statements to model combinational logic

5.2.2 case Statements

A *case* statement is another technique to model signal assignments based on Boolean conditions. As with the if-else statement, a case statement can only be used inside of a procedural block. The statement begins with the keyword **case** followed by the input signal name that assignments will be based off of enclosed within parenthesis. The case statement can be based on multiple input signal names by concatenating the signals within the parenthesis. Then a series of input codes followed by the corresponding assignment is listed. The keyword **default** can be used to provide the desired signal assignment for any input codes not explicitly listed. When multiple input conditions have the same assignment statement, they can be listed on the same line comma-delimited to save space. The keyword **endcase** is used to denote the end of the case statement. The following is the syntax for a case statement.

```
case (<input_name>)
  input_val_1 : statement_1
  input_val_2 : statement_2
                :
  input_val_n : statement_n
  default     : default_statement
endcase
```

Example 5.8 shows how to model a 3-input combinational logic circuit using a case statement within a procedural block. Note in this example the inputs are scalars, so they must be concatenated so that the input values can be listed as 3-bit vectors. In this example, there are three versions of the model provided. The first explicitly lists out all binary input codes. This approach is more readable because it mirrors a truth table form. The second approach only lists the input codes corresponding to an output of one and uses the default clause to handle all other input codes. The third approach shows how to list multiple input codes with the same assignment on the same line using a comma-delimited series.



Example 5.8
Using case statements to model combinational logic

If-else statements can be embedded within a case statement and, conversely, case statements can be embedded within an if-else statement.

5.2.3 casez and casex Statements

Verilog provides two additional case statements that support don't cares in the input conditions. The **casez** statement allows the symbols **?** and **Z** to represent a don't care. The **casex** statement extends the casez statement by also interpreting **X** as a don't care. Care should be taken when using the casez and casex statements as they are easy to create unintended logic when using don't cares in the input codes.

5.2.4 forever Loops

A *loop* within Verilog provides a mechanism to perform repetitive assignments infinitely. This is useful in test benches for creating stimulus such as clocks or other periodic signals. We have already covered a looping construct in the form of an always block. An always block provides a loop with a starting condition. Verilog provides additional looping constructs to model more sophisticated behavior. All looping constructs must reside with a procedural block.

The simplest looping construct is the **forever** loop. As with other conditional programming constructs, if multiple statements are associated with the forever loop they must be enclosed within a statement group. If only one statement is used, the statement group is not needed. A forever loop within an initial block provides identical behavior as an always loop without a sensitivity loop. It is important to provide a time step event or delay within a forever loop or it will cause a simulation to hang. The following is the syntax for a forever loop in Verilog.

```
forever
  begin
   statement_1
   statement_2
        :
   statement_n
  end
```

Consider the following example of a forever loop that generates a clock signal (CLK) with a period of 10 time units. In this example, the forever loop is embedded within an initial block. This allows the initial value of CLK to be set to zero upon the beginning of the simulation. Once the forever loop is entered, it will execute indefinitely. Notice that since there is only one statement after the forever keyword, a statement group (i.e., begin/end) is not needed.

Example:

```
initial
  begin
    CLK = 0;

    forever
      #10 CLK = ~CLK;

  end
```

5.2.5 while Loops

A **while** loop provides a looping structure with a Boolean condition that controls its execution. The loop will only execute as long as the Boolean condition is evaluated true. The following is the syntax for a Verilog while loop.

```
while (<boolean_condition>)
  begin
    statement_1
    statement_2
        :
   statement_n
 end
```

Let us implement the previous example of a loop that generates a clock signal (CLK) with a period of 10 time units as long as EN = 1. The TRUE Boolean condition for the while loop is EN = 1. When EN = 0, the while loop will be skipped. When the loop becomes inactive, CLK will hold its last assigned value.

Example:

```
initial
  begin
    CLK = 0;

    while (EN == 1)
       #10 CLK = ~CLK;

  end
```

5.2.6 repeat Loops

A **repeat** loop provides a looping structure that will execute a fixed number of times. The following is the syntax for a Verilog repeat loop.

```
repeat (<number_of_loops>)
  begin
    statement_1
    statement_2
        :
    statement_n
  end
```

Let us implement the previous example of a loop that generates a clock signal (CLK) with a period of 10 time units, except this time we'll use a repeat loop to only produce ten clock transitions, or five full periods of CLK.

Example:

```
initial
  begin
     CLK = 0;
     repeat (10)
       #10 CLK = ~CLK;
  end
```

5.2.7 for loops

A **for** loop provides the ability to create a loop that can automatically update an internal variable. A *loop variable* within a for loop is altered each time through the loop according to a *step assignment*. The starting value of the loop variable is provided using an *initial assignment*. The loop will execute as long as a Boolean condition associated with the loop variable is TRUE. The following is the syntax for a Verilog for loop:

```
for (<initial_assignment>; <boolean_condition>; <step_assignment>)
  begin
    statement_1
    statement_2
          :
    statement_n
  end
```

The following is an example of creating a simple counter using the loop variable. The loop variable *i* was declared as an integer prior to this block. The signal Count is also of type integer. The loop variable will start at 0 and increment by 1 each time through the loop. The loop will execute as long as i < 15, or 16 times total. For loops allow the loop variable to be used in signal assignments within the block.

Example:

```
initial
  begin
    for (i=0; i<15; i=i+1)
      #10 Count = i;
  end
```

5.2.8 disable

Verilog provides the ability to stop a loop using the keyword **disable**. The disable function only works on named statement groups. The disable function is typically used after a certain fixed amount of time or within a conditional construct such as an if-else or case statement that is triggered by a control signal. Consider the following forever loop example that will generate a clock signal (CLK), but only when an enable (EN) is asserted. When EN = 0, the loop will disable and the simulation will end.

Example:

```
initial
  begin
    CLK = 0;
    forever
       begin: loop_ex
         if (EN == 1)
           #10 CLK = ~CLK;
         else
           disable loop_ex; // The group name to be disabled comes after the keyword
       end
  end
```

CONCEPT CHECK

CC5.2 When using an if-else statement to model a combinational logic circuit, is using the *else* clause the same as using *don't cares* when minimizing a logic expression with a K-map?

A) Yes. The else clause allows the synthesizer to assign whatever output values are necessary in order to create the most minimal circuit.

B) No. The else clause explicitly states the output values for all input codes not listed in the if portion of the statement. This is the same as filling in the truth table with specific values for all input codes covered by the else clause and the synthesizer will create the logic expression accordingly.

5.3 System Tasks

A system task in Verilog is one that is used to insert additional functionality into a model that is not associated with real circuitry. There are three main groups of system tasks in Verilog: (1) text output; (2) file input/output; and (3) simulation control. All system tasks begin with a $ and are only used during simulation. These tasks are ignored by synthesizers, so they can be included in real circuit models. All system tasks must reside within procedural blocks.

5.3.1 Text Output

Text output system tasks are used to print strings and variable values to the console or transcript of a simulation tool. The syntax follows ANSI C where double quotes (**“”**) are used to denote the text string to be printed. Standard text can be entered within the string in addition to variables. Variable can be printed in two ways. The first is to simply list the variable in the system task function outside of the double quotes. In this usage, the default format to be printed will be decimal unless a task is used with a different default format. The second way to print a variable is within a text string. In this usage, a unique code is inserted into the string indicating the format of how to print the value. After the string, a comma separated list of the variable name(s) is listed that corresponds positionally to the codes within the string. The following are the most commonly used text output system tasks.

Task	Description
$display()	Print text string when statement is encountered *and* append a newline.
$displayb()	Same as $display, but default format of any arguments is binary.
$displayo()	Same as $display, but default format of any arguments is octal.
$displayh()	Same as $display, but default format of any arguments is hexadecimal.
$write()	Same as $display, but the string is printed *without* a newline.
$writeb()	Same as $write, but default format of any arguments is binary.
$writeo()	Same as $write, but default format of any arguments is octal.
$writeh()	Same as $write, but default format of any arguments is hexadecimal.
$strobe()	Same as $display, but printing occurs *after* all simulation events are executed.
$strobeb()	Same as $strobe, but default format of any arguments is binary.
$strobeo()	Same as $strobe, but default format of any arguments is octal.
$strobeh()	Same as $strobe, but default format of any arguments is hexadecimal.
$monitor()	Same as $display, but printing occurs when the value of an argument *changes*.
$monitorb()	Same as $monitor, but default format of any arguments is binary.
$monitoro()	Same as $monitor, but default format of any arguments is octal.
$monitorh()	Same as $monitor, but default format of any arguments is hexadecimal.
$monitoron	Begin tracking argument changes in subsequent $monitor tasks.
$monitoroff	Stop tracking argument changes in subsequent $monitor tasks.

The following is a list of the most common text formatting codes for printing variables within a string.

Code	Format
%b	Binary values
%o	Octal values
%d	Decimal values
%h	Hexadecimal values
%f	Real values using decimal form
%e	Real values using exponential form

Code	Format
%t	Time values
%s	Character strings
%m	Hierarchical name of scope (no argument required when printing)
%l	Configuration library binding (no argument required when printing)

The format letters in these codes are not case sensitive (i.e., %d and %D are equivalent). Each of these formatting codes can also contain information about truncation of leading and trailing digits. Rounding will take place when numbers are truncated. The formatting syntax is as follows:

```
%<number_of_leading_digits>.<number_of_trailing_digits><format_code_letter>
```

There are also a set of string formatting and character escapes that are supported for use with the text output system tasks.

Code	Description
\n	Print a new line.
\t	Print a tab.
\”	Print a quote (“).
\cr	Print a backslash (\).
%%	Print a percent sign (%).

The following is a set of examples using common text output system tasks. For these examples, assume two variables have been declared and initialized as follows: A = 3 (integer) and B = 45.6789 (real). Recall that Verilog uses 32-bit codes to represent type integer and real.

Example:

```
$display("Hello World");     // Will print: Hello World
$display("A = %b", A);       // This will print: A = 00000000000000000000000000000011
$display("A = %o", A);       // This will print: A = 00000000003
$display("A = %d", A);       // This will print: A = 3
$display("A = %h", A);       // This will print: A = 00000003
$display("A = %4.0b", A);    // This will print: A = 0011

$display("B = %f", B);       // This will print: B = 45.678900
$display("B = %2.0f", B);    // This will print: B = 46
$display("B = %2.1f", B);    // This will print: B = 45.7
$display("B = %2.2f", B);    // This will print: B = 45.68

$display("B = %e", B);       // This will print: B = 4.567890e+001
$display("B = %1.0e", B);    // This will print: B = 5e+001
$display("B = %1.1e", B);    // This will print: B = 4.6e+001
$display("B = %2.2e", B);    // This will print: B = 4.57e+001

$write("A is ", A, "\n");    // This will print: A is 3
$writeb("A is ", A, "\n");   // This will print: A is 00000000000000000000000000000011
$writeo("A is ", A, "\n");   // Will print: A is 00000000003
$writeh("A is ", A, "\n");   // Will print: A is 00000003
```

5.3.2 File Input/Output

File I/O system tasks allow a Verilog module to create and/or access data files in the same way files are handled in ANSI C. This is useful when the results of a simulation are large and need to be stored in a file as opposed to viewing in a waveform or transcript window. This is also useful when complex stimulus

vectors are to be read from an external file and driven into a device under test. Verilog supports the following file I/O system task functions:

Task	Description
$fopen()	Opens a file and returns a unique file descriptor.
$fclose()	Closes the file associated with the descriptor.
$fdisplay()	Same as $display but statements are directed to the file descriptor.
$fwrite()	Same as $write but statements are directed to the file descriptor.
$fstrobe()	Same as $strobe but statements are directed to the file descriptor.
$fmonitor()	Same as $monitor but statements are directed to the file descriptor.
$readmemb()	Read binary data from file and insert into previously defined memory array.
$readmemh()	Read hexadecimal data from file and insert into previously defined memory array.

The **$fopen()** function will either create and open, or open an existing file. Each file that is opened is given a unique integer called a *file descriptor* that is used to identify the file in other I/O functions. The integer must be declared prior to the first use of $fopen. A file name argument is required and provided within double quotes. By default, the file is opened for writing. If the file name does not exist, it will be created. If the file name does exist, it will be overwritten. An optional *file_type* can be provided that gives specific action for the file opening including opening an existing file and appending to a file. The following are the supported codes for $fopen().

$fopen types	Description
“r” or **“rb”**	Open file for reading.
“w” or **“wb”**	Create for writing.
“a” or **“ab”**	Open for writing and append to the end of file.
“r+” or **“r+b”** or **“rb+”**	Open for update, reading or writing file.
“w+” or **“w+b”** or **“wb+”**	Create for update.
“a+” or **“a+b”** or **“ab+”**	Open or create for update, append to the end of file.

Once a fie is open, data can be written to it using the **$fdisplay()**, **$fwrite()**, **$fstrobe()**, and **$fmonitor()** tasks. These functions require two arguments. The first argument is the file descriptor and the second is the information to be written. The information follows the same syntax as the I/O system tasks. The following example shows how to create a file and write data to it. This example will create a new file called “Data_out.txt” and write two lines of text to it with the values of variables A and B.

Example:

```
integer A = 3;
real   B = 45.6789;
integer FILE_1;

initial
  begin
    FILE_1 = $fopen("Data_out.txt", "w");
    $fdisplay(FILE_1, "A is %d", A);
    $fdisplay(FILE_1, "B is %f", B);
    $fclose(FILE_1);
  end
```

When reading data from a file, the functions **$readmemb()** and **$readmemh()** can be used. These tasks require that a storage array be declared that the contents of the file can be read into. These tasks

have two arguments, the first being the name of the file and the second being the name of the storage array to store the file contents into. The following example shows how to read the contents of a file into a storage array called "memory." Assume the file contains eight lines, each containing a 3-bit vector. The vectors start at 000 and increment to 111 and each symbol will be interpreted as binary using the $readmemb() task. The storage array "memory" is declared to be an 8 × 3 array of type reg. The $readmemb() task will insert each line of the file into each 3-bit vector location within "memory." To illustrate how the data are stored, this example also contains a second procedural block that will print the contents of the storage element to the transcript.

Example:

```
reg[2:0] memory[7:0];

initial
  begin: Read_Block
  $readmemb("Data_in.txt", memory);
  end
initial
  begin: Print_Block
    $display("printing memory %b", memory[0]);    // This will print "000"
    $display("printing memory %b", memory[1]);    // This will print "001"
    $display("printing memory %b", memory[2]);    // This will print "010"
    $display("printing memory %b", memory[3]);    // This will print "011"
   $display("printing memory %b", memory[4]);    // This will print "100"
   $display("printing memory %b", memory[5]);    // This will print "101"
   $display("printing memory %b", memory[6]);    // This will print "110"
   $display("printing memory %b", memory[7]);    // This will print "111"
  end
```

5.3.3 Simulation Control and Monitoring

Verilog also provides a set of simulation control and monitoring tasks. The following are the most commonly used tasks in this group.

Task	Description	
$finish()	Finishes simulation and exits.	
$stop()	Halts the simulation and enters an interactive debug mode.	
$time()	Returns the current simulation time as a 64-bit vector.	
$stime()	Returns the current simulation time as a 32-bit integer.	
$realtime()	Returns the current simulation time as a 32-bit real number.	
$timeformat()	Controls the format used by the %t code in print statements.	
	The arguments are: (<unit>, <precision>, <suffix>, <min_field_width>)	
	where:	
	<unit>	0 = 1 s
		−1 = 100 ms
		−2 = 10 ms
		−3 = 1 ms
		−4 = 100 μs
		−5 = 10 μs
		−6 = 1 μs
		−7 = 100 ns
		−8 = 10 ns
		−9 = 1 ns
		−10 = 100 ps

Task	Description	
		−11 = 10 ps
		−12 = 1 ps
		−13 = 100 fs
		−14 = 10 fs
		−15 = 1 fs
	<precision> = The number of decimal points to display.	
	<suffix> = A string to be appended to time to indicate units.	
	<min_field_width> = The minimum number of characters to display.	

The following shows an example of how these tasks can be used.

Example:

```
initial
  begin

    $timeformat (-9, 2, "ns", 10);
    $display("Stimulus starting at time: %t", $time);

    #10  A_TB=0; B_TB=0; C_TB=0;
    #10  A_TB=0; B_TB=0; C_TB=1;
    #10  A_TB=0; B_TB=1; C_TB=0;
    #10  A_TB=0; B_TB=1; C_TB=1;
    #10  A_TB=1; B_TB=0; C_TB=0;
    #10  A_TB=1; B_TB=0; C_TB=1;
    #10  A_TB=1; B_TB=1; C_TB=0;
    #10  A_TB=1; B_TB=1; C_TB=1;

    $display("Simulation stopping at time: %t", $time);
  end
```

This example will result in the following statements printed to the simulator transcript:

```
Stimulus starting at time:  0.00ns
Simulation stopping at time: 80.00ns
```

Concept Check

CC5.3 How can Verilog system tasks be included in synthesizable circuit models when they provide inherently unsynthesizable functionality?

A) They cannot. System tasks can only be used in test benches.

B) The "$" symbol tells the CAD tool that the task can be ignored during synthesis.

C) The designer must only use system tasks that model sequential logic.

D) The designer must only use system tasks that model combinational logic.

Summary

- To model sequential logic, an HDL needs to be able to trigger signal assignments based on an event. This is accomplished in Verilog using *procedural assignment*.
- There are two types of procedural blocks in Verilog, *initial* and *always*. An initial block executes one time. An always block runs continually.
- A *sensitivity* list is a way to control when a Verilog procedural block is triggered. A sensitivity list contains a list of signals. If any of the signals in the sensitivity list transitions, it will cause the block to trigger. If a sensitivity list is omitted, the block will trigger immediately. Sensitivity lists are most commonly used with always blocks.
- Sensitivity lists and always blocks are used to model synthesizable logic. Initial blocks are typically only used in test benches. Always blocks are also used in test benches.
- There are two types of signal assignments that can be used within a procedural block, blocking and nonblocking.
- A blocking assignment is denoted with the = symbol. All blocking assignments are made immediately within the procedural block. Blocking assignments are used to model combinational logic. Combinational logic models list all input to the circuit in the sensitivity list.
- A nonblocking assignment is denoted with the <= symbol. All nonblocking assignments are made when the procedural block ends and are evaluated in the order they appeared in the block. Blocking assignments are used to model sequential logic. Sequential logic models list only the clock and reset in the sensitivity list.
- Variables can be defined within a procedural block as long as the block is named.
- Procedural blocks allow more advanced modeling constructs in Verilog. These include *if-else statements*, *case statements*, *and loops*.
- Verilog provides numerous looping constructs including *forever*, *while*, *repeat*, and *for*. Loops can be terminated using the *disable* keyword.
- *System Tasks* provide additional functionality to Verilog models. Tasks begin with the $ symbol and are omitted from synthesis. System tasks can be included in synthesizable logic models.
- There are three groups of system tasks: text output, file input/output, and simulation control and monitoring.
- System tasks that perform printing functions can output strings in addition to variable values. Verilog provides a mechanism to print the variable values in a variety of format.

Exercise Problems

Section 5.1: Procedural Assignment

5.1.1 When using a sensitivity list with a procedural block, what will cause the block to *trigger*?

5.1.2 When a sensitivity list is not used with a procedural block, when will the block trigger?

5.1.3 When are statements executed when using blocking assignments?

5.1.4 When are statements executed when using nonblocking assignments?

5.1.5 When is it possible to exclude statement groups from a procedural block?

5.1.6 What is the difference between a begin/end and fork/join group when each contains multiple statements?

5.1.7 What is the difference between a begin/end and fork/join group when each contains only a single statement?

5.1.8 What type of procedural assignment is used when modeling combinational logic?

5.1.9 What type of procedural assignment is used when modeling sequential logic?

5.1.10 What signals should be listed in the sensitivity list when modeling combinational logic?

5.1.11 What signals should be listed in the sensitivity list when modeling sequential logic?

Section 5.2: Conditional Programming Constructs

5.2.1 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 5.1. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output. Hint: Notice that there are far more input codes producing F = 0 than producing F = 1. Can you use this to your advantage to make your if-else statement simpler?

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

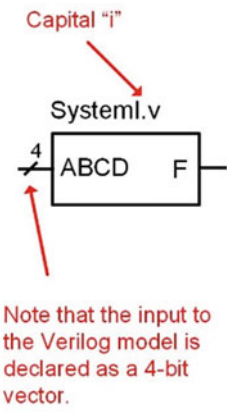


Fig. 5.1
System I functionality

5.2.2 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 5.1. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

5.2.3 Design a Verilog model to implement the behavior described by the 4-input minterm list in Fig. 5.2. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

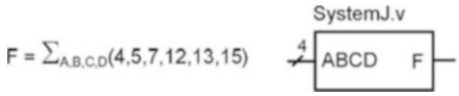


Fig. 5.2
System J functionality

5.2.4 Design a Verilog model to implement the behavior described by the 4-input minterm list in Fig. 5.2. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

5.2.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list in Fig. 5.3. Use procedural assignment and an if-then statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

Fig. 5.3
System K functionality

5.2.6 Design a Verilog model to implement the behavior described by the 4-input maxterm list in Fig. 5.3. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

5.2.7 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 5.4. Use procedural assignment and an if-else statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output. Hint: Notice that there are far more input codes producing F = 1 than producing F = 0. Can you use this to your advantage to make your if-else statement simpler?

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

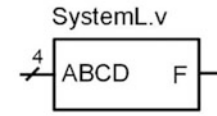


Fig. 5.4
System L functionality

5.2.8 Design a Verilog model to implement the behavior described by the 4-input truth table in Fig. 5.4. Use procedural assignment and a case statement. Declare the module to match the block diagram provided. Use the type wire for the inputs and type reg for the output.

5.2.9 Figure 5.5 shows the topology of a *4-bit shift register* when implemented structurally using

D-Flip-Flops. Design a Verilog model to describe this functionality using a single procedural block and nonblocking assignments instead of instantiating D-Flip-Flops. The figure also provides the block diagram for the module port definition. Use the type wire for the inputs and type reg for the outputs.

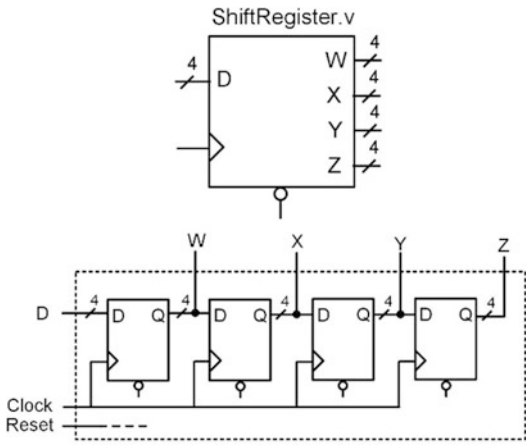


Fig. 5.5
4-bit shift register block diagram

5.2.10 Design a Verilog model for a counter using a for loop with an output type of integer. Figure 5.6 shows the block diagram for the module definition. The counter should increment from 0 to 31 and then start over. Use delay in your loop to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate your counter value.

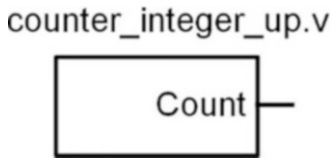


Fig. 5.6
Integer counter block diagram

5.2.11 Design a Verilog model for a counter using a for loop with an output type of reg[4:0]. Figure 5.7 shows the block diagram for the module definition. The counter should increment from 00000_2 to 11111_2 and then start over. Use delay in your loop to update the counter value every 10 ns. Consider using the loop variable of the for loop to generate an integer version of your count value, and then assign it to the output variable of type reg[4:0].

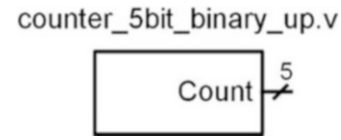


Fig. 5.7
5-bit binary counter block diagram

Section 5.3: System Tasks

5.3.1 Are system tasks synthesizable? Why or why not?

5.3.2 What is the difference between the tasks $display() and $write()?

5.3.3 What is the difference between the tasks $display() and $monitor()?

5.3.4 What is the data type returned by the task $fopen()?

Chapter 6: Test Benches

One of the essential components of the modern digital design flow is verifying functionality through simulation. This functional verification is accomplished using a *test bench*. A test bench is a Verilog model that instantiates the system to be tested as a subsystem, generates the input patterns to drive into the subsystem, and observes the outputs. Test benches are only used for simulation, so they can use abstract modeling techniques that are unsynthesizable to generate the stimulus patterns. Verilog conditional programming constructions and system tasks can also be used to report on the status of a test and also automatically check that the outputs are correct. This chapter provides the details of Verilog's built-in capabilities that allow test benches to be created and some examples of automated stimulus generation.

Learning Outcomes—After completing this chapter, you will be able to:

6.1 Design a Verilog test bench that manually creates each stimulus pattern using a series of signal assignments within a procedural block
6.2 Design a Verilog test bench that uses for loops to automatically generate an exhaustive set of stimulus patterns
6.3 Design a Verilog test bench that automatically checks the outputs of the system being tested using report and assert statements
6.4 Design a Verilog test bench that uses external I/O as part of the testing procedures including reading stimulus patterns from, and writing the results to, external files

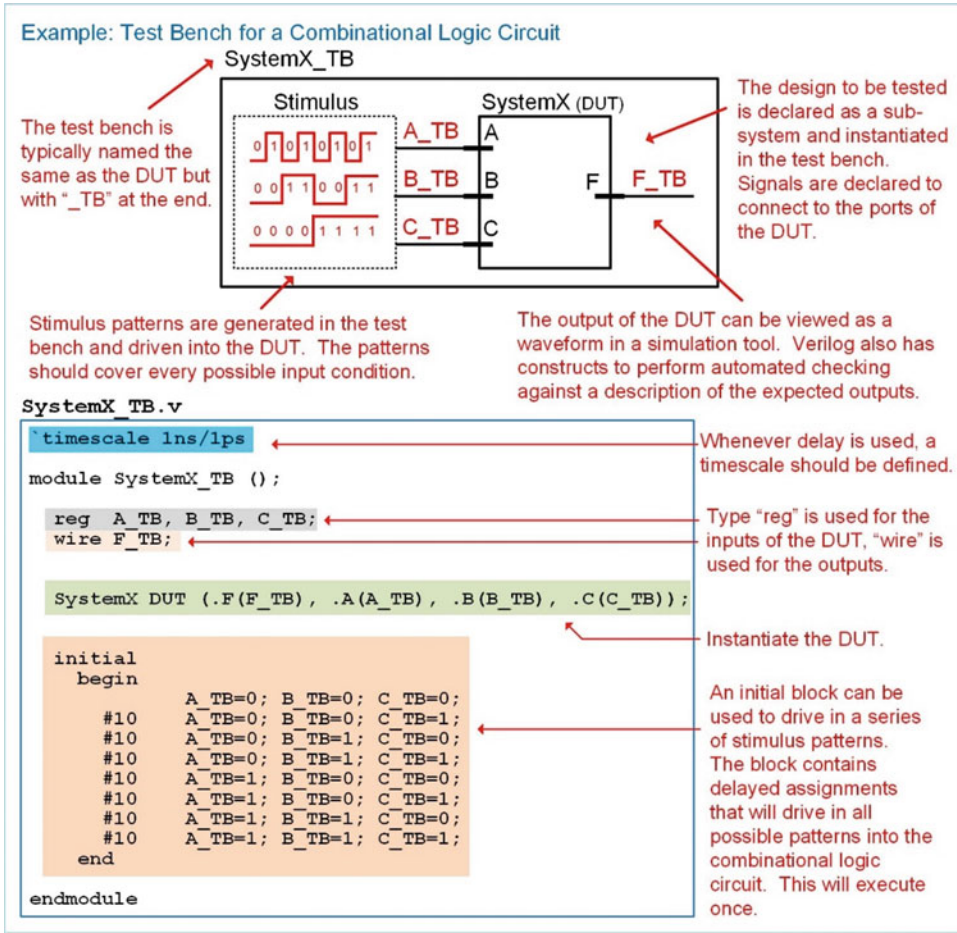
6.1 Test Bench Overview

A test bench is a file in Verilog that has no inputs or outputs. The test bench instantiates the system to be tested as a lower-level module. The system being tested is often called a *device under test (DUT)* or *unit under test (UUT).*

6.1.1 Generating Manual Stimulus

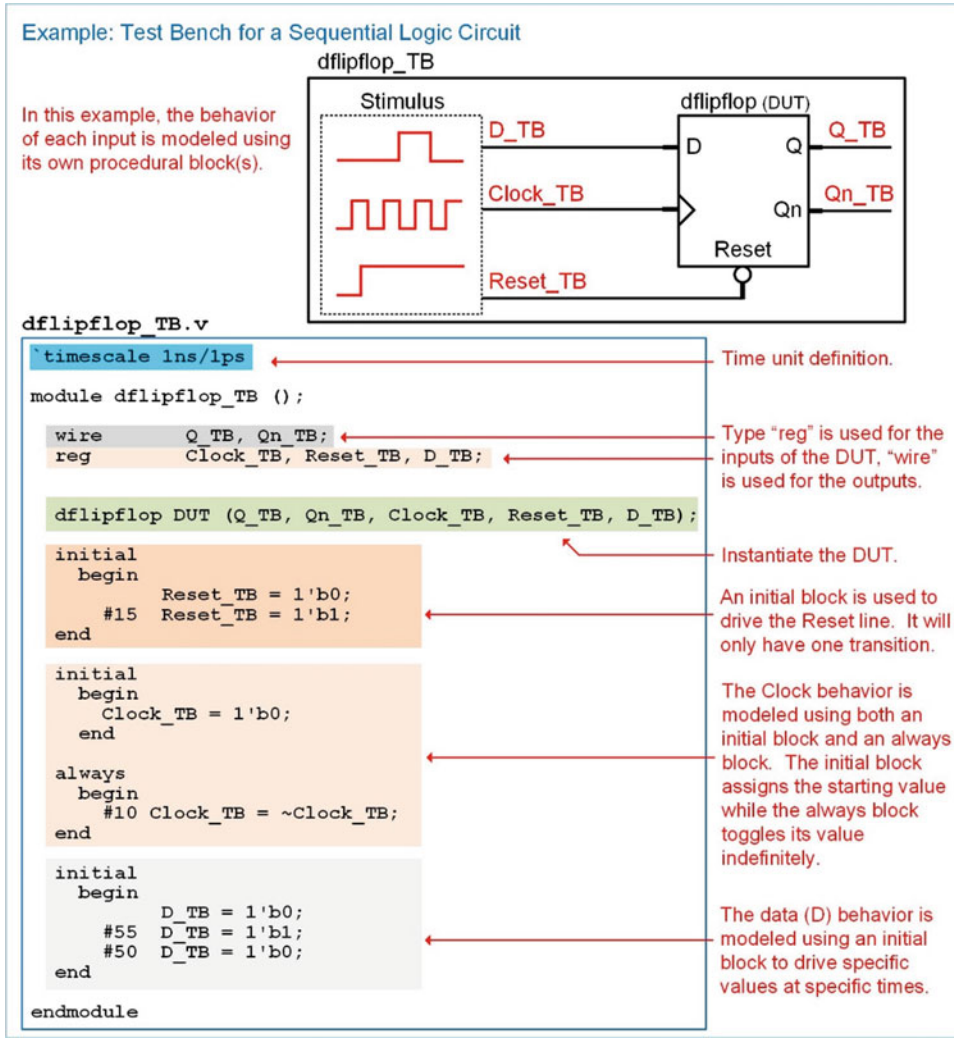
When creating stimulus for combinational logic circuits, it is common to use a procedural block to generate all possible input patterns to drive the DUT and especially any transitions that may cause timing errors. Example 6.1 shows how to create a simple test bench to verify the operation of a DUT called SystemX. The test bench does not have any inputs or outputs, thus there are no ports declared in the module. SystemX is then instantiated (DUT) in the test bench. Internal signals of type *reg* are declared to connect to the DUT inputs (A_TB, B_TB, C_TB) and an internal signal of type *wire* is declared to connect to the DUT output (F_TB). A procedural block is then used to generate the inputs of SystemX. Within the procedural block, delayed assignments are used to control the timing of the input patterns. In this example, each possible input code is generated within an initial block. The output (F_TB) is observed using a simulation tool in either the form of a waveform or a table listing.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_6



Example 6.1
Test bench for a combinational logic circuit with manual stimulus generation

Multiple procedural blocks can be used within a Verilog test bench to provide parallel stimulus generation. Using both initial and always blocks allow the test bench to drive both repetitive and aperiodic signals. Initial and always blocks can also be used to drive the same signal in order to provide a starting value and a repetitive pattern. Example 6.2 shows a test bench for a rising edge triggered D-flip-flop with an asynchronous, active LOW reset in which multiple procedural blocks are used to generate the stimulus patterns for the DUT.

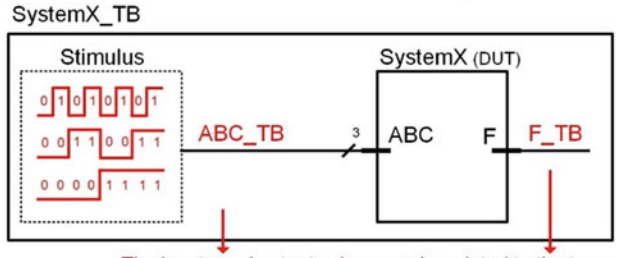


Example 6.2
Test bench for a sequential logic circuit

6.1.2 Printing Results to the Simulator Transcript

In the past test bench examples, the input and output values are observed using either the waveform or listing tool within the simulator tool. It is also useful to print the values of the simulation to a transcript window to track the simulation as each statement is processed. Messages can be printed that show the status of the simulation in addition to the inputs and outputs of the DUT using the text output system tasks. Example 6.3 shows a test bench that prints the inputs and output to the transcript of the simulation tool. Note that the test bench must wait some amount of delay before evaluating the output, even if the DUT does not contain any delay.

Example: Printing Test Bench Results to the Transcript

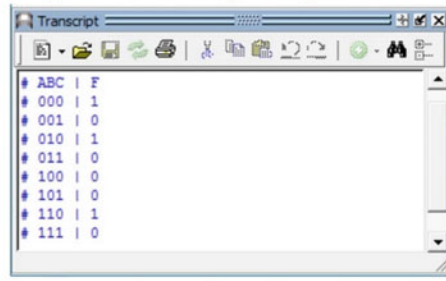


SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;


 SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin
                              $display("ABC | F");
        ABC_TB=3'b000;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b001;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b010;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b011;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b100;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b101;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b110;  #1    $display("%b | %b", ABC_TB, F_TB);
     #9 ABC_TB=3'b111;  #1    $display("%b | %b", ABC_TB, F_TB);
    end

endmodule
```

Even if the DUT model does not contain delay, the test bench needs to delay before evaluating the output.

The above test bench will print out the following message to the transcript of the simulator tool.



Example 6.3
Printing test bench results to the transcript

CONCEPT CHECK

CC6.1 How can the output of a DUT be verified when it is connected to a signal that does not go anywhere?

A) It cannot. The output must be routed to an output port on the test bench.

B) The values of any dangling signal are automatically written to a text file.

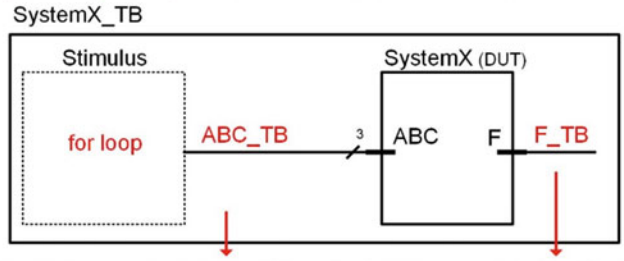
C) It is viewed in the logic simulator as either a waveform or text listing.

D) It cannot. A signal that does not go anywhere will cause an error during simulation.

6.2 Using Loops to Generate Stimulus

When creating stimulus that follow regular patterns such as counting, loops can be an effective way to produce the input vectors. A *for* loop is especially useful for generating exhaustive stimulus patterns for combinational logic circuits. An integer loop variable can increment within the for loop and then be assigned to the DUT inputs as type *reg*. Recall that in Verilog, when an integer is assigned to a variable of type reg, it is truncated to match the size of the reg. This allows a binary count to be created for an input stimulus pattern by using an integer loop variable that increments within a for loop. Example 6.4 shows how the stimulus for a combinational logic circuit can be produced with a for loop.

Example: Using a Loop to Generate Stimulus in a Test Bench



The inputs and output values will be printed to the transcript using $display().

SystemX_TB.v

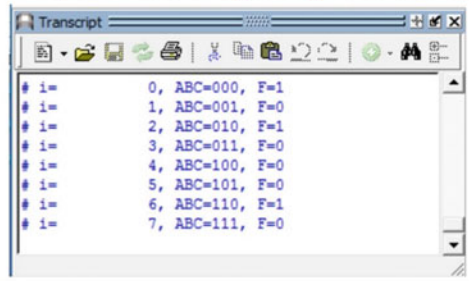
```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;
  integer    i;

  SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin

      for (i=0; i<8; i=i+1)
        begin
          ABC_TB = i;
          #10 $display("i=%d, ABC=%b, F=%b", i, ABC_TB, F_TB);
        end

    end

endmodulee
```

When using a for loop, the loop variable must be declared.

Each time through the loop, *i* will increment by one. It will count from 0 to 7.

The bottom 3-bits of the integer *i* are used in the assignment to ABC_TB.

The above test bench will print out the following message to the transcript of the simulator tool.



Example 6.4
Using a loop to generate stimulus in a test bench

Concept Check

CC6.2 If you used two nested for loops to generate an exhaustive set of patterns for the inputs of an 8-bit adder, how many patterns would be generated? There is no carry-in bit.

A) 16 B) 256 C) 512 D) 65,536

6.3 Automatic Result Checking

Test benches can also perform automated checking of the results using the conditional programming constructs described earlier in this book. Example 6.5 shows an example of a test bench that uses *if-else* statements to check the output of the DUT and print a PASS/FAIL message to the transcript.

Example: Test Bench with Automatic Output Checking

SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;

 SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin
                              $display("ABC | F");

          ABC_TB=3'b000;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b001;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b010;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b011;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b100;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b101;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b110;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b1) $display(" PASS"); else $display(" FAIL");

       #9 ABC_TB=3'b111;  #1  $write("%b | %b", ABC_TB, F_TB);
          if(F_TB == 1'b0) $display(" PASS"); else $display(" FAIL");

    end

endmodule
```

if/else statements check the value of F_TB after each input.

Note that a $write() task is used so that the PASS/FAIL messages are printed on the same line as the I/O values.

This message will be printed to the transcript when the DUT has the correct outputs.

```
# ABC | F
# 000 | 1 PASS
# 001 | 0 PASS
# 010 | 1 PASS
# 011 | 0 PASS
# 100 | 0 PASS
# 101 | 0 PASS
# 110 | 1 PASS
# 111 | 0 PASS
```

The DUT was altered to have the wrong output for input codes 010 and 011.

```
# ABC | F
# 000 | 1 PASS
# 001 | 0 PASS
# 010 | 0 FAIL
# 011 | 1 FAIL
# 100 | 0 PASS
# 101 | 0 PASS
# 110 | 1 PASS
# 111 | 0 PASS
```

Example 6.5
Test bench with automatic output checking

Concept Check

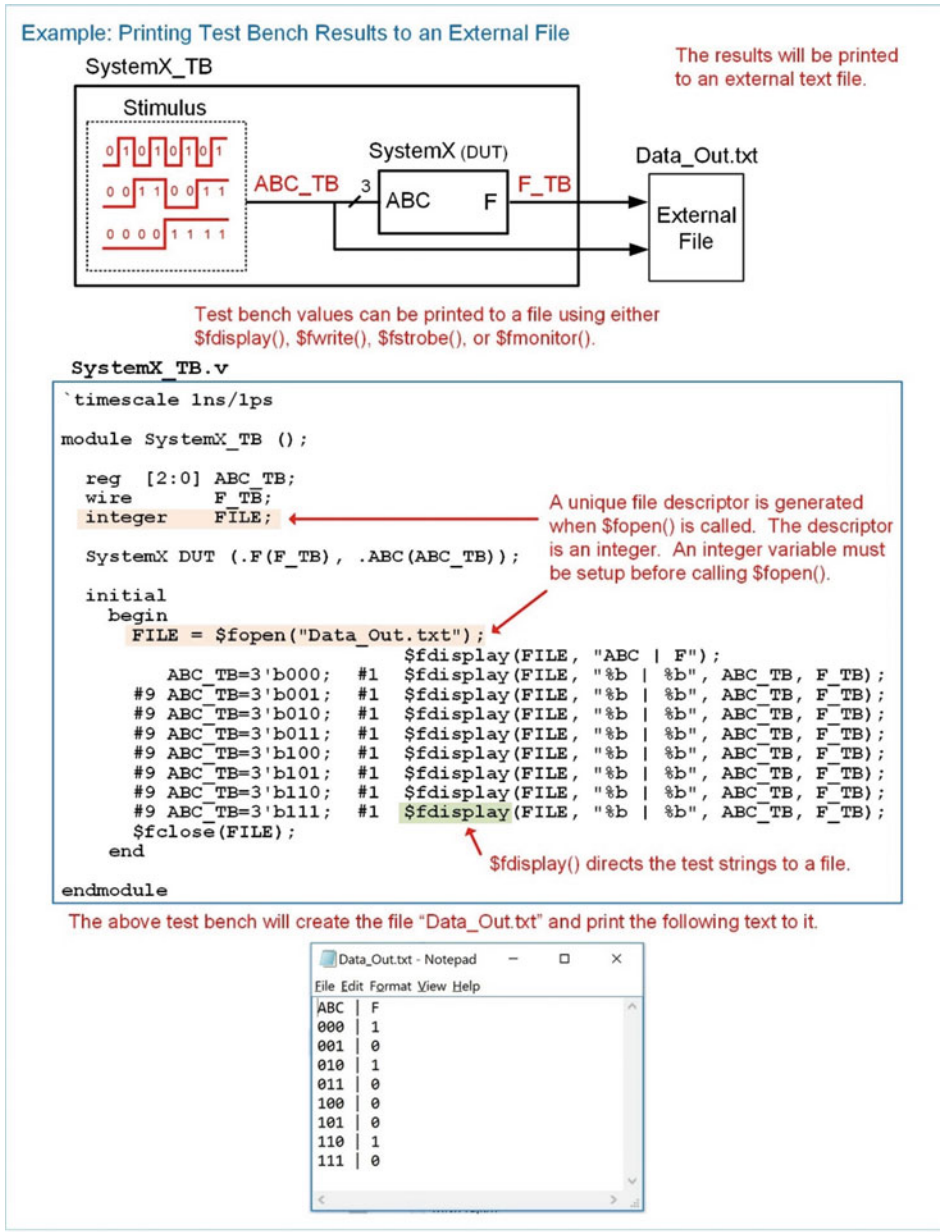
CC6.3 Will the test bench approach of checking the results and then printing PASS/FAIL to the transcript window stop the simulation?

A) Yes. As soon as “FAIL” is printed, the simulation will halt.

B) No. The printing of PASS/FAIL is just simple text and does not influence the simulation.

6.4 Using External Files in Test Benches

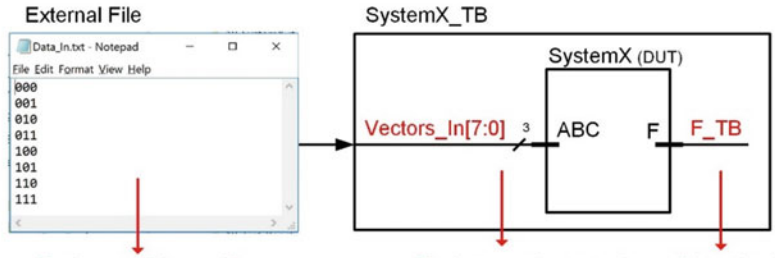
There are often cases where the results of a test bench need to be written to an external file, either because they are too verbose for visual inspection or because there needs to be a stored record of the system’s validation. Verilog allows writing to external files via the file I/O system tasks (i.e., $fdisplay(), $fwrite(), $fstrong(), and $fmonitor()). Example 6.6 shows a test bench in which the input vectors and the output of the DUT are written to an external file using the $fdisplay() system task.



Example 6.6
Printing test bench results to an external file

It is often the case that the input vectors are either too large to enter manually or were created by a separate program. In either case, a useful technique in test benches is to read input vectors from an external file. Example 6.7 shows an example where the input stimulus vectors for a DUT are read from an external file using the $readmemb() system task.

Example: Reading Test Bench Stimulus Vectors from an External File



SystemX_TB.v

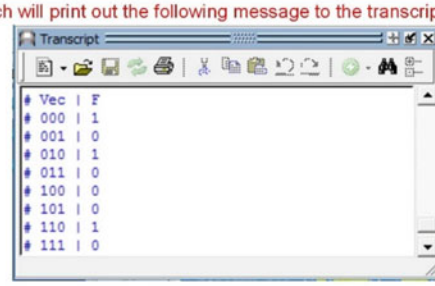
```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  [2:0] ABC_TB;
  wire       F_TB;
  reg [2:0]  Vectors_In[7:0];

  SystemX DUT (.F(F_TB), .ABC(ABC_TB));

  initial
    begin

    $readmemb("Data_In.txt", Vectors_In);

                                $display("Vec | F");
       ABC_TB=Vectors_In[0];  #1 $display("%b | %b", Vectors_In[0], F_TB);
    #9 ABC_TB=Vectors_In[1];  #1 $display("%b | %b", Vectors_In[1], F_TB);
    #9 ABC_TB=Vectors_In[2];  #1 $display("%b | %b", Vectors_In[2], F_TB);
    #9 ABC_TB=Vectors_In[3];  #1 $display("%b | %b", Vectors_In[3], F_TB);
    #9 ABC_TB=Vectors_In[4];  #1 $display("%b | %b", Vectors_In[4], F_TB);
    #9 ABC_TB=Vectors_In[5];  #1 $display("%b | %b", Vectors_In[5], F_TB);
    #9 ABC_TB=Vectors_In[6];  #1 $display("%b | %b", Vectors_In[6], F_TB);
    #9 ABC_TB=Vectors_In[7];  #1 $display("%b | %b", Vectors_In[7], F_TB);

    end

endmodule
```

An 8x3 array called "Vectors_In" is declared to hold the values read from the external file.

$readmemb() treats each symbol in the input file as a binary value. It populates the entire Vectors_In array when called.

Entries in the Vectors_In array can be assigned to the inputs of the DUT.

The above test bench will print out the following message to the transcript of the simulator tool.



Example 6.7
Reading test bench stimulus vectors from an external file

Concept Check

CC6.4 What is an advantage of using external files as the input/output in test benches compared to the built-in stimulus generation and reporting functionality within a Verilog module?

A) External stimulus files allow more complex input stimulus vectors to be used.

B) External output files allow more sophisticated postprocessing of the results.

C) External files allow much larger datasets to be used and analyzed.

D) All of the above.

Summary

- A *test bench* is a way to simulate a device under test (DUT) by instantiating it as a subsystem, driving in stimulus, and observing the outputs.
- Test benches do not have inputs or outputs and are unsynthesizable.
- Test benches for combinational logic typically exercise the DUT under an exhaustive set of stimulus vectors. These include all possible logic inputs in addition to critical transitions that could cause timing errors.
- Text I/O system tasks provide a way to print the results of a test bench to the simulation tool transcript.
- File I/O system tasks provide a way to print the results of a test bench to an external file and also to read in stimulus vectors from an external file.
- Conditional programming constructs can be used within a test bench to perform automatic checking of the outputs of a DUT within a test bench.
- Loops can be used in test benches to automatically generate stimulus patterns. A for loop is a convenient technique to produce a counting pattern.
- Assignment from an integer to a reg in a for loop is allowed. The binary value of the integer is truncated to fit the size of the reg vector.

Exercise Problems

Section 6.1: Test Bench Overview

6.1.1 What is the purpose of a test bench?

6.1.2 Does a test bench have input and output ports?

6.1.3 Can a test bench be simulated?

6.1.4 Can a test bench be synthesized?

6.1.5 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

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

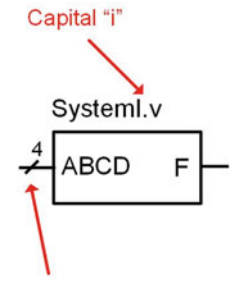


Fig. 6.1
System I functionality

6.1.6 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

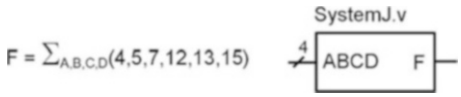


Fig. 6.2
System J functionality

6.1.7 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

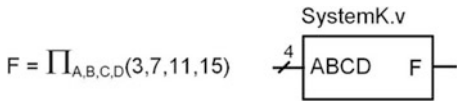


Fig. 6.3
System K functionality

6.1.8 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a procedural block and individual signal assignments for each pattern. Your test bench should change the input pattern every 10 ns.

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

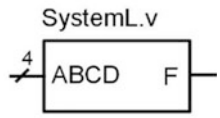


Fig. 6.4
System L functionality

Section 6.2: Generating Stimulus Vectors Using For Loops

6.2.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., “0000,” “0001,” “0010,” . . .). Your test bench should use a single for loop within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 10 ns.

6.2.5 Design a Verilog model for an 8-bit Ripple Carry Adder (RCA) using a structural design approach. This involves creating a half adder (half_adder.v), full adder (full_adder.v), and then finally a top-level adder (rca.v) by instantiating eight full adder subsystems. Model the ripple delay by inserting 1 ns of gate delay for the XOR, AND, and OR operators using a delayed signal assignment. The general topology and module definition for the design are shown in Example 4.8. Design a Verilog test bench to exhaustively verify this design under all input conditions. Your test bench should use two nested for loops within a procedural block to generate all of the stimulus patterns automatically. Your test bench should change the input pattern every 30 ns in order to give sufficient time for the signals to ripple through the adder.

Section 6.3: Automated Result Checking

6.3.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either "PASS" or "FAIL" depending on the output of the DUT.

6.3.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either "PASS" or "FAIL" depending on the output of the DUT.

6.3.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either "PASS" or "FAIL" depending on the output of the DUT.

6.3.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should drive in each input code for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should change the input pattern every 10 ns. Your test bench should include automatic result checking for each input pattern and then print either "PASS" or "FAIL" depending on the output of the DUT.

Section 6.4: Using External Files in Test Benches

6.4.1 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.1. Your test bench should read in the input patterns from an external file called "input.txt." This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called "output.txt."

6.4.2 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.2. Your test bench should read in the input patterns from an external file called "input.txt." This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called "output.txt."

6.4.3 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.3. Your test bench should read in the input patterns from an external file called "input.txt." This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called "output.txt."

6.4.4 Design a Verilog test bench to verify the functional operation of the system in Fig. 6.4. Your test bench should read in the input patterns from an external file called "input.txt." This file should contain an exhaustive list of input patterns for the vector ABCD in the order they would appear in a truth table (i.e., "0000," "0001," "0010," ...). Your test bench should read in a new input pattern every 10 ns. Your test bench should write the input pattern and the corresponding output of the DUT to an external file called "output.txt."

Chapter 7: Modeling Sequential Storage and Registers

In this chapter, we will look at modeling sequential storage devices. We begin by looking at modeling scalar storage devices such as D-latches and D-flip-flops and then move into multiple-bit storage models known as registers. We will then look at register transfer level modeling where data is passed between multiple registers on a common bus.

Learning Outcomes—After completing this chapter, you will be able to:

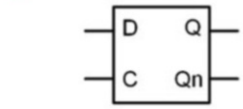
7.1 Design a Verilog model for a single-bit sequential logic storage device
7.2 Design a Verilog model for a register

7.1 Modeling Scalar Storage Devices

7.1.1 D-Latch

Let us begin with the model of a simple D-Latch. Since the outputs of this sequential storage device are not updated continuously, its behavior is modeled using a procedural assignment. Since we want to create a synthesizable model of sequential logic, nonblocking assignments are used. In the sensitivity list, we need to include the C input since it controls when the D-Latch is in track or store mode. We also need to include the D input in the sensitivity list because during the track mode, the output Q will be assigned the value of D so any change on D needs to trigger the procedural assignments. The use of an if-else statement is used to model the behavior during track mode (C = 1). Since the behavior is not explicitly stated for when C = 0, the outputs will hold their last value, which allows us to simply omit the else portion of the if statement to complete the model. Example 7.1 shows the behavioral model for a D-Latch.

Example: Behavioral Model of a D-Latch in Verilog



C	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	0	0	1	Track
1	1	1	0	Track

```
module dlatch (output reg  Q, Qn,
               input  wire C, D);

  always @ (C or D)
    if (C == 1'b1)
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

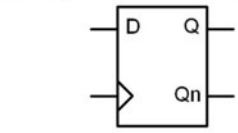
Example 7.1
Behavioral model of a D-latch in Verilog

7.1.2 D-Flip-Flop

The rising edge behavior of a D-Flip-Flop is modeled using a (posedge Clock) Boolean condition in the sensitivity list of a procedural block. Example 7.2 shows the behavioral model for a rising edge-triggered D-Flip-Flop with both Q and Qn outputs.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_7

Example: Behavioral Model of a D-Flip-Flop in Verilog



Clk	D	Q	Qn	
0	X	Last Q	Last Qn	Store
1	X	Last Q	Last Qn	Store
↓	0	0	1	Update
↓	1	1	0	Update

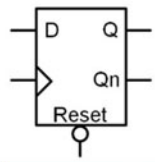
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, D);

  always @ (posedge Clock)
     begin
        Q  <= D;
        Qn <= ~D;
     end

endmodule
```

Example 7.2
Behavioral model of a D-flip-flop in Verilog

7.1.3 D-Flip-Flop with Asynchronous Reset

D-Flip-Flops typically have a reset line to initialize their outputs to known states (e.g., Q = 0, Qn = 1). Resets are asynchronous, meaning whenever they are asserted, assignments to the outputs take place immediately. If a reset was *synchronous*, the outputs would only update on the next rising edge of the clock. This behavior is undesirable because if there is a system failure, there is no guarantee that a clock edge will ever occur. Thus, the reset may never take place. Asynchronous resets are more desirable not only to put the D-Flip-Flops into a known state at startup, but also to recover from a system failure that may have impacted the clock signal. In order to model this asynchronous behavior, the reset signal is included in the sensitivity list. This allows both clock and the reset transitions to trigger the procedural block. The edge sensitivity of the reset can be specified using posedge (active HIGH) or negedge (active LOW). Within the block, an if-else statement is used to determine whether the reset has been asserted or a rising edge of the clock has occurred. The if-else statement first checks whether the reset input has been asserted since it has the highest priority. If it has, it makes the appropriate assignments to the outputs (Q = 0, Qn = 1). If the reset has not been asserted, the *else* clause is executed, which corresponds to a rising edge of clock (Q <= D, Qn <= ~D). No other assignments are listed in the block, thus the outputs are only updated on a transition of the reset or clock. At all other times the outputs remain at their current value, thus modeling the store behavior of the D-Flip-Flop. Example 7.3 shows the behavioral model for a rising edge-triggered D-Flip-Flop with an asynchronous, active LOW reset.

Example: Behavioral Model of a D-Flip-Flop with Asynchronous Reset in Verilog



$\bar{R}$	Clk	D	Q	Qn	
0	X	X	0	1	Reset
1	0	X	Last Q	Last Qn	Store
1	1	X	Last Q	Last Qn	Store
1	↓	0	0	1	Update
1	↓	1	1	0	Update

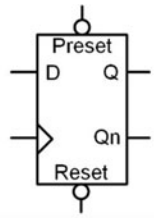
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, D);

  always @ (posedge Clock or negedge Reset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

Example 7.3
Behavioral model of a D-flip-flop with asynchronous reset in Verilog

7.1.4 D-Flip-Flop with Asynchronous Reset and Preset

A D-Flip-Flop with both an asynchronous reset and asynchronous preset is handled in a similar manner as the D-Flip-Flop in the prior section. The preset input is included in the sensitivity list in order to trigger the block whenever a transition occurs on either the clock, reset, or preset inputs. The edge sensitivity keywords are used to dictate whether the preset is active HIGH or LOW. Nested if-else statements are used to first check whether a reset has occurred; then whether a preset has occurred; and finally, whether a rising edge of the clock has occurred. Example 7.4 shows the model for a rising edge-triggered D-Flip-Flop with asynchronous, active LOW reset and preset.

Example: Behavioral Model of a D-Flip-Flop with Asynchronous Reset and Preset in Verilog



R	P	Clk	D	Q	Qn	
0	X	X	X	0	1	Reset
1	0	X	X	1	0	Preset
1	1	0	X	Last Q	Last Qn	Store
1	1	1	X	Last Q	Last Qn	Store
1	1	↓	0	0	1	Update
1	1	↓	1	1	0	Update

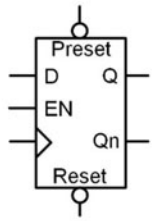
```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else
      begin
        Q  <= D;
        Qn <= ~D;
      end

endmodule
```

Example 7.4
Behavioral model of a D-flip-flop with asynchronous reset and preset in Verilog

7.1.5 D-Flip-Flop with Synchronous Enable

An enable input is also a common feature of modern D-Flip-Flops. Enable inputs are synchronous, meaning that when they are asserted, action is only taken on the rising edge of the clock. This means that the enable input is not included in the sensitivity list of the always block. Since enable is only considered when there is a rising edge of the clock, the logic for the enable is handled in a nested if-else statement that is included in the section that models the behavior for when a rising edge of clock is detected. Example 7.5 shows the model for a D-Flip-Flop with a synchronous enable (EN) input. When EN = 1, the D-Flip-Flop is enabled, and assignments are made to the outputs only on the rising edge of the clock. When EN = 0, the D-Flip-Flop is disabled and assignments to the outputs are not made. When disabled, the D-Flip-Flop effectively ignores rising edges on the clock and the outputs remain at their last values.

Example: Behavioral Model of a D-Flip-Flop with Synchronous Enable in Verilog



$\bar{R}$	$\bar{P}$	Clk	EN	D	Q	Qn	
0	X	X	X	X	0	1	Reset
1	0	X	X	X	1	0	Preset
1	1	0	X	X	Last Q	Last Qn	Store
1	1	1	X	X	Last Q	Last Qn	Store
1	1	↓	0	X	Last Q	Last Qn	Disabled (ignore clock)
1	1	↓	1	0	0	1	Update
1	1	↓	1	1	1	0	Update

```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D, EN);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else
      if (EN)
        begin
          Q  <= D;
          Qn <= ~D;
        end

endmodule
```

Since EN is not listed in the sensitivity list it does not trigger the block when it transitions. This "if" statement is only reached if there is a rising edge of the clock. This models an enable that is synchronous to the clock.

Example 7.5
Behavioral model of a D-flip-flop with synchronous enable in Verilog

Concept Check

CC7.1 Why is the D input not listed in the sensitivity list of a D-flip-flop?

A) To simplify the behavioral model.

B) To avoid a setup time violation if D transitions too closely to the clock.

C) Because a rising edge of clock is needed to make the assignment.

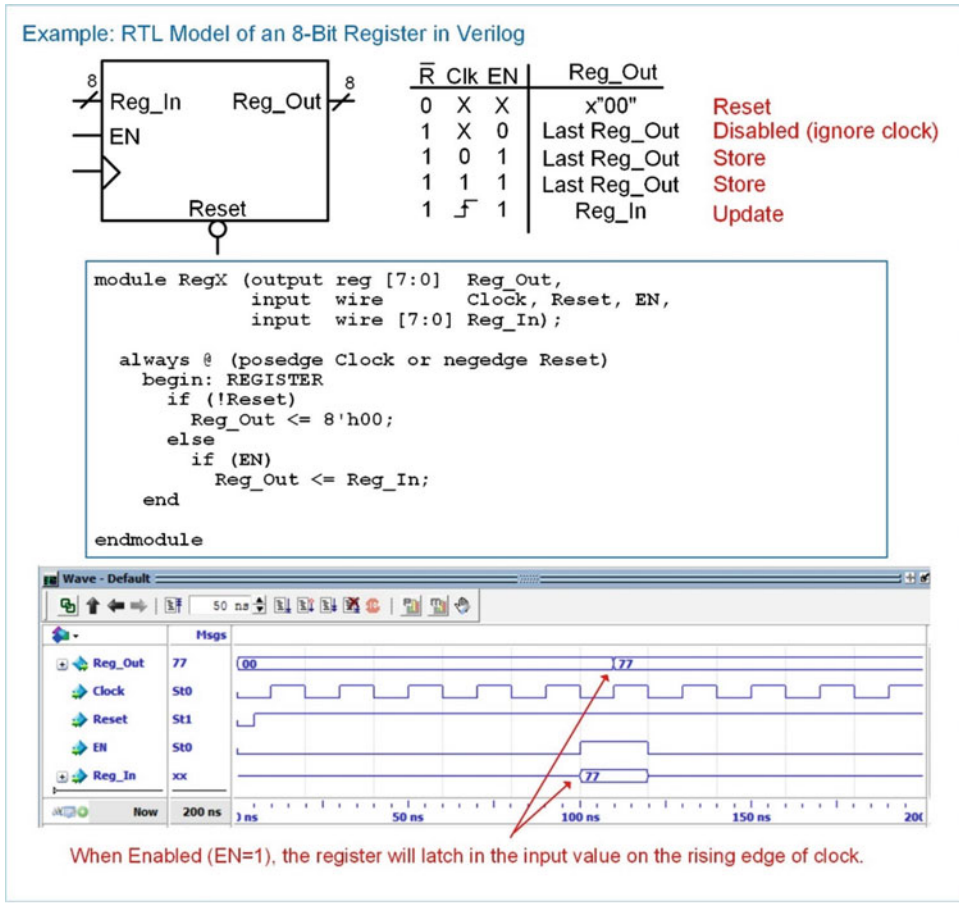
D) Because the outputs of the D-flip-flop are not updated when D changes.

7.2 Modeling Registers

7.2.1 Registers with Enables

The term *register* describes a circuit that operates in a similar manner as a D-Flip-Flop with the exception that the input and output data are vectors.This circuit is implemented with a set of D-Flip-Flops all connected to the same clock, reset, and enable inputs. A register is a higher level of abstraction that allows vector data to be stored without getting into the details of the lower-level implementation of the D-Flip-Flop components. Register Transfer Level (RTL) modeling refers to a level of design abstraction in which vector data are moved and operated on in a synchronous manner. This design methodology is

widely used in data path modeling and computer system design. Example 7.6 shows an RTL model of an 8-bit, synchronous register. This circuit has an active LOW, asynchronous reset that will cause the 8-bit output *Reg_Out* to go to 0 when it is asserted. When the reset is not asserted, the output will be updated with the 8-bit input *Reg_In* if the system is enabled (EN = 1) and there is a rising edge on the clock. If the register is disabled (EN = 0), the input clock is ignored. At all other times, the output holds its last value.

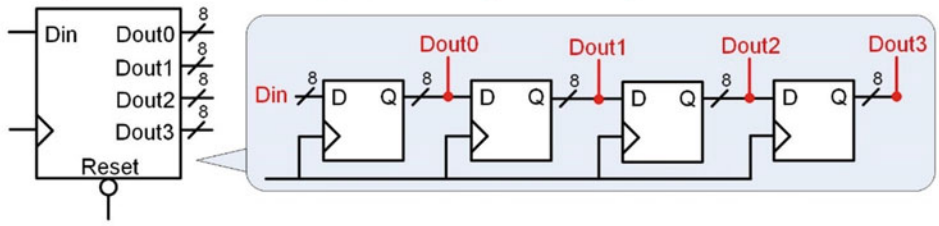


Example 7.6
RTL model of an 8-bit register in Verilog

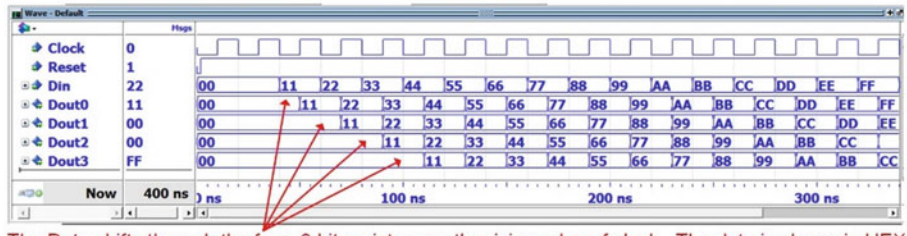
7.2.2 Shift Registers

A shift register is a circuit which consists of multiple registers connected in series. Data are shifted from one register to another on the rising edge of the clock. This type of circuit is often used in serial-to-parallel data converters. Example 7.7 shows an RTL model for a four-stage, 8-bit shift register. In the simulation waveform, the data are shown in hexadecimal format.

Example: RTL Model of a 4-Stage, 8-Bit Shift Register in Verilog




```
module Shift_Register
   (output reg [7:0]  Dout0, Dout1, Dout2, Dout3,
    input  wire       Clock, Reset,
    input  wire [7:0] Din);

  always @ (posedge Clock or negedge Reset)
    begin: SHIFT_REGISTER
      if (!Reset)
        begin
          Dout0 <= 8'h00; Dout1 <= 8'h00; Dout2 <= 8'h00; Dout3 <= 8'h00;
        end
      else
        begin
          Dout0 <= Din;   Dout1 <= Dout0; Dout2 <= Dout1; Dout3 <= Dout2;
        end
    end

endmodule
```

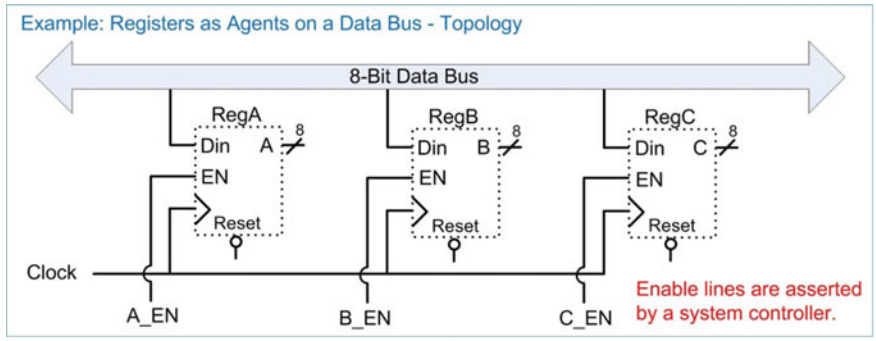


The Data shifts through the four, 8-bit registers on the rising edge of clock. The data is shown in HEX.

Example 7.7
RTL model of a four-stage, 8-bit shift register in Verilog

7.2.3 Registers as Agents on a Data Bus

One of the powerful topologies that registers can easily model is a multidrop bus. In this topology, multiple registers are connected to a data bus as receivers, or *agents*. Each agent has an enable line that controls when it latches information from the data bus into its storage elements. This topology is synchronous, meaning that each agent and the driver of the data bus are connected to the same clock signal. Each agent has a dedicated, synchronous enable line that is provided by a system controller elsewhere in the design. Example 7.8 shows this multidrop bus topology. In this example system, three registers (A, B, and C) are connected to a data bus as receivers. Each register is connected to the same clock and reset signals. Each register has its own dedicated enable line (A_EN, B_EN, and C_EN).



Example 7.8
Registers as agents on a data bus—system topology

This topology can be modeled using RTL abstraction by treating each register as a separate procedural block. Example 7.9 shows how to describe this topology with an RTL model in Verilog. Notice that the three procedural blocks modeling the A, B, and C registers are nearly identical to each other except for the signal names they use.

Example: Registers as Agents on a Data Bus – RTL Model in Verilog

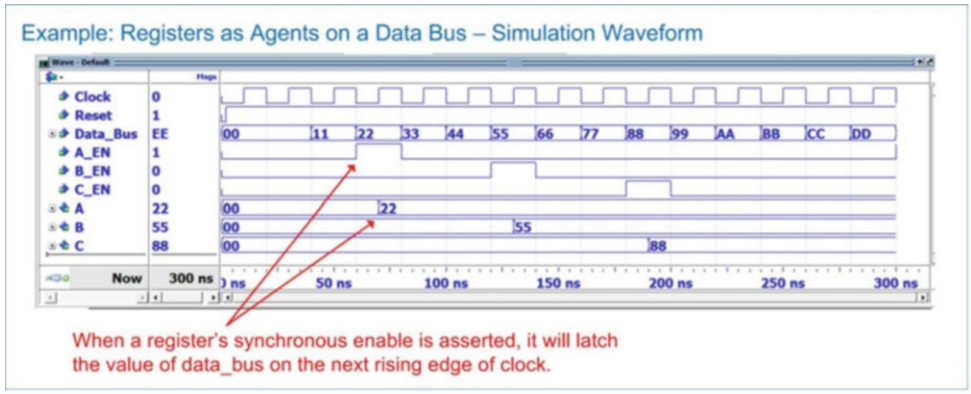
```
module MultiDropBus
   (output reg [7:0]  A, B, C,
    input  wire       Clock, Reset,
    input  wire [7:0] Data_Bus,
    input  wire       A_EN, B_EN, C_EN);

  always @ (posedge Clock or negedge Reset)
    begin: A_REG
      if (!Reset)
        A <= 8'h00;
      else
        if (A_EN ==1)
          A <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: B_REG
      if (!Reset)
        B <= 8'h00;
      else
        if (B_EN ==1)
          B <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: C_REG
      if (!Reset)
        C <= 8'h00;
      else
        if (C_EN ==1)
          C <= Data_Bus;
    end

endmodule
```

Each register is modeled as a separate block. The register has a synchronous enable that controls when it acquires data off of the data bus.

All registers are attached to the data bus as receivers.

Example 7.9
Registers as agents on a data bus—RTL model in Verilog

Example 7.10 shows the resulting simulation waveform for this system. Each register is updated with the value on the data bus whenever its dedicated enable line is asserted.



Example 7.10
Registers as agents on a data bus—simulation waveform

Concept Check

CC7.2 Does RTL modeling synthesize as combinational logic, sequential logic, or both? Why?

A) Combinational logic. Since only one process is used for each register, it will be synthesized using basic gates.

B) Sequential logic. Since the sensitivity list contains clock and reset, it will synthesize into only D-flip-flops.

C) Both. The model has a sensitivity list containing clock and reset and uses an if-else statement indicative of a D-flip-flop. This will synthesize a D-flip-flop to hold the value for each bit in the register. In addition, the ability to manipulate the inputs into the register (using either logical operators, arithmetic operators, or choosing different signals to latch) will synthesize into combinational logic in front of the D input to each D-flip-flop.

Summary

- A synchronous system is modeled with a procedural block and a sensitivity list. The clock and reset signals are always listed by themselves in the sensitivity list. Within the block is an if-else statement. The *if* clause of the statement handles the asynchronous reset condition while the *else* clause handles the synchronous signal assignments.
- Edge sensitivity is modeled within a procedural block using the *(posedge Clock or negedge reset)* syntax in the sensitivity lists.
- Most D-flip-flops and registers contain a synchronous *enable* line. This is modeled using a nested if-else statement within the main procedural block's if-else statement. The nested if-else goes beneath the clause for the synchronous signal assignments.
- Registers are modeled in Verilog in a similar manner to a D-flip-flop with a synchronous enable. The only difference is that the inputs and outputs are vectors.
- Register Transfer Level, or RTL, modeling provides a higher level of abstraction for moving and manipulating vectors of data in a synchronous manner.

Exercise Problems

Section 7.1: Modeling Scalar Storage Devices

7.1.1 How does a Verilog model for a D-flip-flop handle treating reset as the highest priority input?

7.1.2 For a Verilog model of a D-flip-flop with a synchronous enable (EN), why is not EN listed in the sensitivity list?

7.1.3 For a Verilog model of a D-flip-flop with a synchronous enable (EN), what is the impact of listing EN in the sensitivity list?

7.1.4 For a Verilog model of a D-flip-flop with a synchronous enable (EN), why is the behavior of the enable modeled using a nested if-else statement under the else clause handling the logic for the clock edge input?

Section 7.2: Modeling Registers

7.2.1 In *register transfer level* modeling, how does the width of the register relate to the number of D-flip-flops that will be synthesized?

7.2.2 In *register transfer level* modeling, how is the synchronous data movement managed if all registers are using the same clock?

7.2.3 Design a Verilog RTL model of a 32-bit, synchronous register. The block diagram for the port definition is shown in Fig. 7.1. The register has a synchronous enable. The register should be modeled using a single procedural block.

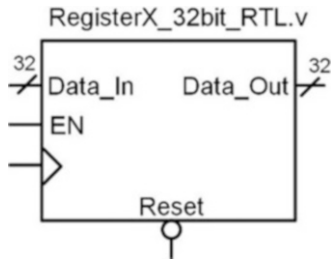


Fig. 7.1
32-bit register block diagram

7.2.4 Design a Verilog RTL model of an 8-stage, 16-bit shift register. The block diagram for the port definition is shown in Fig. 7.2. Each stage of the shift register will be provided as an output of the system (A, B, C, D, E, F, G, and H). The shift register should be modeled using a single procedural block.

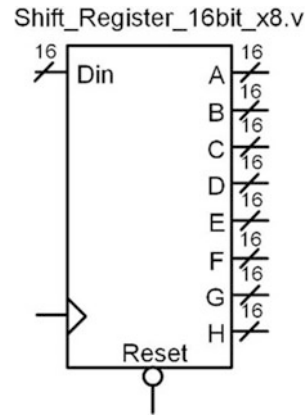


Fig. 7.2
32-bit shift register block diagram

7.2.5 Design a Verilog RTL model of the multidrop bus topology in Fig. 7.3. Each of the 16-bit registers (RegA, RegB, RegC, and RegD) will latch the contents of the 16-bit data bus if their enable line is asserted. Each register should be modeled using an individual procedural block.

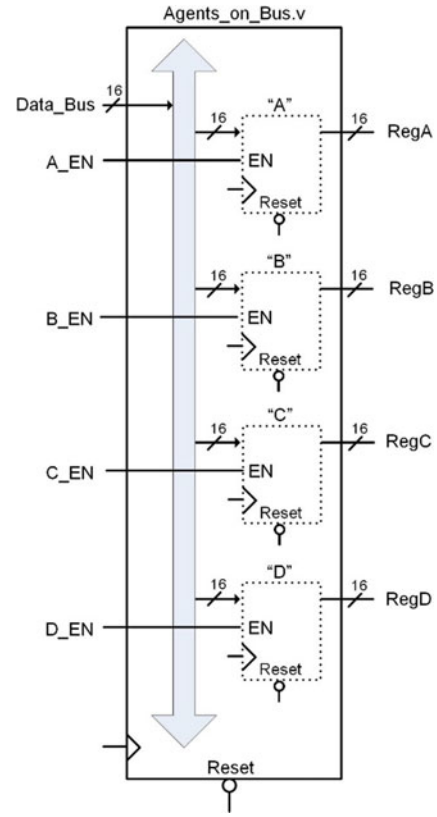


Fig. 7.3
Agents on a bus block diagram

Chapter 8: Modeling Finite State Machines

In this chapter, we will look at modeling finite state machines (FSMs). An FSM is one of the most powerful circuits in a digital system because it can make decisions about the next output based on both the current and past inputs. Finite state machines are modeled using the constructs already covered in this book. In this chapter, we will look at the widely accepted three-process model for designing a FSM.

Learning Outcomes—After completing this chapter, you will be able to:

8.1 Describe the three-process modeling approach for FSM design
8.2 Design a Verilog model for a FSM from a state diagram

8.1 The FSM Design Process and a Push-Button Window Controller Example

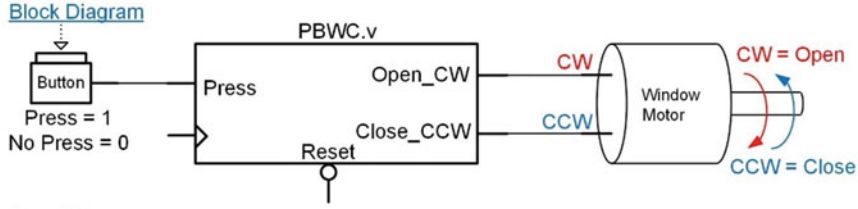
The most common modeling practice for FSMs is to declare two signals of type reg that are called *current_state* and *next_state*. Then a parameter is declared for each descriptive state name in the state diagram. A parameter also requires a value, so the state encoding can be accomplished during the parameter declaration. Once the signals and parameters are created, all of the procedural assignments in the state machine model can use the descriptive state names in their signal assignments. Within the Verilog state machine model, three separate procedural blocks are used to describe each of the functional blocks, *state memory, next state logic,* and *output logic*. In order to examine how to model a finite state machine using this approach, let us use a push-button window controller example. Example 8.1 gives the overview of the design objectives for this example and the state diagram describing the behavior to be modeled in Verilog.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_8

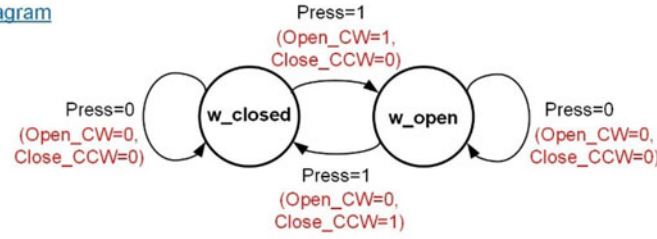
Example: Push-Button Window Controller in Verilog – Design Description

The window controller will send the appropriate control signals to a motor to open or close it whenever a button is pressed. The system must keep track whether the window is open or closed in order to send the correct signal, thus a state machine is needed. The block diagram and state diagram for this system is shown below.

Block Diagram



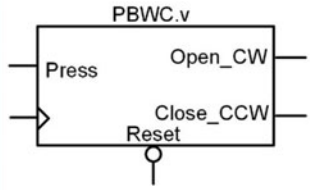
State Diagram



Example 8.1
Push-button window controller in Verilog—design description

Let us begin by defining the ports of the module. The system has an input called *Press* and two outputs called *Open_CW* and *Close_CCW*. The system also has clock and reset inputs. We will design the system to update on the rising edge of the clock and have an asynchronous, active LOW, reset. Example 8.2 shows the port definitions for this example. Note that outputs are declared as type reg while inputs are declared as type wire.

Example: Push-Button Window Controller in Verilog – Port Definition




```
module PBWC (output reg  Open_CW, Close_CCW,
             input  wire Clock, Reset, Press);

                    :
                    :
```

Outputs are defined as type reg while inputs are defined of type wire.

Example 8.2
Push-button window controller in Verilog—port definition

8.1.1 Modeling the States

Now we begin designing the finite state machine in Verilog using behavioral modeling constructs. The first step is to create two signals that will be used for the state variables. In this text, we will always name these signals *current_state* and *next_state*. The signal current_state will represent the outputs of the D-flip-flops forming the state memory and will hold the current state code. The signal next_state will represent the D inputs to the D-flip-flops forming the state memory and will receive the value from the next state logic circuitry. Since the FSM will be modeled using procedural assignment, both of these

signals will be declared of type reg. The width of the reg vector depends on the number of states in the machine and the encoding technique chosen. The next step is to declare parameters for each of the descriptive state names in the state diagram. The state encoding must be decided at this point. The following syntax shows how to declare the current_state and next_state signals and the parameters. Note that since this machine only has two states, the width of these signals is only 1-bit.

```
reg        current_state, next_state;
parameter  w_closed = 1'b0,
           w_open  = 1'b1;
```

8.1.2 The State Memory Block

Now that we have variables and parameters for the states of the FSM, we can create the model for the state memory. State memory is modeled using its own procedural block. This block models the behavior of the D-Flip-Flops in the FSM that are holding the current state on their Q outputs. Each time, there is a rising edge of the clock, the current state is updated with the next state value present on the D inputs of the D-Flip-Flops. This block must also model the reset condition. For this example, we will have the state machine go to the *w_closed* state when Reset is asserted. At all other times, the block will simply update current_state with next_state on every rising edge of the clock. The block model is very similar to the model of a D-Flip-Flop. This is as expected since this block will synthesize into one or more D-Flip-Flops to hold the current state. The sensitivity list contains only Clock and Reset and assignments are only made to the signal current_state. The following syntax shows how to model the state memory of this FSM example.

```
always @ (posedge Clock or negedge Reset)
  begin: STATE_MEMORY
    if (!Reset)
        current_state <= w_closed;
    else
        current_state <= next_state;
  end
```

8.1.3 The Next State Logic Block

Now we model the next state logic of the FSM using a second procedural block. Recall that the next state logic is combinational logic, thus we need to include all of the input signals that the circuit considers in the next state calculation in the sensitivity list. The current_state signal will always be included in the sensitivity list of the next state logic block in addition to any inputs to the system. For this example, the system has one other input called *Press*. This block makes assignments to the next_state signal. It is common to use a case statement to separate out the assignments that occur at each state. At each state within the case statement, an if-else statement is used to model the assignments for different input conditions on Press. The following syntax shows how to model the next state logic of this FSM example. Notice that we include a *default* clause in the case statement to ensure that the state machine has a path back to the reset state in the case of an unexpected fault.

```
always @ (current_state or Press)
  begin: NEXT_STATE_LOGIC
    case (current_state)
      w_closed : if (Press == 1'b1) next_state = w_open;  else next_state = w_closed;
      w_open   : if (Press == 1'b1) next_state = w_closed; else next_state = w_open;
      default  : next_state = w_closed;
  endcase
end
```

8.1.4 The Output Logic Block

Now we model the output logic of the FSM using a third procedural block. Recall that output logic is combinational logic, thus we need to include all of the input signals that this circuit considers in the output assignments. The current_state will always be included in the sensitivity list. If the FSM is a Mealy machine, then the system inputs will also be included in the sensitivity list. If the machine is a Moore machine, then only the current_state will be present in the sensitivity list. For this example, the FSM is a Mealy machine, so the input Press needs to be included in the sensitivity list. Note that this block only makes assignments to the outputs of the machine (Open_CW and Close_CCW). The following syntax shows how to model the output logic of this FSM example. Again, we include a *default* clause to ensure that the state machine has explicit output behavior in the case of a fault.

```
always @ (current_state or Press)
  begin: OUTPUT_LOGIC
    case (current_state)
      w_closed : if (Press == 1'b1)
                  begin
                    Open_CW  = 1'b1;
                    Close_CCW = 1'b0;
                  end
                 else
                  begin
                    Open_CW  = 1'b0;
                    Close_CCW = 1'b0;
                  end
          w_open : if (Press == 1'b1)
                    begin
                      Open_CW  = 1'b0;
                      Close_CCW = 1'b1;
                    end
                   else
                    begin
                      Open_CW  = 1'b0;
                      Close_CCW = 1'b0;
                    end
          default : begin
                     Open_CW  = 1'b0;
                     Close_CCW = 1'b0;
                    end
    endcase
  end
```

Putting this all together yields a behavioral model for the FSM that can be simulated and synthesized. Example 8.3 shows the entire model for this example.

Example: Push-Button Window Controller in Verilog – Full Model

```
module PBWC (output reg  Open_CW, Close_CCW,
             input  wire Clock, Reset, Press);

  reg       current_state, next_state;
  parameter w_closed = 1'b0,
            w_open   = 1'b1;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= w_closed;
      else
        current_state <= next_state;
    end

  always @ (current_state or Press)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        w_closed : if (Press == 1'b1)
                     next_state = w_open;
                   else
                     next_state = w_closed;
        w_open   : if (Press == 1'b1)
                     next_state = w_closed;
                   else
                     next_state = w_open;
        default  : next_state = w_closed;
      endcase
    end

  always @ (current_state or Press)
     begin: OUTPUT_LOGIC
       case (current_state)
         w_closed : if (Press == 1'b1)
                      begin
                        Open_CW  = 1'b1;
                        Close_CCW = 1'b0;
                      end
                    else
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                      end
         w_open   : if (Press == 1'b1)
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b1;
                      end
                    else
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                      end
          default : begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                    end
      endcase
    end

endmodule
```

Declaration of state variables and state encoding.

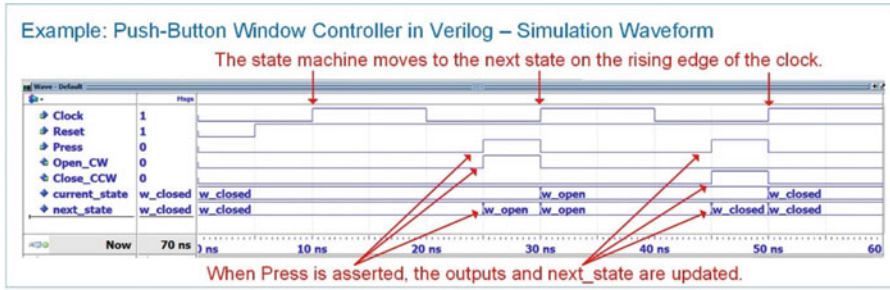
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since this is a Mealy machine, the current state and input are listed in the sensitivity list. This block only makes assignments to the outputs "Open_CW" and "Close_CCW".

Example 8.3
Push-button window controller in Verilog—full model

Example 8.4 shows the simulation waveform for this state machine. This functional simulation was performed using ModelSim-Altera Starter Edition 10.1d. A macro file was used to display the current and next state variables using their parameter names instead of their state codes. This allows the functionality of the FSM to be more easily observed. This approach will be used for the rest of the FSM examples in this book.

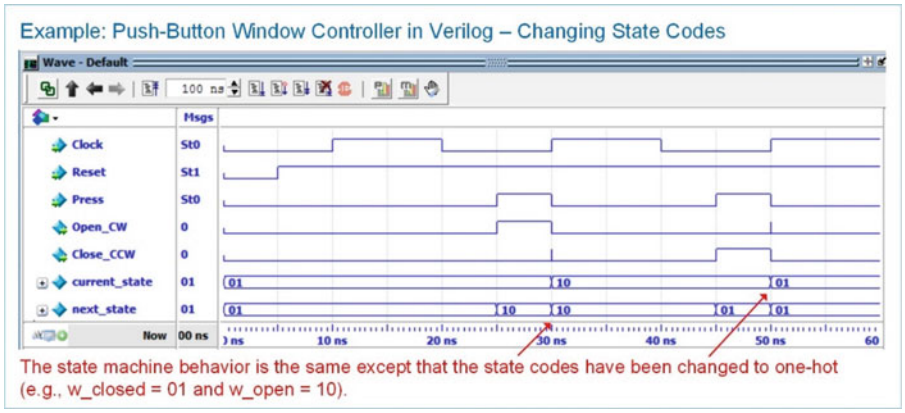


Example 8.4
Push-button window controller in Verilog—simulation waveform

8.1.5 Changing the State Encoding Approach

In the prior example, we had two states that were encoded as: w_closed = 1'b0; w_open_1'b1. This encoding technique is considered *binary* and takes 1-bit; however, a *one-hot* could be adopted that would require 2-bits. The way that state variables and state codes are assigned in Verilog makes is straightforward to change the state codes. The only consideration that must be made is expanding the size of the current_state and next_state variables to accommodate the new state codes. The following example shows how the state encoding would look if a *one-hot* approach was used (w_closed = 2'b01; w_open_2'b10). Note that the state variables now must be two bits wide.This means the state variables need to be declared as type reg[1:0]. Example 8.5 shows the resulting simulation waveforms. The simulation waveform shows the value of the state codes instead of the state names.

```
reg [1:0] current_state, next_state;
parameter w_closed = 2'b01,
          w_open  = 2'b10;
```



Example 8.5
Push-button window controller in Verilog—changing state codes

CONCEPT CHECK

CC8.1 Why is it always a good design approach to model a generic finite state machine using three processes?

A) For readability.

B) So that it is easy to identify whether the machine is a Mealy or Moore.

C) So that the state memory process can be reused in other FSMs.

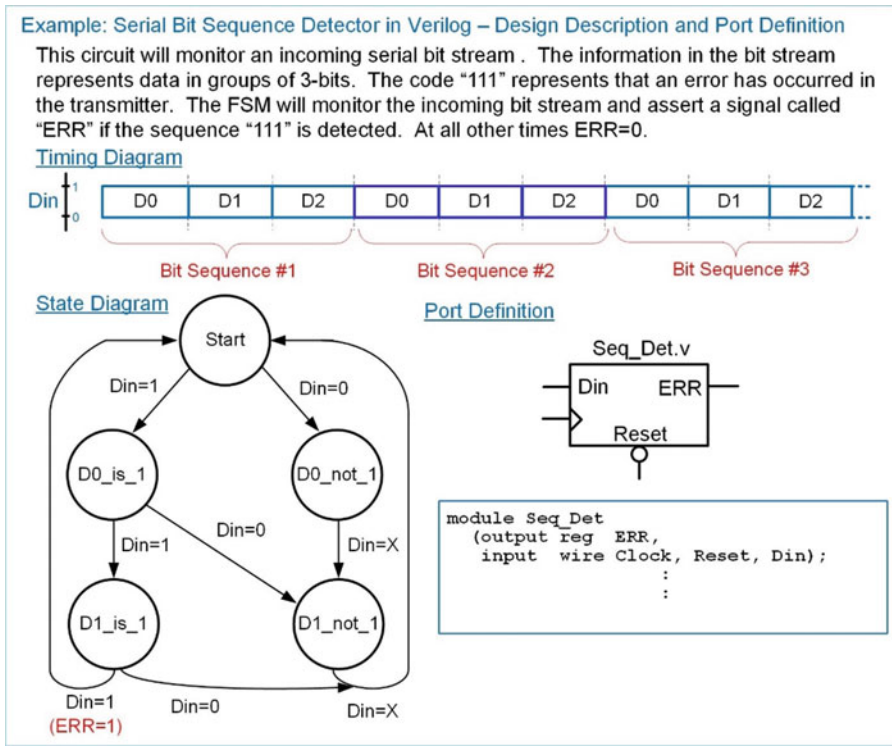
D) Because each of the three subsystems of a FSM has unique inputs and outputs that should be handled using dedicated processes.

8.2 FSM Design Examples

This section presents a set of example finite state machine designs using the behavioral modeling constructs of Verilog.

8.2.1 Serial Bit Sequence Detector in Verilog

Let us look at the design of the serial bit sequence detector finite state machine using the behavioral modeling constructs of Verilog. Example 8.6 shows the design description and port definition for this state machine.



Example 8.6
Serial bit sequence detector in Verilog—design description and port definition

Example 8.7 shows the full model for the serial bit sequence detector. Notice that the states are encoded in binary, which requires three bits for the variables current_state and next_state.

Example: Serial Bit Sequence Detector in Verilog – Full Model

```
module Seq_Det (output reg  ERR,
                input  wire Clock, Reset, Din);

  reg [2:0] current_state, next_state;
  parameter Start    = 3'b000,
            D0_is_1  = 3'b001,
            D1_is_1  = 3'b010,
            D0_not_1 = 3'b011,
            D1_not_1 = 3'b100;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= Start;
      else
        current_state <= next_state;
    end

  always @ (current_state or Din)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        Start    : if (Din == 1'b1)
                      next_state = D0_is_1;
                   else
                      next_state = D0_not_1;
        D0_is_1  : if (Din == 1'b1)
                      next_state = D1_is_1;
                   else
                      next_state = D1_not_1;
        D1_is_1  : next_state = Start;
        D0_not_1 : next_state = D1_not_1;
        D1_not_1 : next_state = Start;
        default  : next_state = Start;
      endcase
    end

  always @ (current_state or Din)
    begin: OUTPUT_LOGIC
      case (current_state)
        D1_is_1  : if (Din == 1'b1)
                      ERR = 1'b1;
                   else
                      ERR = 1'b0;

        default  : ERR = 1'b0;
      endcase
    end

endmodule
```

Declaration of state variables and state encoding.

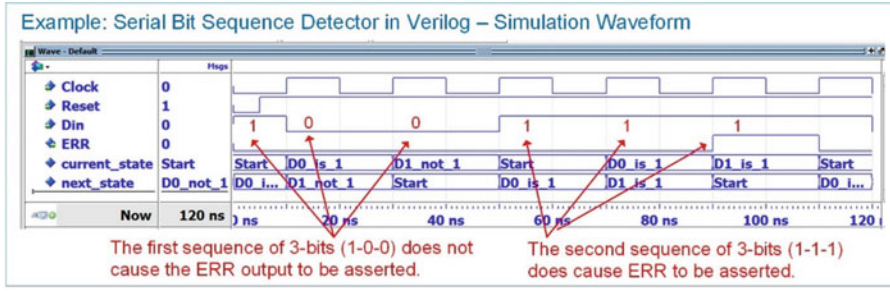
State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since there is only one condition where the output "ERR" is asserted, the default clause can be used for all other conditions.

Example 8.7
Serial bit sequence detector in Verilog—full model

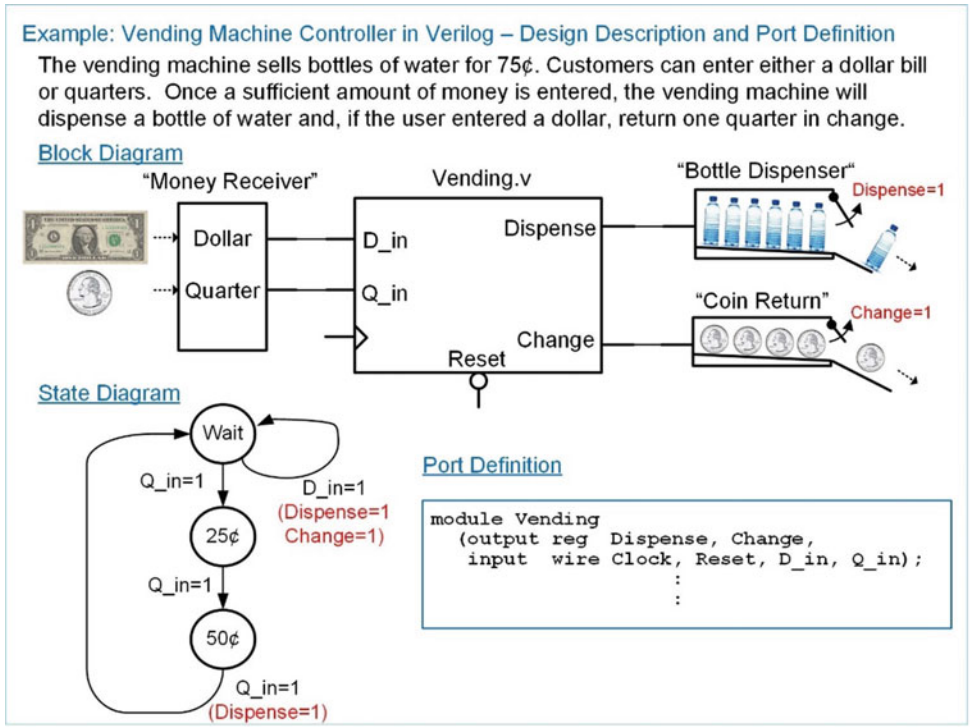
Example 8.8 shows the functional simulation waveform for this design.



Example 8.8
Serial bit sequence detector in Verilog—simulation waveform

8.2.2 Vending Machine Controller in Verilog

Let us now look at the design of the vending machine controller using the behavioral modeling constructs of Verilog. Example 8.9 shows the design description and port definition.



Example 8.9
Vending machine controller in verilog—design description and port definition

Example 8.10 shows the full model for the vending machine controller. In this model, the descriptive state names Wait, 25¢, and 50¢ cannot be used directly. This is because Verilog user-defined names cannot begin with a number. Instead, the letter "s" is placed in front of the state names in order to make them legal Verilog names (i.e., sWait, s25, s50).

Example: Vending Machine Controller in Verilog – Full Model

```
module Vending (output reg  Dispense, Change,
                input  wire Clock, Reset, D_in, Q_in);

  reg [1:0] current_state, next_state;
  parameter sWait = 2'b00, s25 = 2'b01, s50 = 2'b10;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= sWait;
      else
        current_state <= next_state;
    end

  always @ (current_state or D_in or Q_in)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        sWait    : if (Q_in == 1'b1)
                     next_state = s25;
                   else
                     next_state = sWait;
        s25      : if (Q_in == 1'b1)
                     next_state = s50;
                   else
                     next_state = s25;
        s50      : if (Q_in == 1'b1)
                     next_state = sWait;
                   else
                     next_state = s50;
        default  : next_state = sWait;
      endcase
    end

   always @ (current_state or D_in or Q_in)
     begin: OUTPUT_LOGIC
       case (current_state)
         sWait    : if (D_in == 1'b1)
                      begin
                        Dispense = 1'b1; Change = 1'b1;
                      end
                    else
                      begin
                        Dispense = 1'b0; Change = 1'b0;
                      end
         s25      : begin
                      Dispense = 1'b0; Change = 1'b0;
                    end
         s50      : if (Q_in == 1'b1)
                      begin
                        Dispense = 1'b1; Change = 1'b0;
                      end
                    else
                      begin
                        Dispense = 1'b0; Change = 1'b0;
                      end
         default  : begin
                      Dispense = 1'b0; Change = 1'b0;
                    end
       endcase
     end
endmodule
```

State variables and state encoding.

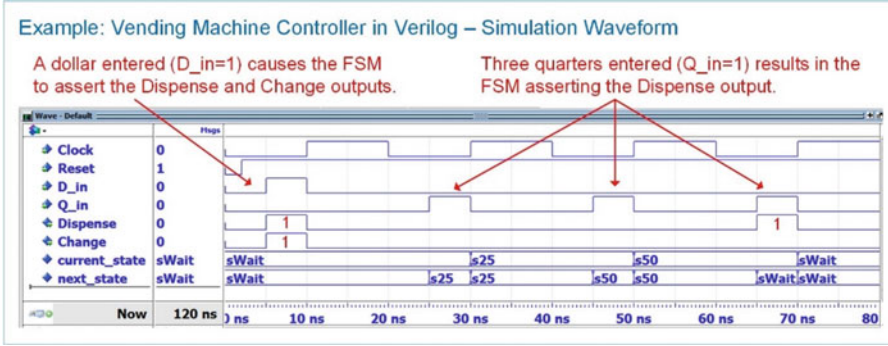
State memory block.

Next state logic block.

Output logic block.

Example 8.10
Vending machine controller in Verilog—full model

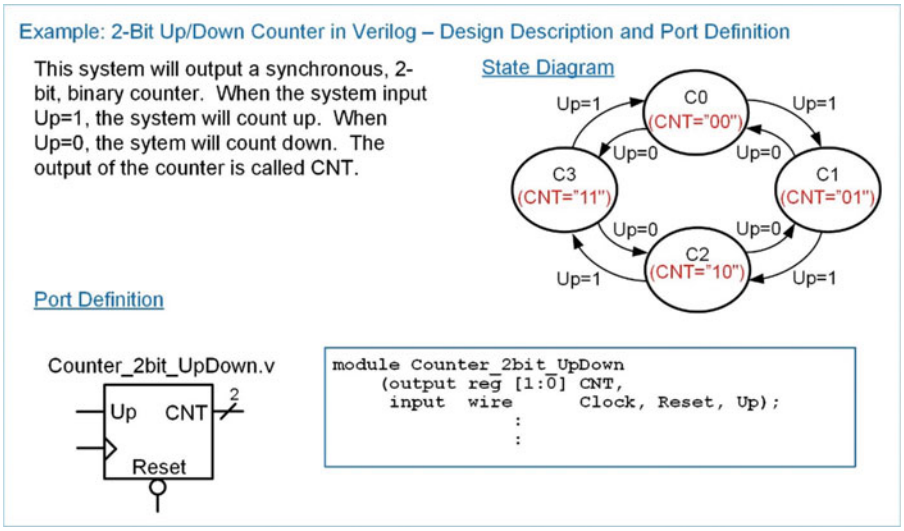
Example 8.11 shows the resulting simulation waveform for this design.



Example 8.11
Vending machine controller in Verilog—simulation waveform

8.2.3 2-Bit, Binary Up/Down Counter in Verilog

Let us now look at how a simple counter can be implemented using the three-block behavioral modeling approach in Verilog. Example 8.12 shows the design description and port definition for the 2-bit, binary up/down counter FSM.



Example 8.12
2-bit up/down counter in Verilog—design description and port definition

Example 8.13 shows the full model for the 2-bit up/down counter using the three-block modeling approach. Since a counter's outputs only depend on the current state, counters are Moore machines. This simplifies the output logic block since it only needs to contain the current state in its sensitivity list.

Example: 2-Bit Up/Down Counter in Verilog – Full Model (Three Block Approach)

```
module Counter_2bit_UpDown (output reg [1:0] CNT,
                            input  wire      Clock, Reset, Up);

  reg [1:0] current_state, next_state;
  parameter C0 = 2'b00,
            C1 = 2'b01,
            C2 = 2'b10,
            C3 = 2'b11;

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= C0;
      else
        current_state <= next_state;
    end

  always @ (current_state or Up)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        C0      : if (Up == 1'b1) next_state = C1; else next_state = C3;
        C1      : if (Up == 1'b1) next_state = C2; else next_state = C0;
        C2      : if (Up == 1'b1) next_state = C3; else next_state = C1;
        C3      : if (Up == 1'b1) next_state = C0; else next_state = C2;
        default : next_state = C0;
      endcase
    end

   always @ (current_state)
     begin: OUTPUT_LOGIC
       case (current_state)
         C0      : CNT = 2'b00;
         C1      : CNT = 2'b01;
         C2      : CNT = 2'b10;
         C3      : CNT = 2'b11;
         default : CNT = 2'b00;
       endcase
   end

endmodule
```

State variables and state encoding.

State memory block.

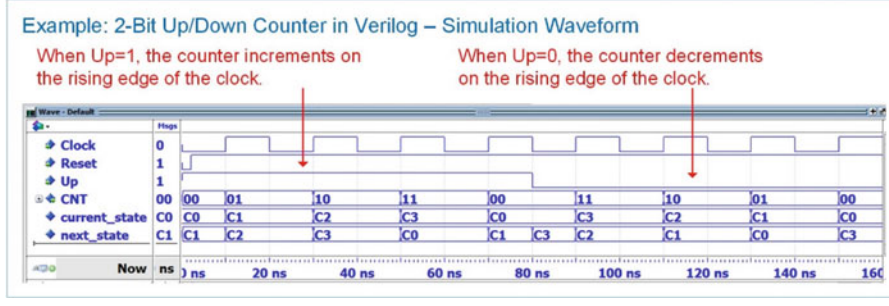
Next state logic block.

Output logic block. Note that since this is a Moore machine only the current state is listed in the sensitivity list.

Example 8.13
2-bit up/down counter in Verilog—full model (three-block approach)

Example 8.14 shows the resulting simulation waveform for this counter finite state machine.

Example: 2-Bit Up/Down Counter in Verilog – Simulation Waveform



Example 8.14
2-bit up/down counter in Verilog—simulation waveform

Concept Check

CC8.2 The procedural block for the state memory is nearly identical for all finite state machines with one exception. What is it?

A) The sensitivity list may need to include a preset signal.

B) Sometimes it is modeled using an SR latch storage approach instead of with D-flip-flop behavior.

C) The name of the reset state will be different.

D) The current_state and next_state signals are often swapped.

Summary

- Generic finite state machines are modeled using three separate procedural blocks that describe the behavior of the next state logic, the state memory, and the output logic. Separate blocks are used because each of the three functions in a FSM are dependent on different input signals.
- In Verilog, descriptive state names can be created for a FSM using parameters. Two signals are first declared called *current_state* and *next_state* of type reg. Then a parameter is defined for each unique state in the machine with the state name and desired state code. Throughout the rest of the model, the unique state names can be used as both assignments to current_state/next_state and as inputs in case and if-else statements. This approach allows the model to be designed using readable syntax while providing a synthesizable design.

Exercise Problems

Section 8.1: The FSM Design Process

8.1.1 What is the advantage of using *parameters* for the state when modeling a finite state machine?

8.1.2 What is the advantage of having to assign the state codes during the parameter declaration for the state names when modeling a finite state machine?

8.1.3 When using the three-procedural block behavioral modeling approach for FSMs, does the next state logic block model combinational or sequential logic?

8.1.4 When using the three-procedural block behavioral modeling approach for FSMs, does the state memory block model combinational or sequential logic?

8.1.5 When using the three-procedural block behavioral modeling approach for FSMs, does the output logic block model combinational or sequential logic?

8.1.6 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the next state logic block?

8.1.7 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the state memory block?

8.1.8 When using the three-procedural block behavioral modeling approach for FSMs, what inputs are listed in the sensitivity list of the output logic block?

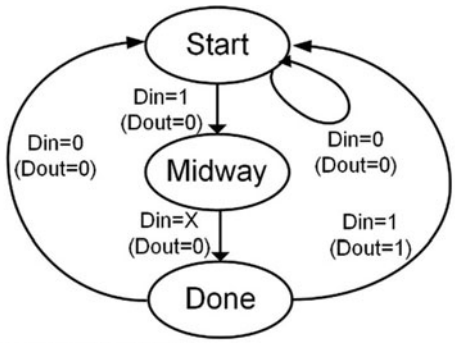
8.1.9 When using the three-procedural block behavioral modeling approach for FSMs, how can the signals listed in the sensitivity list of the output logic block immediately indicate whether the FSM is a Mealy or a Moore machine?

8.1.10 Why is it not a good design approach to combine the next state logic and output logic behavior into a single procedural block?

Section 8.2: FSM Design Examples

8.2.1 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.1. Use the port

definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in binary using the following state codes: Start = "00," Midway = "01," Done = "10."



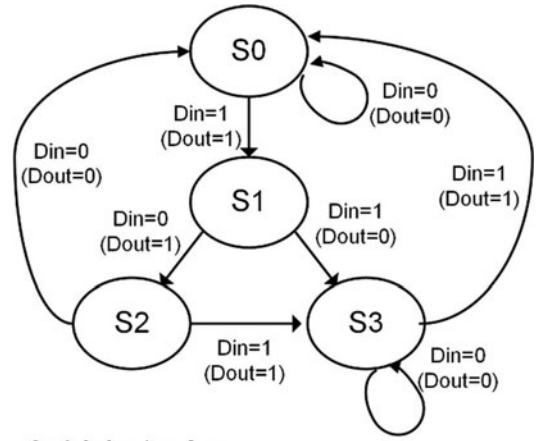
fsm1_behavioral.v

```
module fsm1_behavioral
   (output reg  Dout,
    input  wire Clock, Reset, Din);
                :
```

Fig. 8.1
FSM 1 state diagram and port definition

8.2.2 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.1. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in one-hot using the following state codes: Start = "001," Midway = "010," Done = "100."

8.2.3 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.2. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in binary using the following state codes: S0 = "00," S1 = "01," S2 = "10," and S3 = "11."



fsm2_behavioral.v

```
module fsm2_behavioral
   (output reg  Dout,
    input  wire Clock, Reset, Din);
                :
```

Fig. 8.2
FSM 2 state diagram and port definition

8.2.4 Design a Verilog behavioral model to implement the finite state machine described by the state diagram in Fig. 8.2. Use the port definition provided in this figure for your design. Use the three-block approach to modeling FSMs described in this chapter for your design. Model the state variables using parameters and encode the states in one-hot using the following state codes: S0 = "0001," S1 = "0010," S2 = "0100," and S3 = "1000."

8.2.5 Design a Verilog behavioral model for a 4-bit serial bit sequence detector similar to Example 8.6. Use the port definition provided in Fig. 8.3. Use the three-block approach to modeling FSMs described in this chapter for your design. The input to your sequence detector is called *DIN* and the output is called *FOUND*. Your detector will assert FOUND anytime there is a 4-bit sequence of "0101." Model the states in this machine with parameters. Choose any state encoding approach you wish.

Seq_Det_behavioral.v

```
module Seq_Det_behavioral
   (output reg  FOUND,
    input  wire Clock, Reset,
    input  wire DIN);
                :
```

Fig. 8.3
Sequence detector port definition

8.2.6 Design a Verilog behavioral model for a 20-¢ vending machine controller similar to Example 8.9. Use the port definition provided in Fig. 8.4. Use the three-block approach to modeling FSMs described in this chapter for your design. Your controller will take in nickels and dimes and dispense a product anytime the customer has entered 20¢. Your FSM has two inputs, *Nin* and *Din*. Nin is asserted whenever the customer enters a nickel while Din is asserted anytime the customer enters a dime. Your FSM has two outputs, *Dispense* and *Change*. Dispense is asserted anytime the customer has entered at least 20¢ and Change is asserted anytime the customer has entered more than 20¢ and needs a nickel in change. Model the states in this machine with parameters. Choose any state encoding approach you wish.

Vending_behavioral.v

```
module Vending_behavioral
   (output reg  Dispense, Change,
    input  wire Clock, Reset,
    input  wire Nin, Din);
                :
```

Fig. 8.4
Vending machine port definition

8.2.7 Design a Verilog behavioral model for a finite state machine for a traffic light controller. Use the port definition provided in Fig. 8.5. This time, you will implement the functionality using the behavioral modeling techniques presented in this chapter. Your FSM will control a traffic light at the intersection of a busy highway and a seldom used side road. You will be designing the control signals for just the red, yellow, and green lights facing the highway. Under normal conditions, the highway has a green light. The side road has car detector that indicates when car pulls up by asserting a signal called *CAR*. When CAR is asserted, you will change the highway traffic light from green to yellow, and then from yellow to red. Once in the red position, a built-in timer will begin a countdown and provide your controller a signal called *TIMEOUT* when 15 seconds have passed. Once TIMEOUT is asserted, you will change the highway traffic light back to green. Your system will have three outputs *GRN*, *YLW*, and *RED*, which control when the highway facing traffic lights are on (1 = ON, 0 = OFF). Model the states in this machine with parameters. Choose any state encoding approach you wish.

tlc_behavioral.v

```
module tlc_behavioral
   (output reg  GRN, YLW, RED,
    input  wire Clock, Reset,
    input  wire CAR, TIMEOUT);
                :
```

Fig. 8.5
Traffic light controller port definition

Chapter 9: Modeling Counters

Counters are a special case of finite state machines (FSMs) because they move linearly through their discrete states (either forward or backward) and typically are implemented with state-encoded outputs. Due to this simplified structure and wide spread use in digital systems, Verilog allows counters to be modeled using a single procedural block with arithmetic operators (i.e., + and −).This enables a more compact model and allows much wider counters to be implemented in a practical manner. This chapter will cover some of the most common techniques for modeling counters.

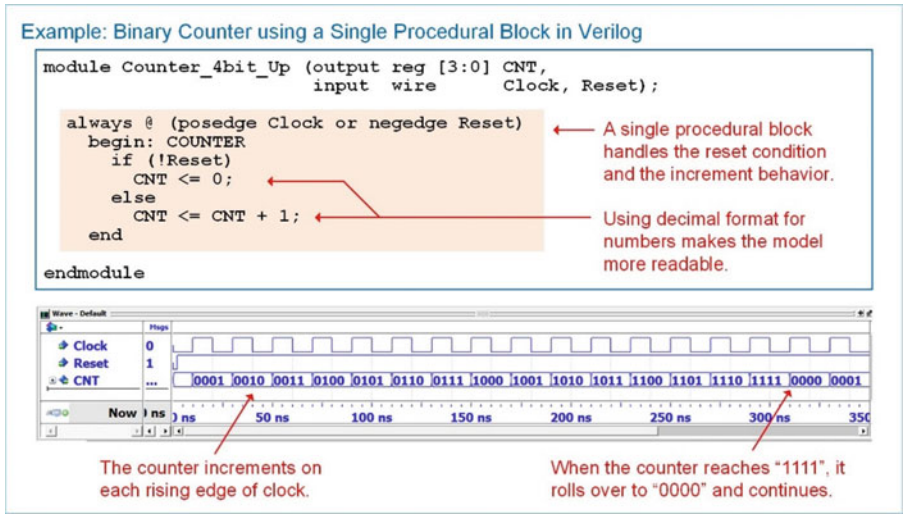
Learning Outcomes—After completing this chapter, you will be able to:

9.1 Design a behavioral model for a counter using a single procedural block
9.2 Design a behavioral model for a counter with enable and load capability

9.1 Modeling Counters with a Single Procedural Block

9.1.1 Counters in Verilog Using the Type Reg

Let us look at how we can model a 4-bit, binary up counter with an output called *CNT*. We want to model this counter using the "+" operator to avoid having to explicitly define a state code for each state as in the three-block modeling approach to FSMs. The "+" operator works on the type reg, so the counting behavior can simply be modeled using CNT <= CNT + 1. The procedural block also needs to handle the reset condition. Both the Clock and Reset signals are listed in the sensitivity list. Within the block, an if-else statement is used to handle both the reset and increment behaviors. Example 9.1 shows the Verilog model and simulation waveform for this counter. When the counter reaches its maximum value of "1111," it rolls over to "0000" and continues counting because it is declared to only contain 4-bits.



Example 9.1
Binary counter using a single procedural block in Verilog

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_9

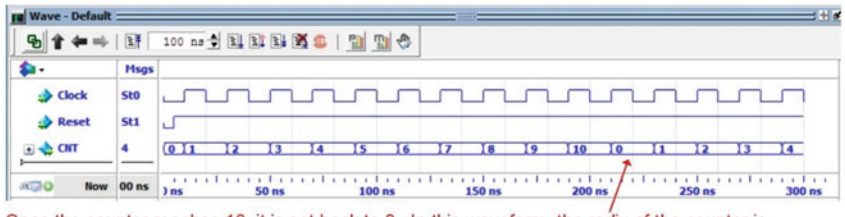
9.1.2 Counters with Range Checking

When a counter needs to have a maximum range that is different from the maximum binary value of the count vector (i.e., $<2^n - 1$), then the procedural block needs to contain *range checking* logic. This can be modeled by inserting a nested if-else statement beneath of the else clause that handles the behavior for when the counter receives a rising clock edge. This nested if-else first checks whether the count has reached its maximum value. If it has, it is reset back to it minimum value. If it has not, the counter is incremented as usual. Example 9.2 shows the Verilog model and simulation waveform for a counter with a minimum count value of 0_{10} and a maximum count value of 10_{10}. This counter still requires 4-bits to be able to encode 10_{10}.

Example: Binary Counter with Range Checking in Verilog

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire      Clock, Reset);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (CNT == 10)
          CNT <= 0;
        else
          CNT <= CNT + 1;
    end

endmodule
```

A nested if-else statement checks if the counter has reached its maximum value. If it has, it is reset back to zero. If it hasn't, it increments.



Once the counter reaches 10, it is set back to 0. In this waveform, the radix of the counter is formatted as unsigned decimal.

Example 9.2
Binary counter with range checking in Verilog

Concept Check

CC9.1 If a counter is modeled using only one procedural block in Verilog, is it still a finite state machine? Why or why not?

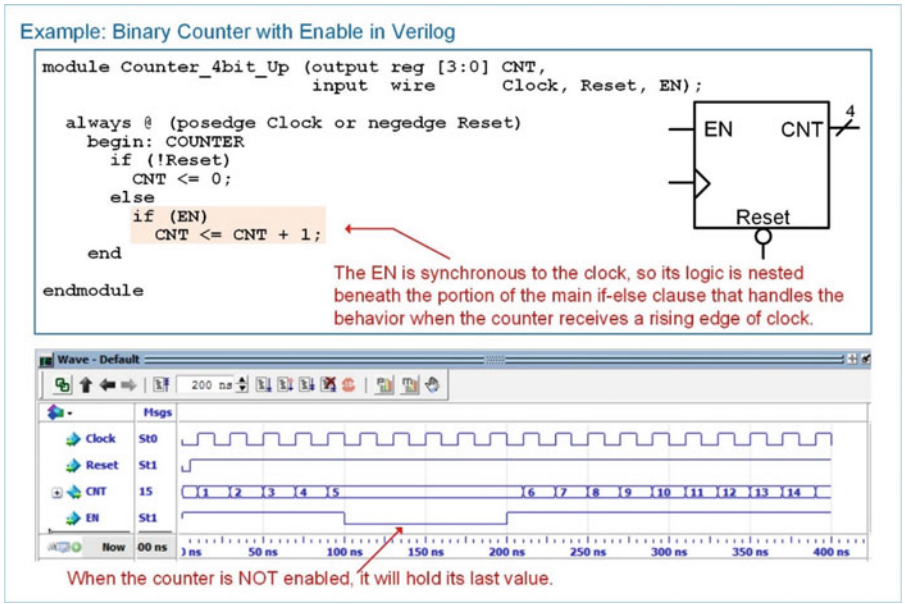
A) Yes. It is just a special case of a FSM that can easily be modeled using one block Synthesizers will recognize the single block model as a FSM.

B) No. Using only one block will synthesize into combinational logic. Without the ability to store a state, it is not a finite state machine.

9.2 Counter with Enables and Loads

9.2.1 Modeling Counters with Enables

Including an *enable* in a counter is a common technique to prevent the counter from running continuously. When the enable is asserted, the counter will increment on the rising edge of the clock as usual. When the enable is deasserted, the counter will simply hold its last value. Enable lines are synchronous, meaning that they are only evaluated on the rising edge of the clock. As such, they are modeled using a nested if-else statement within the main if-else statement checking for a rising edge of the clock. Example 9.3 shows an example model for a 4-bit counter with enable.



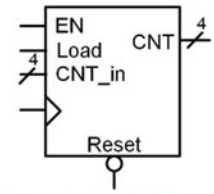
Example 9.3
Binary counter with enable in Verilog

9.2.2 Modeling Counters with Loads

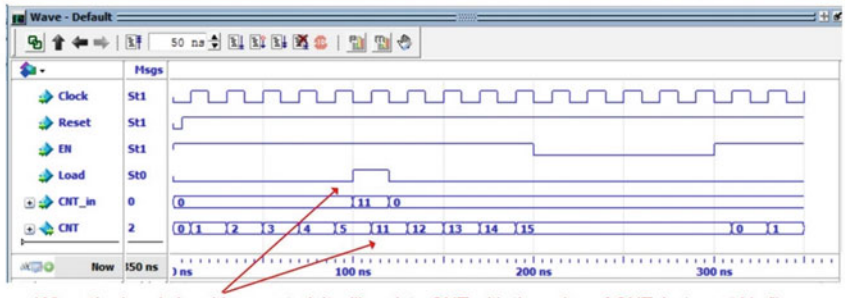
A counter with a *load* has the ability to set the counter to a specified value. The specified value is provided on an input port (i.e., CNT_in) with the same width as the counter output (CNT). A synchronous load input signal (i.e., Load) is used to indicate when the counter should set its value to the value present on CNT_in. Example 9.4 shows an example model for a 4-bit counter with load capability.

Example: Binary Counter with Load in Verilog

```
module Counter_4bit_Up (output reg [3:0]  CNT,
                        input  wire       Clock, Reset, EN, Load,
                        input  wire [3:0] CNT_in);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (EN)
          if (Load)
            CNT <= CNT_in;
          else
            CNT <= CNT + 1;
    end

endmodule
```



A nested if-else statement is used to load CNT with CNT_in when the Load signal is asserted and the counter receives a rising edge of clock.



When the Load signal is asserted, it will update CNT with the value of CNT_in (e.g., "11_{10}").

Example 9.4
Binary counter with load in Verilog

Concept Check

CC9.2 If a counter is modeled using only one procedural block in Verilog, is it still a finite state machine? Why or why not?

A) Yes. It is just a special case of a FSM that can easily be modeled using one block. Synthesizers will recognize the single block model as a FSM.

B) No. Using only one block will synthesize into combinational logic. Without the ability to store a state, it is not a finite state machine.

Summary

- Counters are a special type of finite state machine that can be modeled using a single procedural block. Only the clock and reset signals are listed in the sensitivity list of the counter block.
- Registers are modeled in Verilog in a similar manner to a D-flip-flop with a synchronous enable. The only difference is that the inputs and outputs are vectors.
- Register Transfer Level, or RTL, modeling provides a higher level of abstraction for moving and manipulating vectors of data in a synchronous manner.

Exercise Problems

Section 9.1: Modeling Counters with a Single Procedural Block

9.1.1 Design a Verilog behavioral model for a 16-bit, binary up counter using a single procedural block. The block diagram for the port definition is shown in Fig. 9.1.

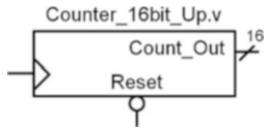


Fig. 9.1
16-bit binary up counter block diagram

9.1.2 Design a Verilog behavioral model for a 16-bit, binary up counter with range checking using a single procedural block. The block diagram for the port definition is shown in Fig. 9.1. Your counter should count up to 60,000 and then start over at 0.

Section 9.2: Counters with Enables and Loads

9.2.1 Design a Verilog behavioral model for a 16-bit, binary up counter with enable using a single procedural block. The block diagram for the port definition is shown in Fig. 9.2.

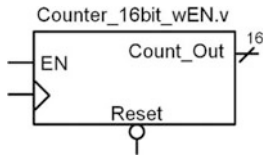


Fig. 9.2
16-bit binary up counter with enable block diagram

9.2.2 Design a Verilog behavioral model for a 16-bit, binary up counter with enable and load using a single procedural block. The block diagram for the port definition is shown in Fig. 9.3.

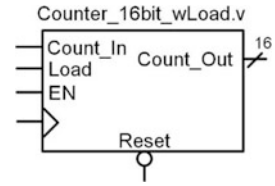


Fig. 9.3
16-bit binary up counter with load block diagram

9.2.3 Design a Verilog behavioral model for a 16-bit, binary up/down counter using a single procedural block. The block diagram for the port definition is shown in Fig. 9.4. When Up $=$ 1, the counter will increment. When Up $=$ 0, the counter will decrement.

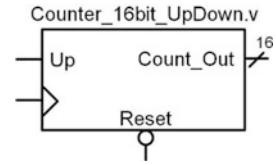


Fig. 9.4
16-bit binary up/down counter block diagram

Chapter 10: Modeling Memory

This chapter covers how to model memory arrays in Verilog. These models are technology independent, meaning that they can be ultimately synthesized into a wide range of semiconductor memory devices.

Learning Outcomes—After completing this chapter, you will be able to:

10.1 Describe the basic architecture and terminology for semiconductor-based memory systems
10.2 Design a Verilog model for a read-only memory array
10.3 Design a Verilog model for a read/write memory array

10.1 Memory Architecture & Terminology

The term *memory* is used to describe a system with the ability to store digital information. The term *semiconductor memory* refers to systems that are implemented using integrated circuit technology. These types of systems store the digital information using transistors, fuses, and/or capacitors on a single semiconductor substrate. Memory can also be implemented using technology other than semiconductors. Disk drives store information by altering the polarity of magnetic fields on a circular substrate. The two magnetic polarities (north and south) are used to represent different logic values (i.e., 0 or 1). Optical disks use lasers to burn pits into reflective substrates. The binary information is represented by light either being reflected (no pit) or not reflected (pit present). Semiconductor memory does not have any moving parts, so it is called *solid state memory* and can hold more information per unit area than disk memory. Regardless of the technology used to store the binary data, all memory has common attributes and terminology that are discussed in this chapter.

10.1.1 Memory Map Model

The information stored in memory is called the **data**. When information is placed into memory, it is called a **write**. When information is retrieved from memory, it is called a **read**. In order to access data in memory, an **address** is used. While data can be accessed as individual bits, in order to reduce the number of address locations needed, data are typically grouped into *N-bit words*. If a memory system has $N = 8$, this means that 8-bits of data are stored at each address. The number of address locations are described using the variable M. The overall size of the memory is typically stated by saying "$M \times N$." For example, if we had a 16×8 memory system, that means that there are 16 address locations, each capable of storing a byte of data. This memory would have a **capacity** of $16 \times 8 = 128$ bits. Since the address is implemented as a binary code, the number of lines in the address bus (n) will dictate the number of address locations that the memory system will have ($M = 2^n$). Figure 10.1 shows a graphical depiction of how data resides in memory. This type of graphic is called a *memory map model*.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_10

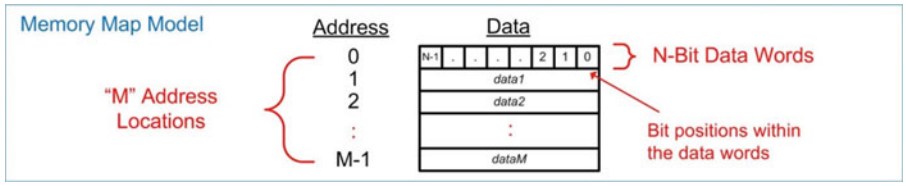


Fig. 10.1
Memory map model

10.1.2 Volatile Versus Nonvolatile Memory

Memory is classified into two categories depending on whether it can store information when power is removed or not. The term **nonvolatile** is used to describe memory that *holds* information when the power is removed, while the term **volatile** is used to describe memory that loses its information when power is removed. Historically, volatile memory is able to run at faster speeds compared to nonvolatile memory, so it is used as the primary storage mechanism while a digital system is running. Nonvolatile memory is necessary in order to hold critical operation information for a digital system such as start-up instructions, operations systems, and applications.

10.1.3 Read-Only Versus Read/Write Memory

Memory can also be classified into two categories with respect to how data are accessed. **Read-Only Memory (ROM)** is a device that cannot be written to during normal operation. This type of memory is useful for holding critical system information or programs that should not be altered while the system is running. **Read/Write** memory refers to memory that can be read and written to during normal operation and is used to hold temporary data and variables.

10.1.4 Random Access Versus Sequential Access

Random Access Memory (RAM) describes memory in which any location in the system can be accessed at any time. The opposite of this is **sequential access** memory, in which not all address locations are immediately available. An example of a sequential access memory system is a tape drive. In order to access the desired address in this system, the tape spool must be spun until the address is in a position that can be observed. Most semiconductor memory in modern systems is random access. The terms RAM and ROM have been adopted, somewhat inaccurately, to also describe groups of memory with particular behavior. While the term ROM technically describes a system that cannot be written to, it has taken on the additional association of being the term to describe nonvolatile memory. While the term RAM technically describes how data are accessed, it has taken on the additional association of being the term to describe volatile memory. When describing modern memory systems, the terms RAM and ROM are used most commonly to describe the characteristics of the memory being used; however, modern memory systems can be both read/write and nonvolatile, and the majority of memory is random access.

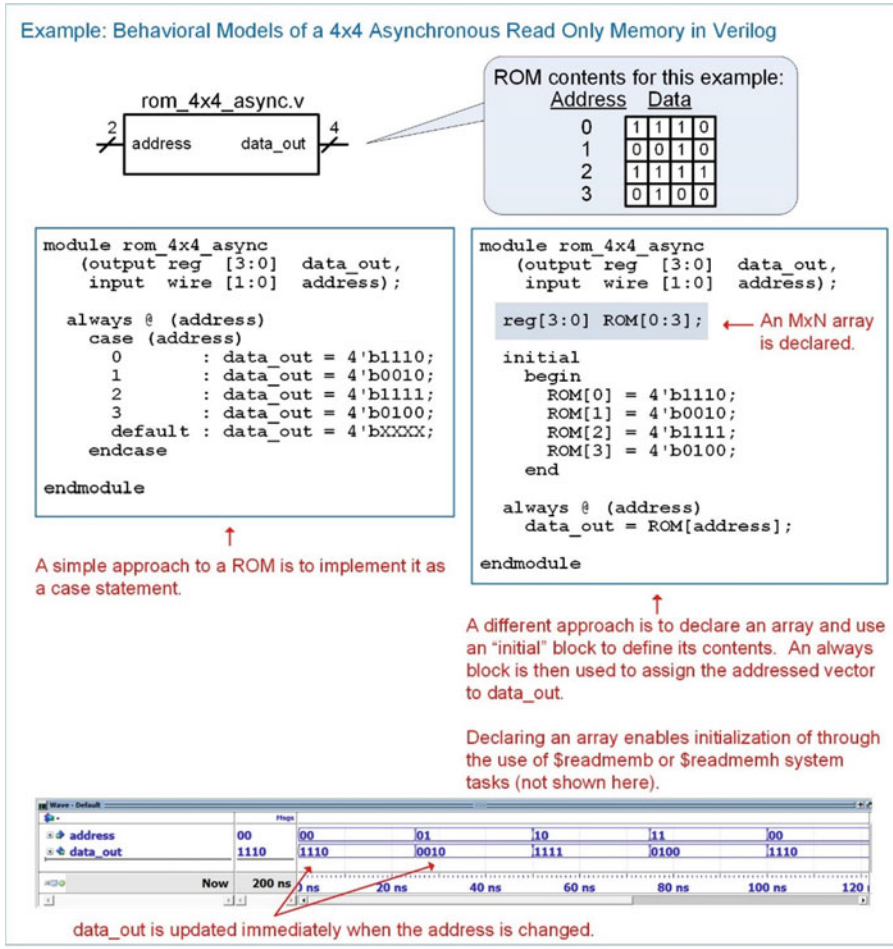
Concept Check

CC10.1 An 8-bit wide memory has eight address lines. What is its capacity in bits?

A) 64 B) 256 C) 1064 D) 2048

10.2 Modeling Read-Only Memory

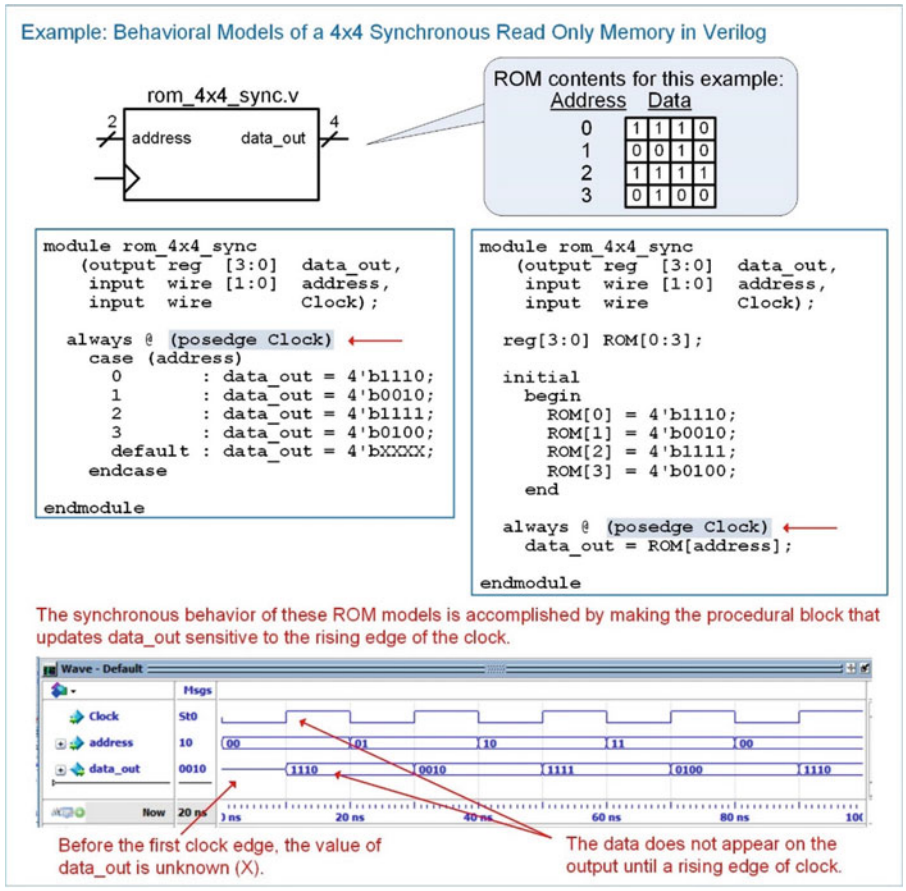
A read-only memory in Verilog can be defined in two ways. The first is to simply use a case statement to define the contents of each location in memory based on the incoming address. A second approach is to declare an array and then initialize its contents. When using an array, a separate procedural block handles assigning the contents of the array to the output based on the incoming address. The array can be initialized using either an initial block or through the file I/O system tasks $readmemb() or $readmemh(). Example 10.1 shows two approaches for modeling a 4 × 4 ROM memory. In this example the memory is asynchronous, meaning that as soon as the address changes, the data from the ROM will appear immediately. To model this asynchronous behavior, the procedural blocks are sensitive to the incoming address. In the simulation, each possible address is provided (i.e., "00," "01," "10," and "11") to verify that the ROM was initialized correctly.



Example 10.1
Behavioral models of a 4 × 4 asynchronous read-only memory in Verilog

A synchronous ROM can be created in a similar manner as in the asynchronous approach. The only difference is that in a synchronous ROM, a clock edge is used to trigger the procedural block that updates data_out. A sensitivity list is used that contains the clock to trigger the assignment. Example 10.2 shows

two Verilog models for a synchronous ROM. Notice that prior to the first clock edge, the simulator does not know what to assign to data_out so it lists the value as unknown (X).



Example 10.2
Behavioral models of a 4 × 4 synchronous read-only memory in Verilog

Concept Check

CC10.2 Explain the advantage of modeling memory in Verilog without going into the details of the storage cell operation.

A) It allows the details of the storage cell to be abstracted from the functional operation of the memory system.

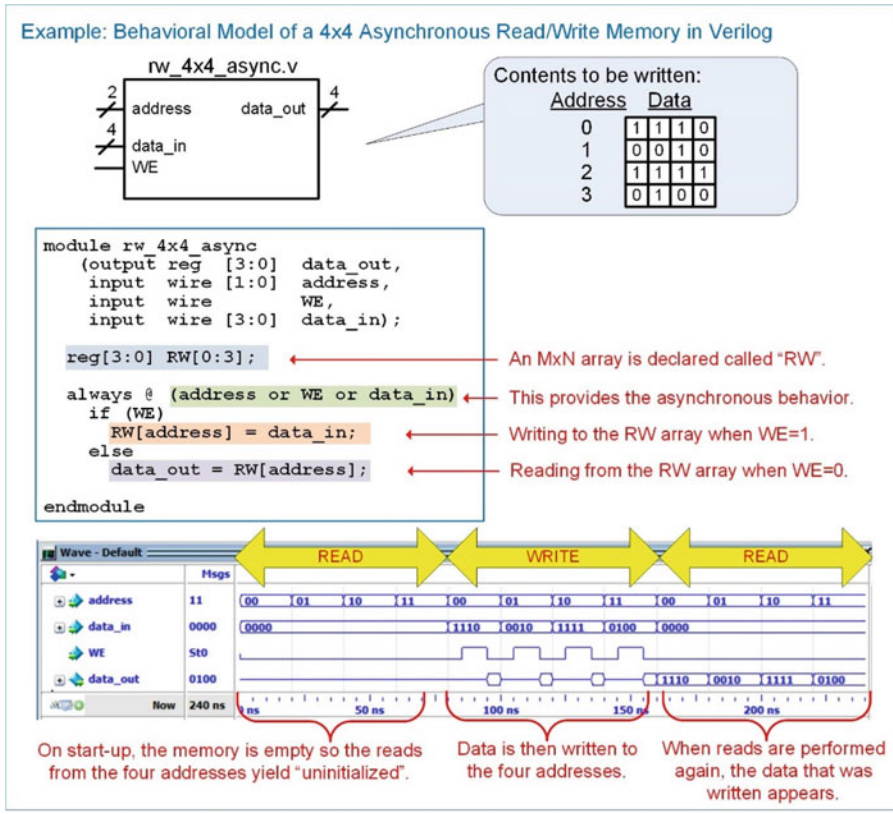
B) It is too difficult to model the analog behavior of the storage cell.

C) There are too many cells to model so the simulation would take too long.

D) It lets both ROM and R/W memory to be modeled in a similar manner.

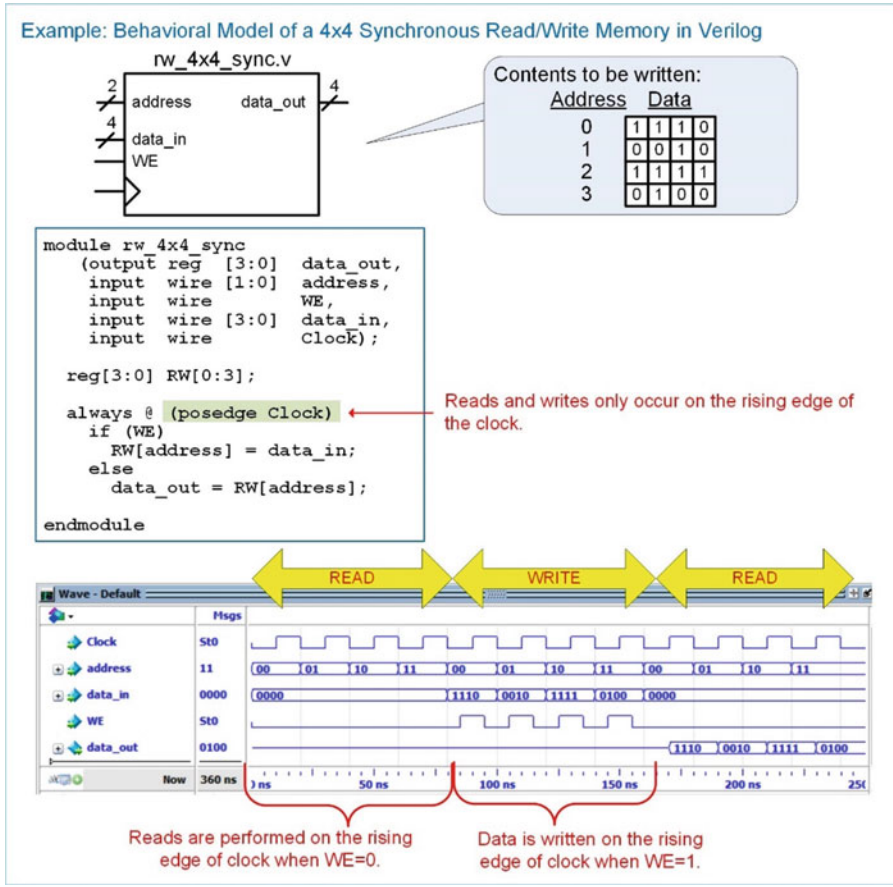
10.3 Modeling Read/Write Memory

In a simple read/write memory model, there is an output port that provides data when reading (data_out) and an input port that receives data when writing (data_in). Within the module, an array signal is declared with elements of type reg. To write to the array, signal assignment is made from the data_in port to the element within the array corresponding to the incoming address. To read from the array, the data_out port is assigned the element within the array corresponding to the incoming address. A *write enable* (WE) signal tells the system when to write to the array (WE = 1) or when to read from the array (WE = 0). In an asynchronous R/W memory, data are immediately written to the array when WE = 1 and data are immediately read from the array when WE = 0. This is modeled using a procedural block with a sensitivity list containing every input to the system. Example 10.3 shows an asynchronous R/W 4 × 4 memory system and functional simulation results. In the simulation, each address is initially read from to verify that it does not contain data. The data_out port produces unknown (X) for the initial set of read operations. Each address in the array is then written to. Finally, the array is read from verifying that the data that were written can be successfully retrieved.



Example 10.3
Behavioral model of a 4 × 4 asynchronous read/write memory in Verilog

A synchronous read/write memory is made in a similar manner with the exception that a clock is used to trigger the procedural block managing the signal assignments. In this case, the WE signal acts as a synchronous control signal indicating whether assignments are read from or written to the RW array. Example 10.4 shows the Verilog model for a synchronous read/write memory and the simulation waveform showing both read and write cycles.



Example 10.4
Behavioral model of a 4 × 4 synchronous read/write memory in Verilog

Concept Check

CC10.3 Does modeling the R/W memory as an uninitialized array accurately describe the behavior of real R/W memory technology?

A) Yes. Read/Write memory is not initialized upon power-up.

B) No. Read/Write memory should be initialized to all zeros to model the reset behavior found in memory.

Summary

- The term memory refers to arrays of storage. The technology used in memory is typically optimized for storage density at the expense of control capability. This is different from a D-flip-flop, which is optimized for complete control at the bit level.
- A memory device always contains an address bus input. The number of bits in the address bus dictate how many storage locations can be accessed. An n-bit address bus can access 2^n (or M) storage locations.

- The width of each storage location (N) allows the density of the memory array to be increased by reading and writing vectors of data instead of individual bits.
- A memory map is a graphical depiction of a memory array. A memory map is useful to give an overview of the capacity of the array and how different address ranges of the array are used.
- A read is an operation in which data are retrieved from memory. A write is an operation in which data are stored to memory.
- An asynchronous memory array responds immediately to its control inputs. A synchronous memory array only responds on the triggering edge of clock.
- Volatile memory will lose its data when the power is removed. Nonvolatile memory will retain its data when the power is removed.
- Read-Only Memory (ROM) is a memory type that cannot be written to during normal operation. Read/Write (R/W) memory is a memory type that can be written to during normal operation. Both ROM and R/W memory can be read from during normal operation.
- Random Access Memory (RAM) is a memory type in which any location in memory can be accessed at any time. In Sequential Access Memory, the data can only be retrieved in a linear sequence. This means that in sequential memory, the data cannot be accessed arbitrarily.
- ROM Memory can be modeled in Verilog using a case statement or an array data type consisting of elements of type reg that are initialized with an initial procedural block.
- R/W Memory can be modeled in Verilog using an array data type consisting of elements of type reg that are uninitialized.

Exercise Problems

Section 10.1: Memory Architecture and Terminology

10.1.1 For a 512k × 32 memory system, how many unique address locations are there? Give the exact number.

10.1.2 For a 512k × 32 memory system, what is the data width at each address location?

10.1.3 For a 512k × 32 memory system, what is the *capacity* in bits?

10.1.4 For a 512k × 32-bit memory system, what is the *capacity* in bytes?

10.1.5 For a 512k × 32 memory system, how wide does the incoming address bus need to be in order to access every unique address location?

Section 10.2: Modeling Read-Only Memory

10.2.1 Design a Verilog model for the 16 × 8, asynchronous, read-only memory system shown in Fig. 10.2. The system should contain the information provided in the memory map. Create a test bench to simulate your model by reading from each of the 16 unique addresses and observing data_out to verify it contains the information in the memory map.

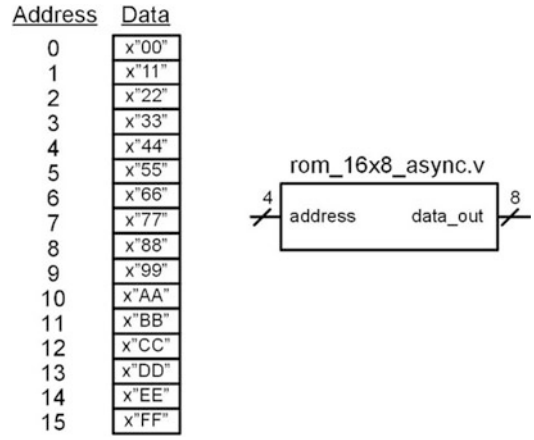


Fig. 10.2
16 × 8 asynchronous ROM block diagram

10.2.2 Design a Verilog model for the 16 × 8, synchronous, read-only memory system shown in Fig. 10.3. The system should contain the information provided in the memory map. Create a test bench to simulate your model by reading from each of the 16 unique addresses and observing data_out to verify it contains the information in the memory map.

Address	Data
0	x"FF"
1	x"EE"
2	x"DD"
3	x"CC"
4	x"BB"
5	x"AA"
6	x"99"
7	x"88"
8	x"77"
9	x"66"
10	x"55"
11	x"44"
12	x"33"
13	x"22"
14	x"11"
15	x"00"

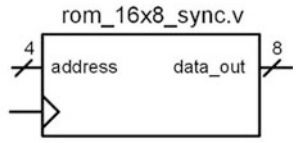


Fig. 10.3
16 × 8 synchronous ROM block diagram

Section 10.3: Modeling Read/Write Memory

10.3.1 Design a Verilog model for the 16 × 8, asynchronous, read/write memory system shown in Fig. 10.4. Create a test bench to simulate your model. Your test bench should first read from all of the address locations to verify they are uninitialized. Next, your test bench should write unique information to each of the address locations. Finally, your test bench should read from each address location to verify that the information that was written was stored and can be successfully retrieved.

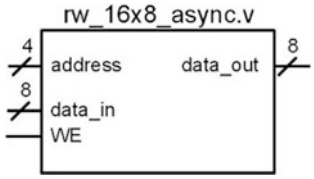


Fig. 10.4
16 × 8 asynchronous R/W block diagram

10.3.2 Design a Verilog model for the 16 × 8, synchronous, read/write memory system shown in Fig. 10.5. Create a test bench to simulate your model. Your test bench should first read from all of the address locations to verify they are uninitialized. Next, your test bench should write unique information to each of the address locations. Finally, your test bench should read from each address location to verify that the information that was written was stored and can be successfully retrieved.

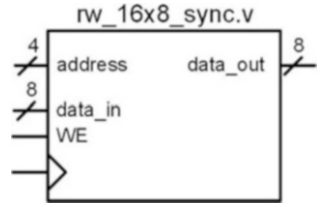


Fig. 10.5
16 × 8 synchronous R/W block diagram

Chapter 11: Computer System Design

This chapter presents the design of a simple computer system that will illustrate the use of many of the Verilog modeling techniques covered in this book. The goal of this chapter is not to provide an in-depth coverage of modern computer architecture, but rather to present a simple operational computer that can be implemented in Verilog to show how to use many of the modeling techniques covered thus far. The chapter begins with some architectural definitions so that consistent terminology can be used throughout the computer design example.

Learning Outcomes—After completing this chapter, you will be able to:

11.1 Describe the basic components and operation of computer hardware
11.2 Describe the basic components and operation of computer software
11.3 Design a fully operational computer system using Verilog

11.1 Computer Hardware

A computer accomplishes tasks through an architecture that uses both *hardware* and *software*. The hardware in a computer consists of many of the elements that we have covered so far. These include registers, arithmetic and logic circuits, finite state machines, and memory. What makes a computer so useful is that the hardware is designed to accomplish a predetermined set of **instructions**. These instructions are relatively simple, such as moving data between memory and a register or performing arithmetic on two numbers. The instructions are comprised of binary codes that are stored in a memory device and represent the sequence of operations that the hardware will perform to accomplish a task. This sequence of instructions is called a computer **program**. What makes this architecture so useful is that the preexisting hardware can be *programmed* to perform an almost unlimited number of tasks by simply defining the sequence of instructions to be executed. The process of designing the sequence of instructions, or program, is called *software development* or *software engineering*.

The idea of a general-purpose computing machine dates back to the nineteenth century. The first computing machines were implemented with mechanical systems and were typically analog in nature. As technology advanced, computer hardware evolved from electromechanical switches to vacuum tubes and ultimately to integrated circuits. These newer technologies enabled switching circuits and provided the capability to build binary computers. Today's computers are built exclusively with semiconductor materials and integrated circuit technology. The term *microcomputer* is used to describe a computer that has its processing hardware implemented with integrated circuitry. Nearly all modern computers are binary. Binary computers are designed to operate on a fixed set of bits. For example, an 8-bit computer would perform operations on 8-bits at a time. This means it moves data between registers and memory and performs arithmetic and logic operations in groups of 8-bits.

Computer hardware refers to all of the physical components within the system. This hardware includes all circuit components in a computer such as the memory devices, registers, and finite state machines. Figure 11.1 shows a block diagram of the basic hardware components in a computer.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_11

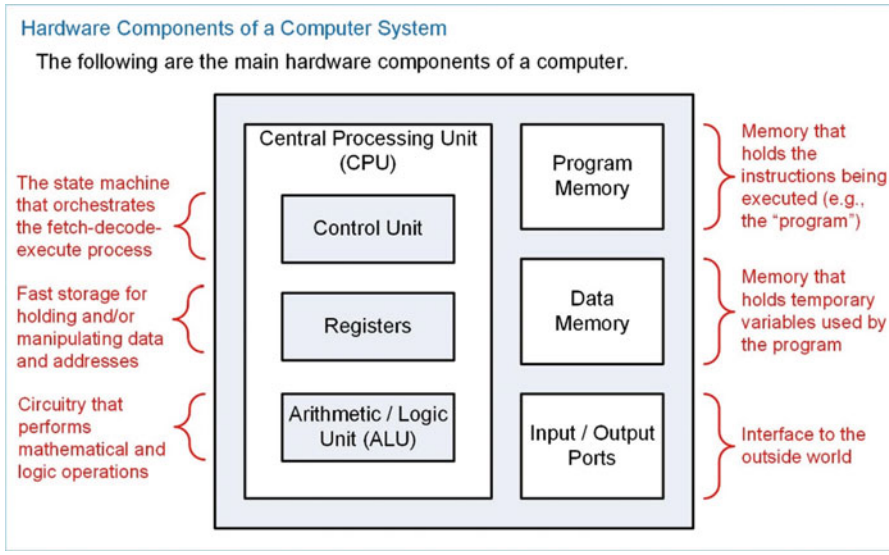


Fig. 11.1
Hardware components of a computer system

11.1.1 Program Memory

The instructions that are executed by a computer are held in *program memory*. Program memory is treated as read only during execution in order to prevent the instructions from being overwritten by the computer. Programs are typically held in nonvolatile memory so that the computer system does not lose its program when power is removed. Modern computers will often copy a program from nonvolatile memory (e.g., a hard disk drive) to volatile memory (i.e., SRAM or DRAM) after startup in order to speed up instruction execution as volatile memory is often a faster technology.

11.1.2 Data Memory

Computers also require *data memory*, which can be written to and read from during normal operation. This memory is used to hold temporary variables that are created by the software program. This memory expands the capability of the computer system by allowing large amounts of information to be created and stored by the program. Additionally, computations can be performed that are larger than the width of the computer system by holding interim portions of the calculation (e.g., performing a 128-bit addition on a 32-bit computer). Data memory is typically implemented with volatile memory as it is often faster than read-only memory technology.

11.1.3 Input/Output Ports

The term *port* is used to describe the mechanism to get information from the output world into or out of the computer. Ports can be input, output, or bidirectional. I/O ports can be designed to pass information in a serial or parallel format.

11.1.4 Central Processing Unit

The *central processing unit* (CPU) is considered the *brains* of the computer. The CPU handles reading instructions from memory, decoding them to understand which instruction is being performed, and executing the necessary steps to complete the instruction. The CPU also contains a set of registers

that are used for general-purpose data storage, operational information, and system status. Finally, the CPU contains circuitry to perform arithmetic and logic operations on data.

11.1.4.1 Control Unit

The *control unit* is a finite state machine that controls the operation of the computer. This FSM has states that perform fetching the instruction (i.e., reading it from program memory), decoding the instruction, and executing the appropriate steps to accomplish the instruction. This process is known as *fetch, decode, and execute* and is repeated each time an instruction is performed by the CPU. As the control unit state machine traverses through its states, it asserts control signals that move and manipulate data in order to achieve the desired functionality of the instruction.

11.1.4.2 Data Path—Registers

The CPU groups its registers and ALU into a subsystem called the *data path*. The data path refers to the fast storage and data manipulations within the CPU. All of these operations are initiated and managed by the control unit state machine. The CPU contains a variety of registers that are necessary to execute instructions and hold status information about the system. Basic computers have the following registers in their CPU:

- **Instruction Register (IR)**—The instruction register holds the current binary code of the instruction being executed. This code is read from program memory as the first part of instruction execution. The IR is used by the control unit to decide which states in its FSM to traverse in order to execute the instruction.
- **Memory Address Register (MAR)**—The memory address register is used to hold the current address being used to access memory. The MAR can be loaded with addresses in order to fetch instructions from program memory or with addresses to access data memory and/or I/O ports.
- **Program Counter (PC)**—The program counter holds the address of the current instruction being executed in program memory. The program counter will increment sequentially through the program memory reading instructions until a dedicated instruction is used to set it to a new location.
- **General-Purpose Registers**—These registers are available for temporary storage by the program. Instructions exist to move information from memory into these registers and to move information from these registers into memory. Instructions also exist to perform arithmetic and logic operations on the information held in these registers.
- **Condition Code Register (CCR)**—The condition code register holds status flags that provide information about the arithmetic and logic operations performed in the CPU. The most common flags are *negative* (N), zero (Z), two's complement overflow (V), and carry (C). This register can also contain flags that indicate the status of the computer, such as if an interrupt has occurred or if the computer has been put into a low-power mode.

11.1.4.3 Data Path—Arithmetic Logic Unit (ALU)

The *arithmetic logic unit* is the system that performs all mathematical (i.e., addition, subtraction, multiplication, and division) and logic operations (i.e., and, or, not, shifts, etc.). This system operates on data being held in CPU registers. The ALU has a unique symbol associated with it to distinguish it from other functional units in the CPU.

Figure 11.2 shows the typical organization of a CPU. The registers and ALU are grouped into the data path. In this example, the computer system has two general-purpose registers called A and B. This CPU organization will be used throughout this chapter to illustrate the detailed execution of instructions.

Typical CPU Organization

A CPU is functionally organized into a control unit and a data path. The control unit contains the FSM to orchestrate the fetch-decode-execute process. The registers and ALU are grouped into a unit called the data path. The control unit sends control signals to the data path to move and manipulate data. The control unit uses status signals from the data path to decide which states to traverse in its FSM.

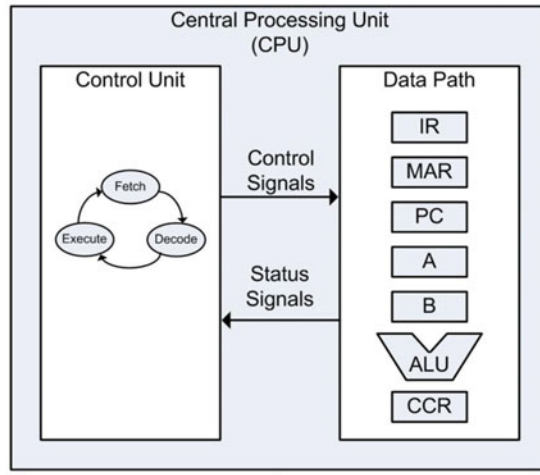


Fig. 11.2
Typical CPU organization

11.1.5 A Memory-Mapped System

A common way to simplify moving data in or out of the CPU is to assign a unique address to all hardware components in the memory system. Each input/output port and each location in both program and data memory are assigned a unique address. This allows the CPU to access everything in the memory system with a dedicated address. This reduces the number of lines that must pass into the CPU. A *bus system* facilitates transferring information within the computer system. An address bus is driven by the CPU to identify which location in the memory system is being accessed. A data bus is used to transfer information to/from the CPU and the memory system. Finally, a control bus is used to provide other required information about the transactions such as *read* or *write* lines. Figure 11.3 shows the computer hardware in a memory-mapped architecture.

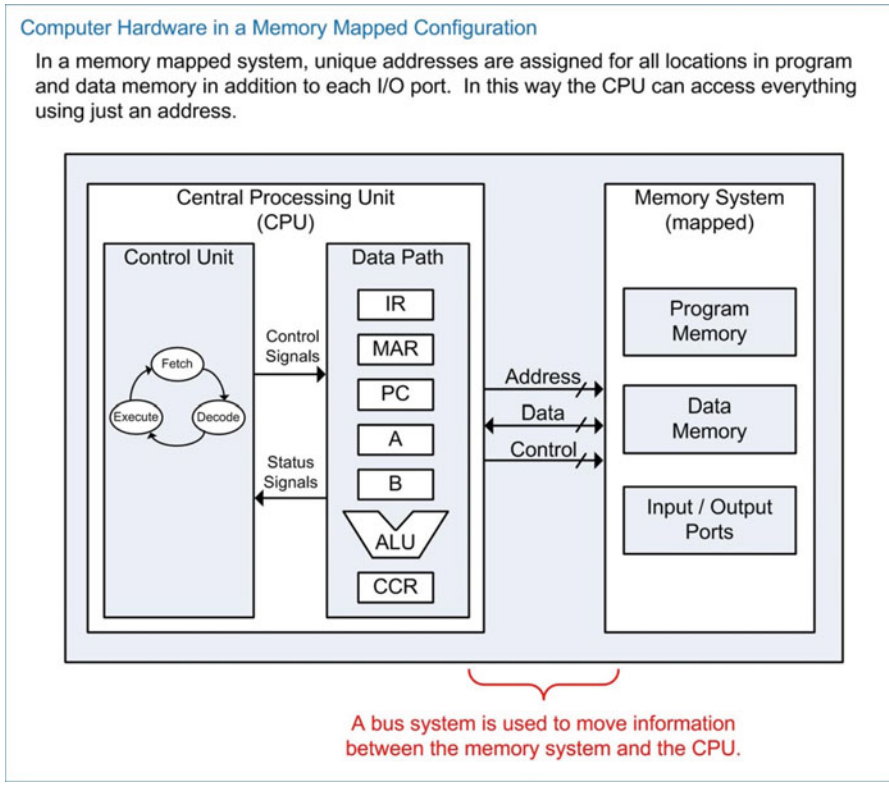
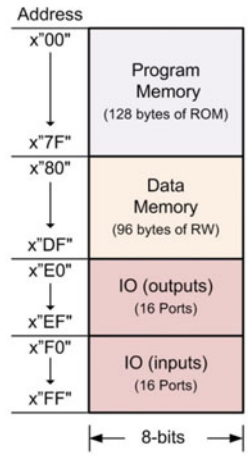


Fig. 11.3
Computer hardware in a memory-mapped configuration

To help visualize how the memory addresses are assigned, a *memory map* is used. This is a graphical depiction of the memory system. In the memory map, the ranges of addresses are provided for each of the main subsections of memory. This gives the programmer a quick overview of the available resources in the computer system. Example 11.1 shows a representative memory map for a computer system with an address bus with a width of 8-bits. This address bus can provide 256 unique locations. For this example, the memory system is also 8-bits wide, thus the entire memory system is 256 $\times$ 8 in size. In this example, 128 bytes are allocated for program memory; 96 bytes are allocated for data memory; 16 bytes are allocated for output ports; and 16 bytes are allocated for input ports.

Example: Memory Map for a 256x8 Memory System

The following is a memory map for an example 8-bit computer system.



Example 11.1
Memory map for a 256 × 8 memory system

Concept Check

CC11.1 Is the hardware of a computer programmed in a similar way to a programmable logic device?

A) Yes. The control unit is reconfigured to produce the correct logic for each unique instruction just like a logic element in an FPGA is reconfigured to produce the desired logic expression.

B) No. The instruction code from program memory simply tells the state machine in the control unit which path to traverse in order to accomplish the desired task.

11.2 Computer Software

Computer software refers to the instructions that the computer can execute and how they are designed to accomplish various tasks. The specific group of instructions that a computer can execute is known as its **instruction set**. The instruction set of a computer needs to be defined first before the computer hardware can be implemented. Some computer systems have a very small number of instructions in order to reduce the physical size of the circuitry needed in the CPU. This allows the CPU to execute the instructions very quickly but requires a large number of operations to accomplish a given task. This architectural approach is called a **reduced instruction set computer** (RISC). The alternative to this approach is to make an instruction set with a large number of dedicated instructions that can accomplish a given task in fewer CPU operations. The drawback of this approach is that the physical size of the CPU must be larger in order to accommodate the various instructions. This architectural approach is called a **complex instruction set computer** (CISC).

11.2.1 Opcodes and Operands

A computer instruction consists of two fields, an *opcode* and an *operand*. The opcode is a unique binary code given to each instruction in the set. The CPU decodes the opcode in order to know which instruction is being executed and then takes the appropriate steps to complete the instruction. Each opcode is assigned a **mnemonic**, which is a descriptive name for the opcode that can be used when discussing the instruction functionally. An operand is additional information for the instruction that may be required. An instruction may have any number of operands including zero. Figure 11.4 shows an example of how the instruction opcodes and operands are placed into program memory.

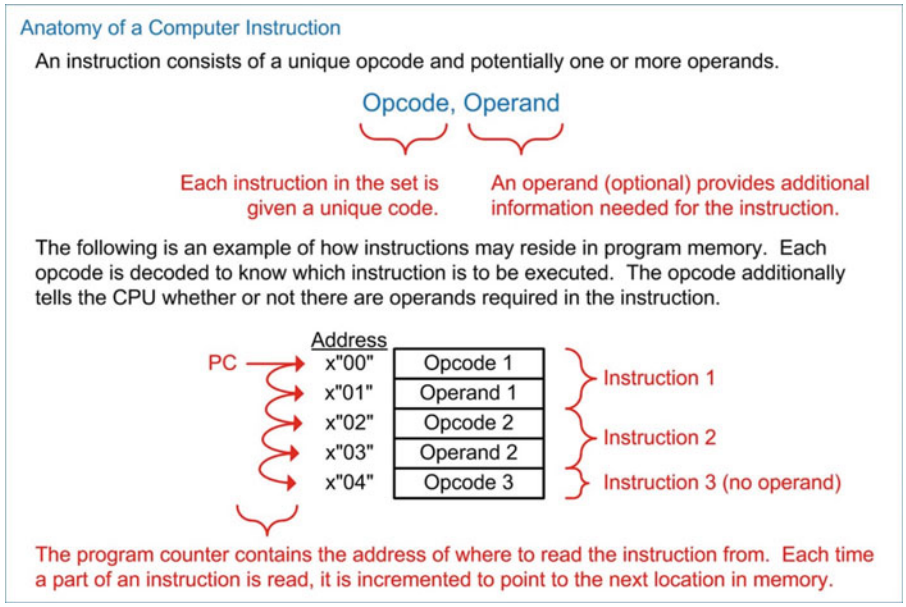


Fig. 11.4
Anatomy of a computer instruction

11.2.2 Addressing Modes

An *addressing mode* describes the way in which the operand of an instruction is used. While modern computer systems may contain numerous addressing modes with varying complexities, we will focus on just a subset of basic addressing modes. These modes are immediate, direct, inherent, and indexed.

11.2.2.1 Immediate Addressing (IMM)

Immediate addressing is when the operand of an instruction *is* the information to be used by the instruction. For example, if an instruction existed to put a constant into a register within the CPU using immediate addressing, the operand would *be* the constant. When the CPU reads the operand, it simply inserts the contents into the CPU register and the instruction is complete.

11.2.2.2 Direct Addressing (DIR)

Direct addressing is when the operand of an instruction contains the *address* of where the information to be used is located. For example, if an instruction existed to put a constant into a register within the CPU using direct addressing, the operand would contain the address of *where* the constant was located

in memory. When the CPU reads the operand, it puts this value out on the address bus and performs an additional read to retrieve the contents located at that address. The value read is then put into the CPU register and the instruction is complete.

11.2.2.3 Inherent Addressing (INH)

Inherent addressing refers to an instruction that does not require an operand because the opcode itself contains all of the necessary information for the instruction to complete. This type of addressing is used on instructions that perform manipulations on data held in CPU registers without the need to access the memory system. For example, if an instruction existed to increment the contents of a register (A), then once the opcode is read by the CPU, it knows everything it needs to know in order to accomplish the task. The CPU simply asserts a series of control signals in order to increment the contents of A and then the instruction is complete. Notice that no operand is needed for this task. Instead, the location of the register to be manipulated (i.e., A) is inherent within the opcode.

11.2.3 Classes of Instructions

There are three general classes of instructions: (1) loads and stores; (2) data manipulations; and (3) branches. To illustrate how these instructions are executed, examples will be given based on the computer architecture shown in Fig. 11.3.

11.2.3.1 Loads and Stores

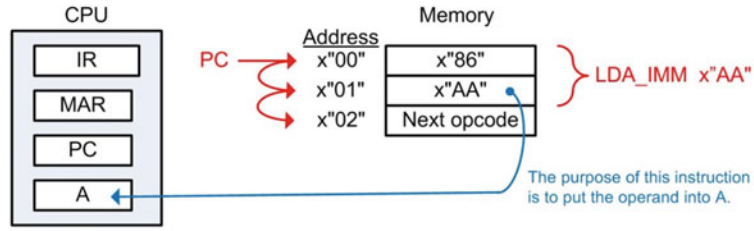
This class of instructions accomplishes moving information between the CPU and memory. A **load** is an instruction that moves information from memory *into* a CPU register. When a load instruction uses immediate addressing, the operand of the instruction *is* the data to be loaded into the CPU register. As an example, let us look at an instruction to load the general-purpose register A using immediate addressing. Let us say that the opcode of the instruction is x”86”, has a mnemonic LDA_IMM, and is inserted into program memory starting at x”00”. Example 11.2 shows the steps involved in executing the LDA_IMM instruction.

Example: Execution of an Instruction to "Load Register A Using Immediate Addressing"

A load instruction using immediate addressing will put the value of the operand into a CPU register. Let's create a program that will load register A in the CPU with the value x"AA". The program is as follows:

Using Mnemonics		Using Hex Values
LDA_IMM x"AA"	or	x"86" x"AA"

When the opcode and operand are put into program memory at x"00", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"00", meaning that this address is the location of the first instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"86" and the PC holds x"01".

Step 2 – Decode the instruction

The CPU decodes x"86" and understands that it is a "load A with immediate addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"01") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is placed into register A. After this step, A=x"AA". Also in this step, the PC is incremented to point to the next location in memory (x"02"), which holds the opcode of the next instruction to be executed.

Example 11.2
Execution of an instruction to "load register A using immediate addressing"

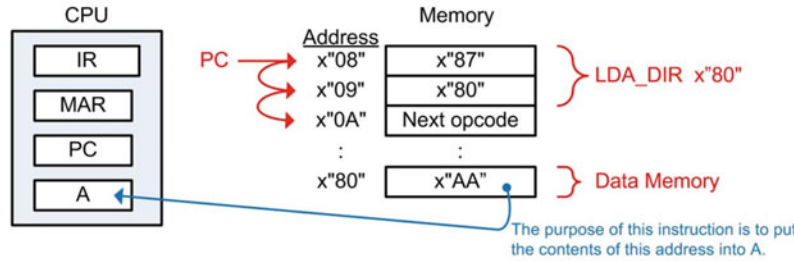
Now let us look at a load instruction using direct addressing. In direct addressing, the operand of the instruction is the *address* of where the data to be loaded resides. As an example, let us look at an instruction to load the general-purpose register A. Let us say that the opcode of the instruction is x"87", has a mnemonic LDA_DIR, and is inserted into program memory starting at x"08". The value to be loaded into A resides at address x"80", which has already been initialized with x"AA" before this instruction. Example 11.3 shows the steps involved in executing the LDA_DIR instruction.

Example: Execution of an Instruction to "Load Register A Using Direct Addressing"

A load instruction using direct addressing will put the value located at the address provided by the operand into a CPU register. Let's create a program that will load register A in the CPU with the contents located at address x"80", which has already been initialized to x"AA". The program is as follows:

Using Mnemonics		Using Hex Values
LDA_DIR x"80"	or	x"87" x"80"

When the opcode and operand are put into program memory at x"08", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"08", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"87" and the PC holds x"09".

Step 2 – Decode the instruction

The CPU decodes x"87" and understands that it is a "load A with direct addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"09") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address that contains the value to be put into A. The operand is immediately put on the address bus using the MAR and another read is performed. The value read from address x"80" is placed into register A. After this step, A=x"AA". Also in this step, the PC is incremented to point to the next location in memory (x"0A"), which holds the opcode of the next instruction to be executed.

Example 11.3
Execution of an instruction to "load register A using direct addressing"

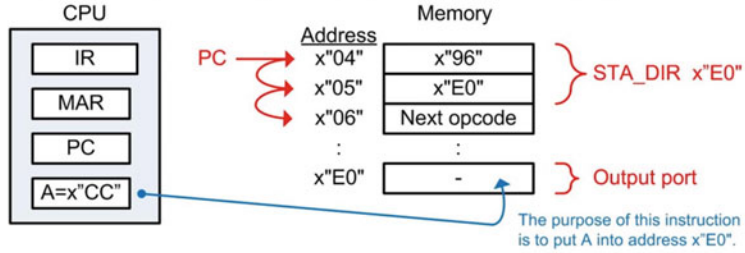
A **store** is an instruction that moves information from a CPU register *into* memory. The operand of a store instruction indicates the address of where the contents of the CPU register will be written. As an example, let us look at an instruction to store the general-purpose register A into memory address x"E0". Let us say that the opcode of the instruction is x"96", has a mnemonic STA_DIR, and is inserted into program memory starting at x"04". The initial value of A is x"CC" before the instruction is executed. Example 11.4 shows the steps involved in executing the STA_DIR instruction.

Example: Execution of an Instruction to "Store Register A Using Direct Addressing"

A store instruction using direct addressing will put the value held in a CPU register into memory at the address provided by the operand. Let's create a program that will store register A in the CPU to address location x"E0". We can assume A holds x"CC" prior to this instruction. The program is as follows:

Using Mnemonics		Using Hex Values
STA_DIR x"E0"	or	x"96" x"E0"

When the opcode and operand are put into program memory at x"04", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"04", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the IR holds x"96" and the PC holds x"05".

Step 2 – Decode the instruction

The CPU decodes x"96" and understands that it is a "store A with direct addressing". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"05") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address of where A will be written. The operand is immediately put on the address bus using the MAR, A is put on the data bus, and a write is performed. After this step, location x"E0" in memory contains x"CC". Also in this step, the PC is incremented to point to the next location in memory (x"06"), which holds the opcode of the next instruction to be executed. The write did not effect register A so it still contains x"CC" after the instruction completes.

Example 11.4
Execution of an instruction to "store register A using direct addressing"

11.2.3.2 Data Manipulations

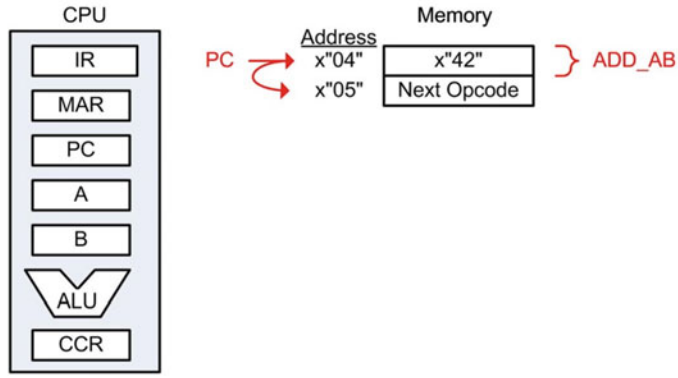
This class of instructions refers to ALU operations. These operations act on data that reside in the CPU registers. These instructions include arithmetic, logic operators, shifts and rotates, and tests and compares. Data manipulation instructions typically use inherent addressing because the operations are conducted on the contents of CPU registers and do not require additional memory access. As an example, let us look at an instruction to perform addition on registers A and B. The sum will be placed back in A. Let us say that the opcode of the instruction is x"42", has a mnemonic ADD_AB, and is inserted into program memory starting at x"04". Example 11.5 shows the steps involved in executing the ADD_AB instruction.

Example: Execution of an Instruction to "Add Registers A and B"

This instruction adds A and B and puts the sum back into A (A = A+B). This instruction does not require an operand because the inputs and output of the operation reside completely within the CPU. This type of instruction uses inherent addressing, meaning that the location of the information impacted is inherent in the opcode. Let's create a program to perform this addition. The program is as follows:

Using Mnemonics		Using Hex Values
ADD_AB	or	x"42"

When the opcode is put into program memory at x"04", it looks like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"04", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"05" and the IR holds x"42".

Step 2 – Decode the instruction

The CPU decodes x"42" and understands that it is an "Add A and B". It also knows that there is no operand associated with this instruction.

Step 3 – Execute the instruction

The CPU asserts the necessary control signals to route A and B to the ALU, performs the addition, and places the sum back into A. The CCR is also updated to provide additional status information about the operation.

Example 11.5
Execution of an instruction to "add registers A and B"

11.2.3.3 Branches

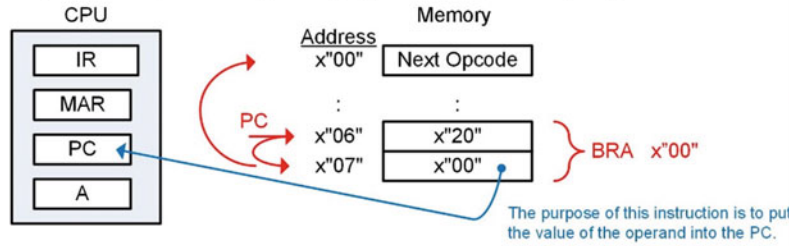
In the previous examples, the program counter was always incremented to point to the address of the next instruction in program memory. This behavior only supports a linear execution of instructions. To provide the ability to specifically set the value of the program counter, instructions called *branches* are used. There are two types of branches: **unconditional** and **conditional**. In an unconditional branch, the program counter is always loaded with the value provided in the operand. As an example, let us look at an instruction to *branch always* to a specific address. This allows the program to perform loops. Let us say that the opcode of the instruction is x"20", has a mnemonic BRA, and is inserted into program memory starting at x"06". Example 11.6 shows the steps involved in executing the BRA instruction.

Example: Execution of an Instruction to "Branch Always"

A *branch always* instruction will set the program counter to the value provided by the operand. Let's create a program that will set the program counter to x"00". The program is as follows:

Using Mnemonics		Using Hex Values
BRA x"00"	or	x"20" x"00"

When the opcode and operand are put into program memory at x"06", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"06", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"07" and the IR holds x"20".

Step 2 – Decode the instruction

The CPU decodes x"20" and understands that it is a "branch always". It also knows from the opcode that the instruction has an operand that exists at the next address location.

Step 3 – Execute the instruction

The CPU now needs to read the operand. It places the PC address (x"07") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is the address to load into the PC. The operand is latched into the PC and the instruction is complete. After this instruction, the PC=x"00" and the program will begin executing instructions at that address.

Example 11.6
Execution of an instruction to "branch always"

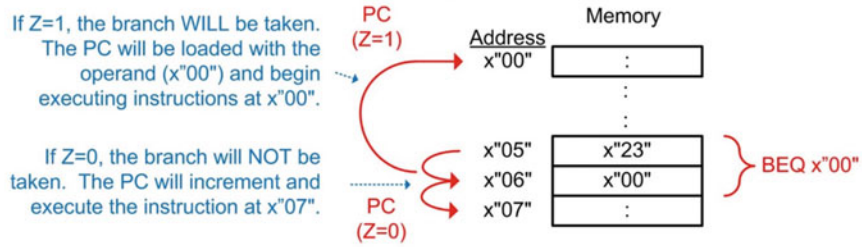
In a conditional branch, the program counter is only updated if a particular condition is true. The conditions come from the status flags in the condition code register (NZVC). This allows a program to selectively execute instructions based on the result of a prior operation. Let us look at an example instruction that will branch only if the Z flag is asserted. This instruction is called a *branch if equal to zero*. Let us say that the opcode of the instruction is x"23", has a mnemonic BEQ, and is inserted into program memory starting at x"05". Example 11.7 shows the steps involved in executing the BEQ instruction.

Example: Execution of an Instruction to "Branch if Equal to Zero"

This instruction will update the program counter with the address in the operand if the zero flag (Z) in the condition code register is asserted (Z=1). If Z=0, the program counter will simply increment to the next location in program memory. Let's look at how this program is executed. The instruction resides in program memory at addresses x"05" and x"06".

Using Mnemonics		Using Hex Values
BEQ x"00"	or	x"23" x"00"

When the opcode and operand are put into program memory at x"02", they look like this:



When the CPU begins executing the program, it will perform the following steps:

Step 1 – Fetch the opcode

The program counter begins at x"05", meaning that this address is the location of the instruction opcode. The PC address is put on the address bus using the MAR and a read is performed. The information read from memory (e.g., the opcode) is placed into the instruction register. The PC is then incremented to point to the next address in program memory. After this step, the PC holds x"06" and the IR holds x"23".

Step 2 – Decode the instruction

The CPU decodes x"23" and understands that it is a "branch if equal to zero". It also knows from the opcode that the instruction has an operand that exists at the next address location. The FSM now looks at the Z flag and decides which path in the FSM to take in order to execute the instruction properly.

Step 3 – Execute the instruction

Z=1 – The branch will be taken by loading the PC with the operand. It places the PC address (x"06") on the address bus using the MAR and a read is performed. The information read from memory (e.g., the operand) is then loaded into the PC. If this action is taken, the PC=x"00".

Z=0 – The branch will not be taken. Instead, the PC is simply incremented to point to the next location in memory, bypassing the operand. If this action is taken, the PC=x"07".

Example 11.7
Execution of an instruction to "branch if equal to zero"

Conditional branches allow computer programs to make *decisions* about which instructions to execute based on the results of previous instructions. This gives computers the ability to react to input signals or act based on the results of arithmetic or logic operations. Computer instruction sets typically contain conditional branches based on the NZVC flags in the condition code registers. The following instructions are a set of possible branches that could be created using the values of the NZVC flags.

- BMI—Branch if minus (N = 1)
- BPL—Branch if plus (N = 0)
- BEQ—Branch if equal to Zero (Z = 1)
- BNE—Branch if not equal to Zero (Z = 0)
- BVS—Branch if two's complement overflow occurred, or V is set (V = 1)

- BVC—Branch if two's complement overflow did not occur, or V is clear (V = 0)
- BCS—Branch if a carry occurred, or C is set (C = 1)
- BCC—Branch if a carry did not occur, or C is clear (C = 0)

Combinations of these flags can be used to create more conditional branches.

- BHI—Branch if higher (C = 1 and Z = 0)
- BLS—Branch if lower or the same (C = 0 and Z = 1)
- BGE—Branch if greater than or equal ((N = 0 and V = 0) or (N = 1 and V = 1)), only valid for signed numbers
- BLT—Branch if less than ((N = 1 and V = 0) or (N = 0 and V = 1)), only valid for signed numbers
- BGT—Branch if greater than ((N = 0 and V = 0 and Z = 0) or (N = 1 and V = 1 and Z = 0)), only valid for signed numbers
- BLE—Branch if less than or equal ((N = 1 and V = 0) or (N = 0 and V = 1) or (Z = 1)), only valid for signed numbers

CONCEPT CHECK

CC11.2 Software development consists of choosing which instructions, and in what order, will be executed to accomplish a certain task. The group of instructions is called the *program* and is inserted into program memory. Which of the following might a software developer care about?

A) Minimizing the number of instructions that need to be executed to accomplish the task in order to increase the computation rate.

B) Minimizing the number of registers used in the CPU to save power.

C) Minimizing the overall size of the program to reduce the amount of program memory needed.

D) Both A and C.

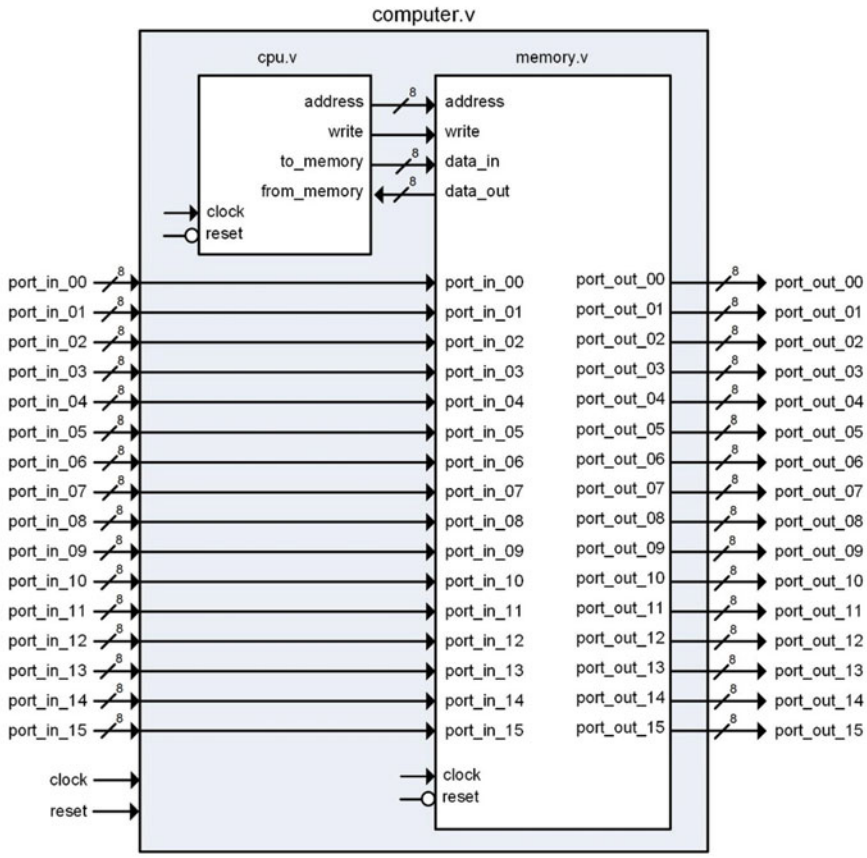
11.3 Computer Implementation—An 8-Bit Computer Example

11.3.1 Top-Level Block Diagram

Let us now look at the detailed implementation and instruction execution of a computer system. In order to illustrate the detailed operation, we will use a simple 8-bit computer system design. Example 11.8 shows the block diagram for the 8-bit computer system. This block diagram also contains the Verilog file and module names, which will be used when the behavioral model is implemented.

Example: Top Level Block Diagram for the 8-Bit Computer System

The following is the top level block diagram for our 8-bit computer system example.



Example 11.8
Top-level block diagram for the 8-bit computer system

We will use the memory map shown in Example 11.1 for our example computer system. This mapping provides 128 bytes of program memory, 96 bytes of data memory, 16× output ports, and 16× input ports. To simplify the operation of this example computer, the address bus is limited to 8-bits. This only provides 256 locations of memory access but allows an entire address to be loaded into the CPU as a single operand of an instruction.

11.3.2 Instruction Set Design

Example 11.9 shows a basic instruction set for our example computer system. This set provides a variety of loads and stores, data manipulations, and branch instructions that will allow the computer to be programmed to perform more complex tasks through software development. These instructions are sufficient to provide a baseline of functionality in order to get the computer system operational. Additional instructions can be added as desired to increase the complexity of the system.

Example: Instruction Set for the 8-Bit Computer System

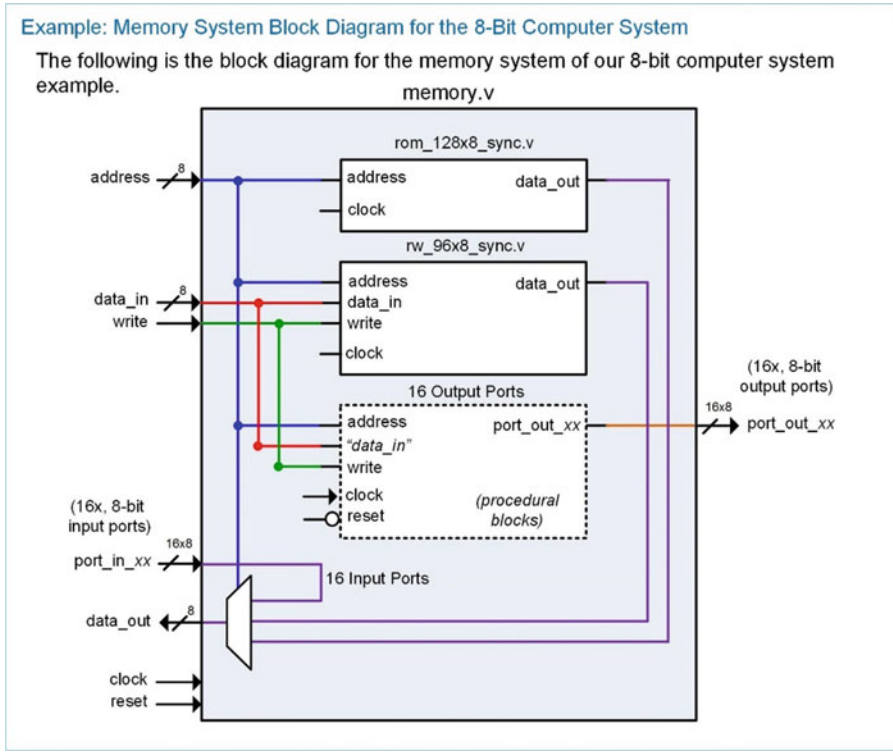
The following is a base set of instructions that the 8-bit computer system will be able to perform. Each instruction is given a descriptive mnemonic, which allows the system implementation and the programming to be more intuitive. Each instruction is also provided with a unique binary opcode. Some instructions have an operand, which provides additional information necessary for the instruction. If an instruction contains an operand, a description is provided as to how it is used (e.g., as data or as an address).

Mnemonic	Opcode, Operand	Description
"Loads and Stores"		
LDA_IMM	x"86", <*data*>	Load Register A using Immediate Addressing
LDA_DIR	x"87", <*addr*>	Load Register A using Direct Addressing
LDB_IMM	x"88", <*data*>	Load Register B with Immediate Addressing
LDB_DIR	x"89", <*addr*>	Load Register B with Direct Addressing
STA_DIR	x"96", <*addr*>	Store Register A to Memory using Direct Addressing
STB_DIR	x"97", <*addr*>	Store Register B to Memory using Direct Addressing
"Data Manipulations"		
ADD_AB	x"42"	A = A + B (plus)
SUB_AB	x"43"	A = A - B (minus)
AND_AB	x"44"	A = A · B (AND)
OR_AB	x"45"	A = A + B (OR)
INCA	x"46"	A = A + 1 (plus)
INCB	x"47"	B = B + 1 (plus)
DECA	x"48"	A = A - 1 (minus)
DECB	x"49"	B = B - 1 (minus)
"Branches"		
BRA	x"20", <*addr*>	Branch Always to Address Provided
BMI	x"21", <*addr*>	Branch to Address Provided if N=1
BPL	x"22", <*addr*>	Branch to Address Provided if N=0
BEQ	x"23", <*addr*>	Branch to Address Provided if Z=1
BNE	x"24", <*addr*>	Branch to Address Provided if Z=0
BVS	x"25", <*addr*>	Branch to Address Provided if V=1
BVC	x"26", <*addr*>	Branch to Address Provided if V=0
BCS	x"27", <*addr*>	Branch to Address Provided if C=1
BCC	x"28", <*addr*>	Branch to Address Provided if C=0

Example 11.9
Instruction set for the 8-bit computer system

11.3.3 Memory System Implementation

Let us now look at the memory system details. The memory system contains program memory, data memory, and input/output ports. Example 11.10 shows the block diagram of the memory system. The program and data memory will be implemented using lower-level components (rom_128x8_sync.v and rw_96x8_sync.v), while the input and output ports can be modeled using a combination of RTL blocks and combinational logic. The program and data memory subsystems contain dedicated circuitry to handle their addressing ranges. Each output port also contains dedicated circuitry to handle its unique address. A multiplexer is used to handle the signal routing back to the CPU based on the address provided.



Example 11.10
Memory system block diagram for the 8-bit computer system

11.3.3.1 Program Memory Implementation in Verilog

The program memory can be implemented in Verilog using the modeling techniques presented in Chap. 12. To make the Verilog more readable, the instruction mnemonics can be declared as parameters. This allows the mnemonic to be used when populating the program memory array. The following Verilog shows how the mnemonics for our basic instruction set can be defined as parameters.

```
parameter LDA_IMM = 8'h86;  //-- Load Register A with Immediate Addressing
parameter LDA_DIR = 8'h87;  //-- Load Register A with Direct Addressing
parameter LDB_IMM = 8'h88;  //-- Load Register B with Immediate Addressing
parameter LDB_DIR = 8'h89;  //-- Load Register B with Direct Addressing
parameter STA_DIR = 8'h96;  //-- Store Register A to memory (RAM or IO)
parameter STB_DIR = 8'h97;  //-- Store Register B to memory (RAM or IO)
parameter ADD_AB  = 8'h42;  //-- A <= A + B
parameter BRA     = 8'h20;  //-- Branch Always
parameter BEQ     = 8'h23;  //-- Branch if Z=1
```

Now the program memory can be declared as an array type with initial values to define the program. The following Verilog shows how to declare the program memory and an example program to perform a load, store, and a branch always. This program will continually write x"AA" to port_out_00.

```
reg[7:0] ROM[0:127];

initial
 begin
   ROM[0] = LDA_IMM;
   ROM[1] = 8'hAA;
   ROM[2] = STA_DIR;
   ROM[3] = 8'hE0;
   ROM[4] = BRA;
   ROM[5] = 8'h00;
 end
```

The address mapping for the program memory is handled in two ways. First, notice that the array type defined above uses indices from 0 to 127. This provides the appropriate addresses for each location in the memory. The second step is to create an internal enable line that will only allow assignments from ROM to data_out when a valid address is entered. Consider the following Verilog to create an internal enable (EN) that will only be asserted when the address falls within the valid program memory range of 0 to 127.

```
always @ (address)
  begin
    if ( (address >= 0) && (address <= 127) )
       EN = 1'b1;
    else
       EN = 1'b0;
  end
```

If this enable signal is not created, the simulation and synthesis will fail because data_out assignments will be attempted for addresses outside of the defined range of the ROM array. This enable line can now be used in the behavioral model for the ROM as follows:

```
always @ (posedge clock)
  begin
    if (EN)
       data_out = ROM[address];
  end
```

11.3.3.2 Data Memory Implementation in Verilog

The data memory is created using a similar strategy as the program memory. An array signal is declared with an address range corresponding to the memory map for the computer system (i.e., 128 to 223). An internal enable is again created that will prevent data_out assignments for addresses outside of this valid range. The following is the Verilog to declare the R/W memory array:

```
reg[7:0] RW[128:223];
```

The following is the Verilog to model the local enable and signal assignments for the R/W memory:

```
always @ (address)
   begin
     if ( (address >= 128) && (address <= 223) )
        EN = 1'b1;
     else
        EN = 1'b0;
   end
```

```
always @ (posedge clock)
  begin
    if (write && EN)
       RW[address] = data_in;
    else if (!write && EN)
       data_out = RW[address];
  end
```

11.3.3.3 Implementation of Output Ports in Verilog

Each output port in the computer system is assigned a unique address. Each output port also contains storage capability. This allows the CPU to update an output port by writing to its specific address. Once the CPU is done storing to the output port address and moves to the next instruction in the program, the output port holds its information until it is written to again. This behavior can be modeled using an RTL procedural block that uses the address bus and the write signal to create a synchronous enable condition. Each output port is modeled with its own block. The following Verilog shows how the output ports at x"E0" and x"E1" are modeled using address specific procedural blocks.

```
//-- port_out_00 (address E0)
   always @ (posedge clock or negedge reset)
    begin
      if (!reset)
         port_out_00 <= 8'h00;
      else
         if ((address == 8'hE0) && (write))
            port_out_00 <= data_in;
    end

//-- port_out_01 (address E1)
   always @ (posedge clock or negedge reset)
    begin
      if (!reset)
         port_out_01 <= 8'h00;
      else
         if ((address == 8'hE1) && (write))
            port_out_01 <= data_in;
    end

       :
    "the rest of the output port models go here..."
      :
```

11.3.3.4 Implementation of Input Ports in Verilog

The input ports do not contain storage but do require a mechanism to selectively route their information to the data_out port of the memory system. This is accomplished using the multiplexer shown in Example 11.10. The only functionality that is required for the input ports is connecting their ports to the multiplexer.

11.3.3.5 Memory data_out Bus Implementation in Verilog

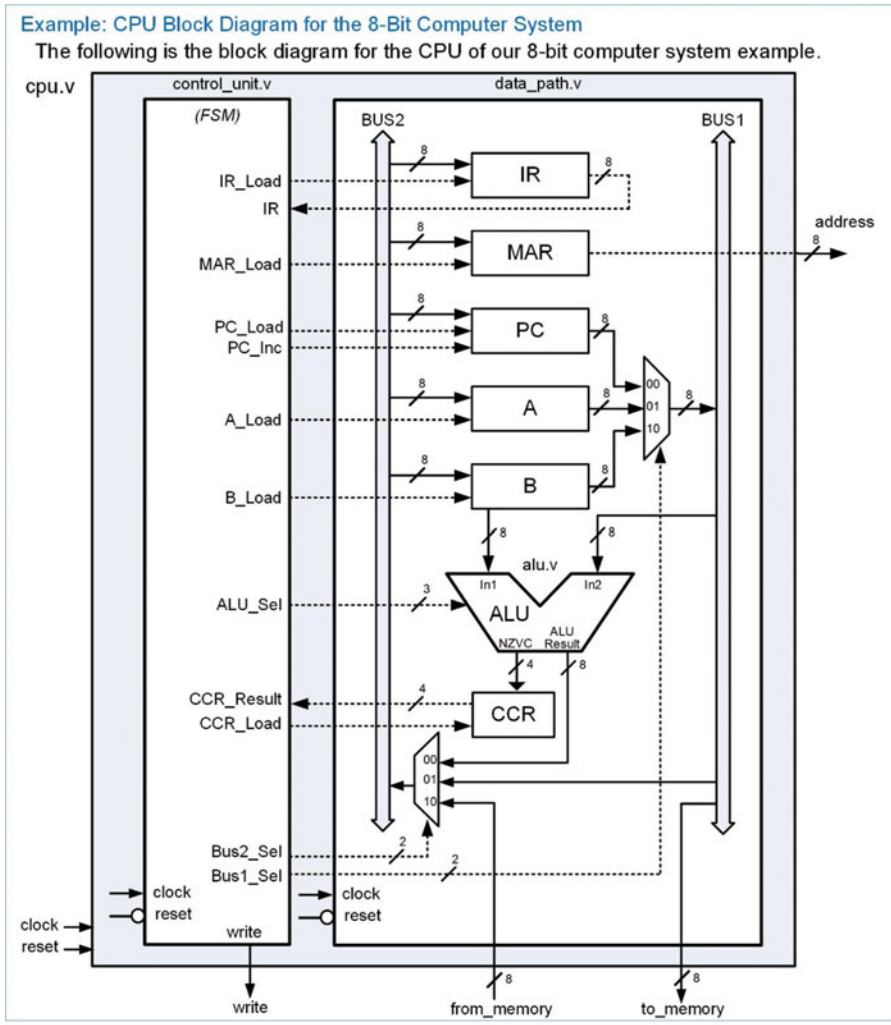
Now that all of the memory functionality has been designed, the final step is to implement the multiplexer that handles routing the appropriate information to the CPU on the data_out bus based on the incoming address. The following Verilog provides a model for this behavior. Recall that a multiplexer is combinational logic, so if the behavior is to be modeled using a procedural block, all inputs must be listed in the sensitivity list and blocking assignments are used. These inputs include the outputs from the program and data memory in addition to all of the input ports. The sensitivity list must also include the

address bus as it acts as the select input to the multiplexer. Within the block, an if-else statement is used to determine which subsystem drives data_out. Program memory will drive data_out when the incoming address is in the range of 0 to 127 (x"00" to x"7F"). Data memory will drive data_out when the address is in the range of 128 to 223 (x"80" to x"DF"). An input port will drive data_out when the address is in the range of 240 to 255 (x"F0" to x"FF"). Each input port has a unique address, so the specific addresses are listed as nested if-else clauses.

```
always @ (address, rom_data_out, rw_data_out,
          port_in_00, port_in_01, port_in_02, port_in_03,
          port_in_04, port_in_05, port_in_06, port_in_07,
          port_in_08, port_in_09, port_in_10, port_in_11,
          port_in_12, port_in_13, port_in_14, port_in_15)

   begin: MUX1

     if ( (address >= 0) && (address <= 127) )
        data_out = rom_data_out;
     else if ( (address >= 128) && (address <= 223) )
        data_out = rw_data_out;
     else if (address == 8'hF0) data_out = port_in_00;
     else if (address == 8'hF1) data_out = port_in_01;
     else if (address == 8'hF2) data_out = port_in_02;
     else if (address == 8'hF3) data_out = port_in_03;
     else if (address == 8'hF4) data_out = port_in_04;
     else if (address == 8'hF5) data_out = port_in_05;
     else if (address == 8'hF6) data_out = port_in_06;
     else if (address == 8'hF7) data_out = port_in_07;
     else if (address == 8'hF8) data_out = port_in_08;
     else if (address == 8'hF9) data_out = port_in_09;
     else if (address == 8'hFA) data_out = port_in_10;
     else if (address == 8'hFB) data_out = port_in_11;
     else if (address == 8'hFC) data_out = port_in_12;
     else if (address == 8'hFD) data_out = port_in_13;
     else if (address == 8'hFE) data_out = port_in_14;
     else if (address == 8'hFF) data_out = port_in_15;

   end
```

11.3.4 CPU Implementation

Let us now look at the central processing unit details. The CPU contains two components, the control unit (control_unit.v) and the data path (data_path.v). The data path contains all of the registers and the ALU. The ALU is implemented as a subsystem within the data path (alu.v). The data path also contains a bus system in order to facilitate data movement between the registers and memory. The bus system is implemented with two multiplexers that are controlled by the control unit. The control unit contains the finite state machine that generates all control signals for the data path as it performs the fetch-decode-execute steps of each instruction. Example 11.11 shows the block diagram of the CPU in our 8-bit microcomputer example.



Example 11.11
CPU block diagram for the 8-bit computer system

11.3.4.1 Data Path Implementation in Verilog

Let us first look at the data path bus system that handles internal signal routing. The system consists of two 8-bit busses (Bus1 and Bus2) and two multiplexers. Bus1 is used as the destination of the PC, A, and B register outputs, while Bus2 is used as the input to the IR, MAR, PC, A, and B registers. Bus1 is connected directly to the *to_memory* port of the CPU to allow registers to write data to the memory system. Bus2 can be driven by the *from_memory* port of the CPU to allow the memory system to provide data for the CPU registers. The two multiplexers handle all signal routing and have their select lines (Bus1_Sel and Bus2_Sel) driven by the control unit. The following Verilog shows how the multiplexers are implemented. Again, a multiplexer is combinational logic, so all inputs must be listed in the sensitivity list of its procedural block and blocking assignments are used. Two additional signal assignments are also required to connect the MAR to the address port and to connect Bus1 to the to_memory port.

```
always @ (Bus1_Sel, PC, A, B)
  begin: MUX_BUS1
    case (Bus1_Sel)
      2'b00   : Bus1 = PC;
      2'b01   : Bus1 = A;
      2'b10   : Bus1 = B;
      default : Bus1 = 8'hXX;
    endcase
  end

always @ (Bus2_Sel, ALU_Result, Bus1, from_memory)
  begin: MUX_BUS2
    case (Bus2_Sel)
      2'b00   : Bus2 = ALU_Result;
      2'b01   : Bus2 = Bus1;
      2'b10   : Bus2 = from_memory;
      default : Bus1 = 8'hXX;
    endcase
  end

always @ (Bus1, MAR)
  begin
    to_memory = Bus1;
    address = MAR;
  end
```

Next, let us look at implementing the registers in the data path. Each register is implemented using a dedicated procedural block that is sensitive to clock and reset. This models the behavior of synchronous latches, or registers. Each register has a synchronous enable line that dictates when the register is updated. The register output is only updated when the enable line is asserted and a rising edge of the clock is detected. The following Verilog shows how to model the instruction register (IR). Notice that the signal IR is only updated if IR_Load is asserted and there is a rising edge of the clock. In this case, IR is loaded with the value that resides on Bus2.

```
always @ (posedge clock or negedge reset)
  begin: INSTRUCTION_REGISTER
    if (!reset)
      IR <= 8'h00;
    else
      if (IR_Load)
        IR <= Bus2;
  end
```

A nearly identical block is used to model the memory address register. A unique signal is declared called *MAR* in order to make the Verilog more readable. MAR is always assigned to address in this system.

```
always @ (posedge clock or negedge reset)
  begin: MEMORY_ADDRESS_REGISTER
    if (!reset)
      MAR <= 8'h00;
    else
      if (MAR_Load)
        MAR <= Bus2;
  end
```

Now let us look at the program counter block. This register contains additional functionality beyond simply latching in the value of Bus2. The program counter also has an increment feature that will take place synchronously when the signal PC_Inc coming from the control unit is asserted. This is handled using an additional nested if-else clause under the portion of the block handling the rising edge of clock condition.

```
always @ (posedge clock or negedge reset)
  begin: PROGRAM_COUNTER
    if (!reset)
      PC <= 8'h00;
    else
      if (PC_Load)
       PC <= Bus2;
      else if (PC_Inc)
       PC <= MAR + 1;
  end
```

The two general-purpose registers A and B are modeled using individual procedural blocks as follows:

```
always @ (posedge clock or negedge reset)
  begin: A_REGISTER
    if (!reset)
      A <= 8'h00;
    else
      if (A_Load)
        A <= Bus2;
  end

always @ (posedge clock or negedge reset)
  begin: B_REGISTER
    if (!reset)
      B <= 8'h00;
    else
      if (B_Load)
        B <= Bus2;
  end
```

The condition code register latches in the status flags from the ALU (NZVC) when the CCR_Load line is asserted. This behavior is modeled using a similar approach as follows:

```
always @ (posedge clock or negedge reset)
  begin: CONDITION_CODE_REGISTER
    if (!reset)
      CCR_Result <= 8'h00;
    else
      if (CCR_Load)
        CCR_Result <= NZVC;
  end
```

11.3.4.2 ALU Implementation in Verilog

The ALU is a set of combinational logic circuitry that performs arithmetic and logic operations. The output of the ALU operation is called *Result*. The ALU also outputs four status flags as a 4-bit bus called *NZVC*. The ALU behavior can be modeled using case and if-else statements that decide which operation to perform based on the input control signal *ALU_Sel*. The following Verilog shows an example of how to implement the ALU addition functionality. A case statement is used to decide which operation is being performed based on the ALU_Sel input. Under each operation clause, a series of procedural statements are used to compute the result and update the NZVC flags. Each of these flags is updated individually. The N flag can be simply driven with position 7 of the ALU result since this bit is the sign bit for signed numbers. The Z flag can be driven using an if-else condition that checks whether the result was x"00". The V flag is updated based on the type of the operation. For the addition operation, the V flag will be asserted if a POS + POS=NEG or a NEG + NEG = POS. These conditions can be checked by looking at the sign bits of the inputs and the sign bit of the result. Finally, the C flag can be computed as the eighth bit in the addition of In1 + In2.

```
always @ (In1, In2, ALU_Sel)
   begin
     case (ALU_Sel)
       3'b000   : begin //-- Addition

                  //-- Sum and Carry Flag
                    {NZVC[0], Result} = In1 + In2;

                  //-- Negative Flag
                    NZVC[3] = Result[7];

                  //-- Zero Flag
                    if (Result == 0)
                       NZVC[2] = 1;
                    else
                       NZVC[2] = 0;


                  //-- Two's Comp Overflow Flag
                    if ( ((In1[7]==0) && (In2[7]==0) && (Result[7] == 1)) ||
                         ((In1[7]==1) && (In2[7]==1) && (Result[7] == 0)) )
                       NZVC[1] = 1;
                    else
                       NZVC[1] = 0;

                  end

                        :
        //-- other ALU operations go here...
                        :

        default  : begin
                     Result  = 8'hXX;
                     NZVC    = 4'hX;
                   end
     endcase

   end
```

11.3.4.3 Control Unit Implementation in Verilog

Let us now look at how to implement the control unit state machine. We'll first look at the formation of the Verilog to model the FSM and then turn to the detailed state transitions in order to accomplish a variety of the most common instructions. The control unit sends signals to the data path in order to move data in and out of registers and into the ALU to perform data manipulations. The finite state machine is implemented with the behavioral modeling techniques presented in Chap. 9. The model contains three processes in order to implement the state memory, next state logic, and output logic of the FSM. Parameters are created for each of the states defined in the state diagram of the FSM. The states associated with fetching (S_FETCH_0, S_FETCH_1, S_FETCH_2) and decoding the opcode (S_DECODE_3) are performed each time an instruction is executed. A unique path is then added after the decode state to perform the steps associated with executing each individual instruction. The FSM can be created one instruction at a time by adding additional state paths after the decode state. The following Verilog code shows how the user-defined state names are created for nine basic instructions (LDA_IMM, LDA_DIR, STA_DIR, LDB_IMM, LDB_DIR, STB_DIR, ADD_AB, BRA, and BEQ). Eight-bit state variables are created for current_state and next_state to accommodate future state codes. The state codes are assigned in binary using integer format to allow additional states to be easily added.

```
reg    [7:0] current_state, next_state;
parameter S_FETCH_0  = 0,    //-- Opcode fetch states
          S_FETCH_1  = 1,
          S_FETCH_2  = 2,

          S_DECODE_3 = 3,    //-- Opcode decode state

          S_LDA_IMM_4 = 4,   //-- Load A (Immediate) states
          S_LDA_IMM_5 = 5,
          S_LDA_IMM_6 = 6,

          S_LDA_DIR_4 = 7,   //-- Load A (Direct) states
          S_LDA_DIR_5 = 8,
          S_LDA_DIR_6 = 9,
          S_LDA_DIR_7 = 10,
          S_LDA_DIR_8 = 11,

          S_STA_DIR_4 = 12,  //-- Store A (Direct) States
          S_STA_DIR_5 = 13,
          S_STA_DIR_6 = 14,
          S_STA_DIR_7 = 15,

          S_LDB_IMM_4 = 16,  //-- Load B (Immediate) states
          S_LDB_IMM_5 = 17,
          S_LDB_IMM_6 = 18,

          S_LDB_DIR_4 = 19,  //-- Load B (Direct) states
          S_LDB_DIR_5 = 20,
          S_LDB_DIR_6 = 21,
          S_LDB_DIR_7 = 22,
          S_LDB_DIR_8 = 23,

          S_STB_DIR_4 = 24,  //-- Store B (Direct) States
          S_STB_DIR_5 = 25,
          S_STB_DIR_6 = 26,
          S_STB_DIR_7 = 27,

          S_BRA_4   = 28,    //-- Branch Always States
          S_BRA_5   = 29,
          S_BRA_6   = 30,

          S_BEQ_4   = 31,    //-- Branch if Equal States
          S_BEQ_5   = 32,
          S_BEQ_6   = 33,
          S_BEQ_7   = 34,

          S_ADD_AB_4 = 35;   //-- Addition States
```

Within the control unit module, the state memory is implemented as a separate procedural block that will update the current state with the next state on each rising edge of the clock. The reset state will be the first fetch state in the FSM (i.e., S_FETCH_0). The following Verilog shows how the state memory in the control unit can be modeled. Note that this block models sequential logic, so nonblocking assignments are used.

```
always @ (posedge clock or negedge reset)
  begin: STATE_MEMORY
    if (!reset)
       current_state <= S_FETCH_0;
    else
       current_state <= next_state;
  end
```

The next state logic is also implemented as a separate procedural block. The next state logic depends on the current state, instruction register (IR), and the condition code register (CCR_Result). The following Verilog gives a portion of the next state logic process showing how the state transitions can be modeled.

```
always @ (current_state, IR, CCR_Result)
  begin: NEXT_STATE_LOGIC
    case (current_state)
      S_FETCH_0   : next_state = S_FETCH_1;    //-- Path for FETCH instruction
      S_FETCH_1   : next_state = S_FETCH_2;
      S_FETCH_2   : next_state = S_DECODE_3;

      S_DECODE_3  : if      (IR == LDA_IMM) next_state = S_LDA_IMM_4; //-- Register A
                    else if (IR == LDA_DIR) next_state = S_LDA_DIR_4;
                    else if (IR == STA_DIR) next_state = S_STA_DIR_4;
                    else if (IR == LDB_IMM) next_state = S_LDB_IMM_4; //-- Register B
                    else if (IR == LDB_DIR) next_state = S_LDB_DIR_4;
                    else if (IR == STB_DIR) next_state = S_STB_DIR_4;
                    else if (IR == BRA)     next_state = S_BRA_4;     //-- Branch Always
                    else if (IR == ADD_AB)  next_state = S_ADD_AB_4;  //-- ADD
                    else                    next_state = S_FETCH_0;   //-- others go here

      S_LDA_IMM_4 : next_state = S_LDA_IMM_5;  //-- Path for LDA_IMM instruction
      S_LDA_IMM_5 : next_state = S_LDA_IMM_6;
      S_LDA_IMM_6 : next_state = S_FETCH_0;

                            :
         Next state logic for other states goes here…
                            :
    endcase
  end
```

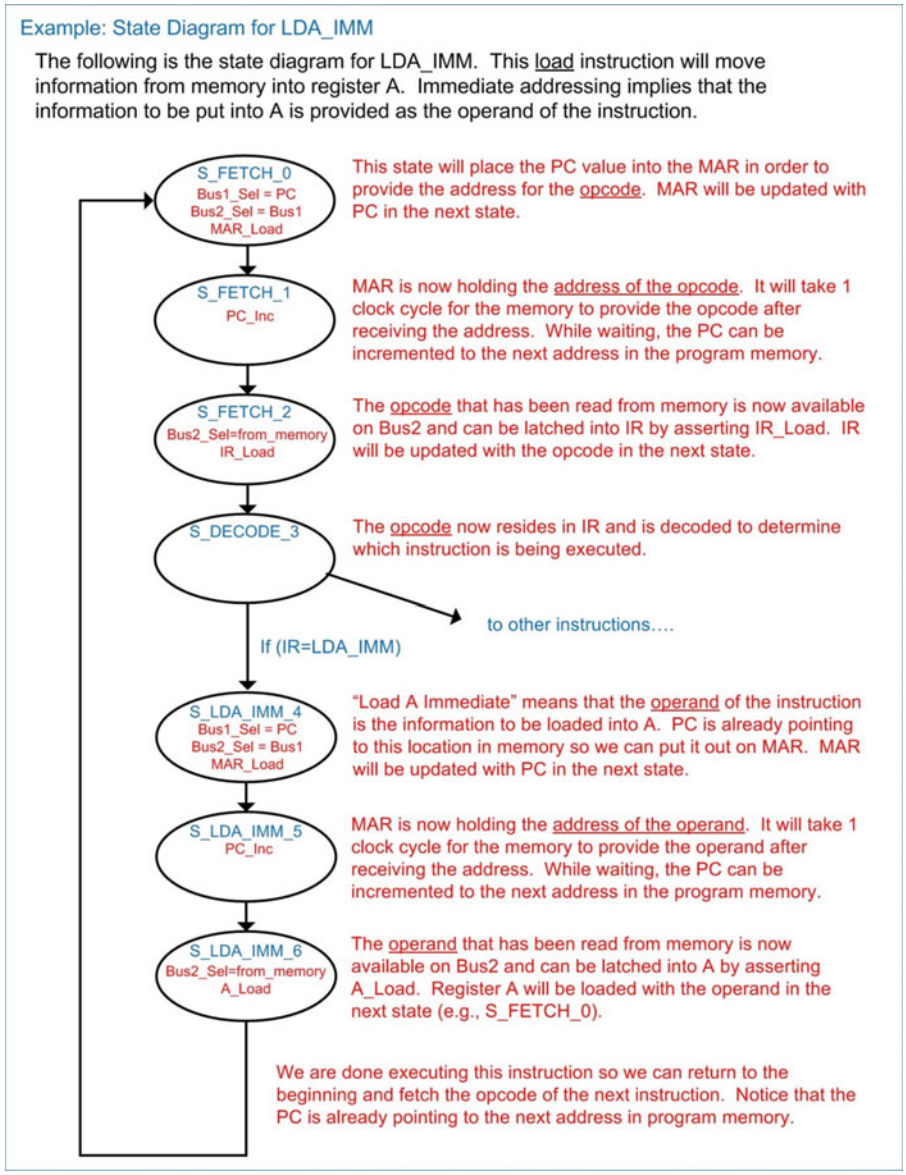
Finally, the output logic is modeled as a third, separate procedural block. It is useful to explicitly state the outputs of the control unit for each state in the machine to allow easy debugging and avoid synthesizing latches. Our example computer system has Moore type outputs, so the process only depends on the current state. The following Verilog shows a portion of the output logic process.

```
always @ (current_state)
  begin: OUTPUT_LOGIC
    case (current_state)

      S_FETCH_0   : begin           //-- Put PC onto MAR to provide address of Opcode
                      IR_Load  = 0;
                      MAR_Load = 1;
                      PC_Load  = 0;
                      PC_Inc   = 0;
                      A_Load   = 0;
                      B_Load   = 0;
                      ALU_Sel  = 3'b000;
                      CCR_Load = 0;
                      Bus1_Sel = 2'b00; //-- "00"=PC,  "01"=A,    "10"=B
                      Bus2_Sel = 2'b01; //-- "00"=ALU, "01"=Bus1, "10"=from_memory
                      write    = 0;
                    end
```

```
    S_FETCH_1   : begin          //-- Increment PC, Opcode will be available next state
                    IR_Load  = 0;
                    MAR_Load = 0;
                    PC_Load  = 0;
                    PC_Inc   = 1;
                    A_Load   = 0;
                    B_Load   = 0;
                    ALU_Sel  = 3'b000;
                    CCR_Load = 0;
                    Bus1_Sel = 2'b00;  //-- "00"=PC,  "01"=A,    "10"=B
                    Bus2_Sel = 2'b00;  //-- "00"=ALU, "01"=Bus1, "10"=from_memory
                    write    = 0;
                  end;

                          :
        Output logic for other states goes here…
                          :
  endcase
end
```

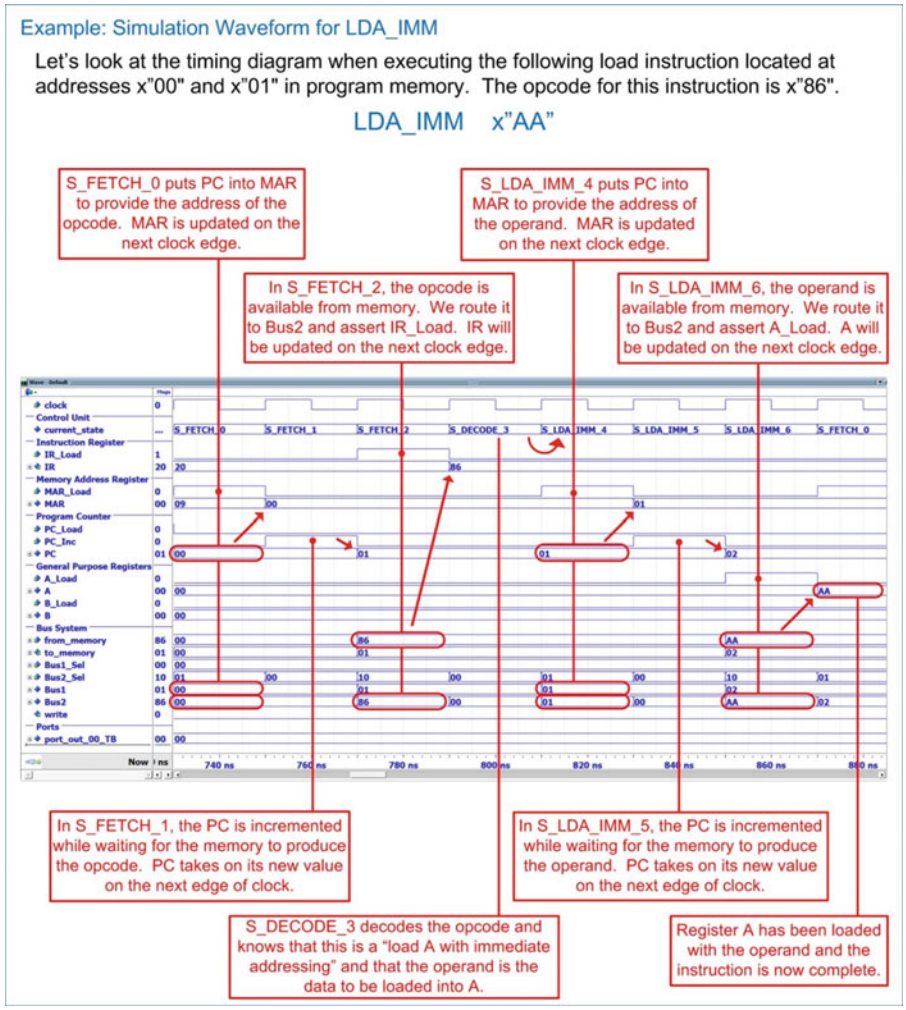
11.3.4.3.1 Detailed Execution of LDA_IMM

Now let us look at the details of the state transitions and output signals in the control unit FSM when executing a few of the most common instructions. Let us begin with the instruction to load register A using immediate addressing (LDA_IMM). Example 11.12 shows the state diagram for this instruction. The first three states (S_FETCH_0, S_FETCH_1, S_FETCH_2) handle fetching the opcode. The purpose of these states is to read the opcode from the address being held by the program counter and put it into the instruction register. Multiple states are needed to handle putting PC into MAR to provide the address of the opcode, waiting for the memory system to provide the opcode, latching the opcode into IR, and incrementing PC to the next location in program memory. Another state is used to decode the opcode (S_DECODE_3) in order to decide which path to take in the state diagram based on the instruction being executed. After the decode state, a series of three more states are needed (S_LDA_IMM_4, S_LDA_IMM_5, S_LDA_IMM_6) to execute the instruction. The purpose of these states is to read the operand from the address being held by the program counter and put it into A. Multiple states are needed to handle putting PC into MAR to provide the address of the operand, waiting for the memory system to provide the operand, latching the operand into A, and incrementing PC to the next location in program memory. When the instruction completes, the value of the operand resides in A and PC is pointing to the next location in program memory, which is the opcode of the next instruction to be executed.



Example 11.12
State diagram for LDA_IMM

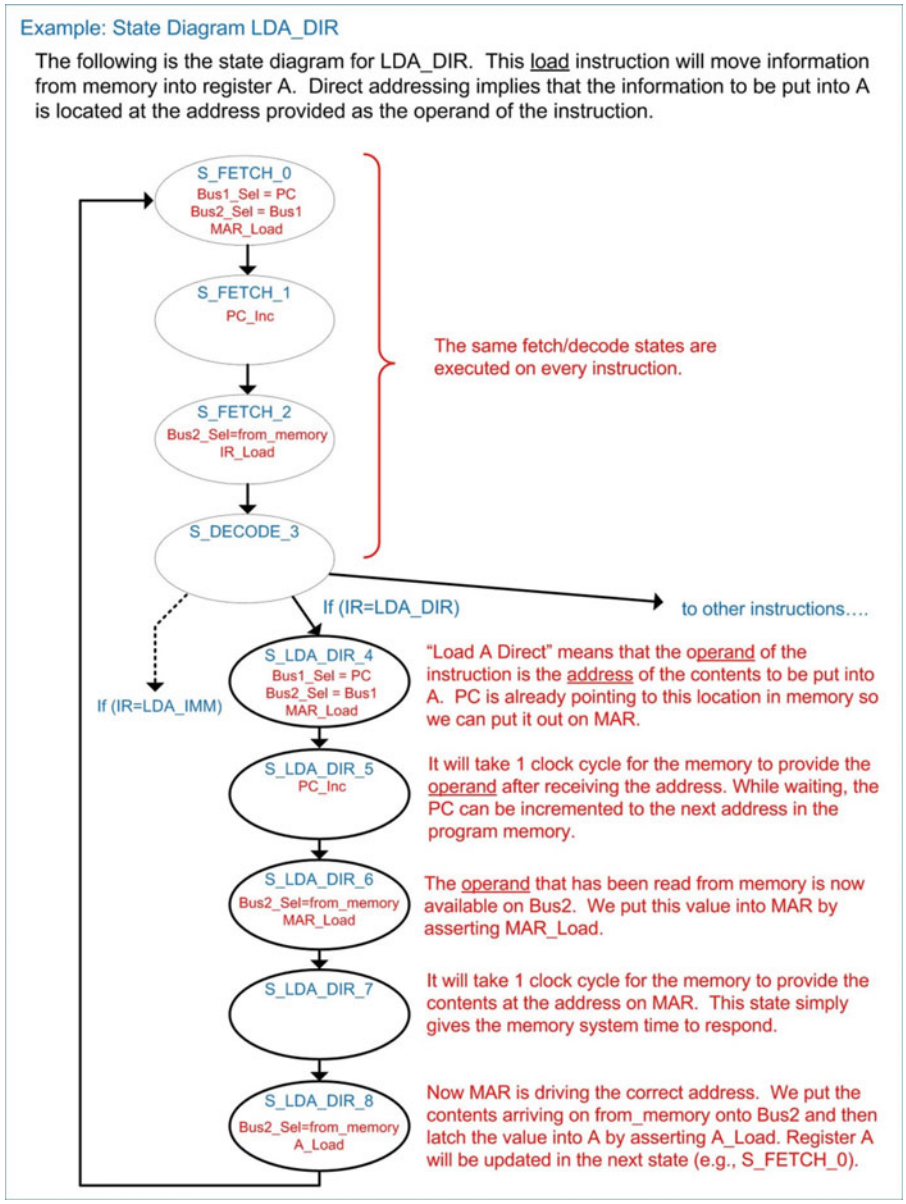
Example 11.13 shows the simulation waveform for executing LDA_IMM. In this example, register A is loaded with the operand of the instruction, which holds the value x"AA".



Example 11.13
Simulation waveform for LDA_IMM

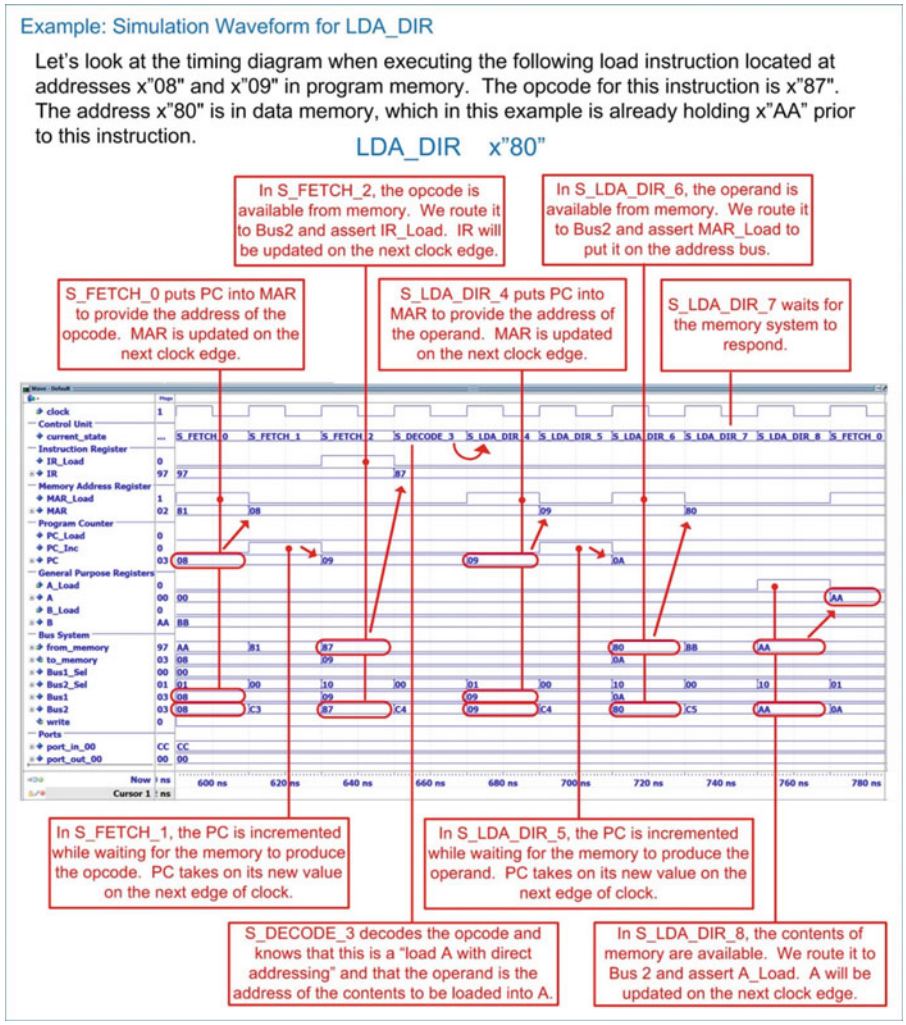
11.3.4.3.2 Detailed Execution of LDA_DIR

Now let us look at the details of the instruction to load register A using direct addressing (LDA_DIR). Example 11.14 shows the state diagram for this instruction. The first four states to fetch and decode the opcode are the same states as in the previous instruction and are performed each time a new instruction is executed. Once the opcode is decoded, the state machine traverses five new states to execute the instruction (S_LDA_DIR_4, S_LDA_DIR_5, S_LDA_DIR_6, S_LDA_DIR_7, S_LDA_DIR_8). The purpose of these states is to read the operand and then use it as the address of where to read the contents to put into A.



Example 11.14
State diagram for LDA_DIR

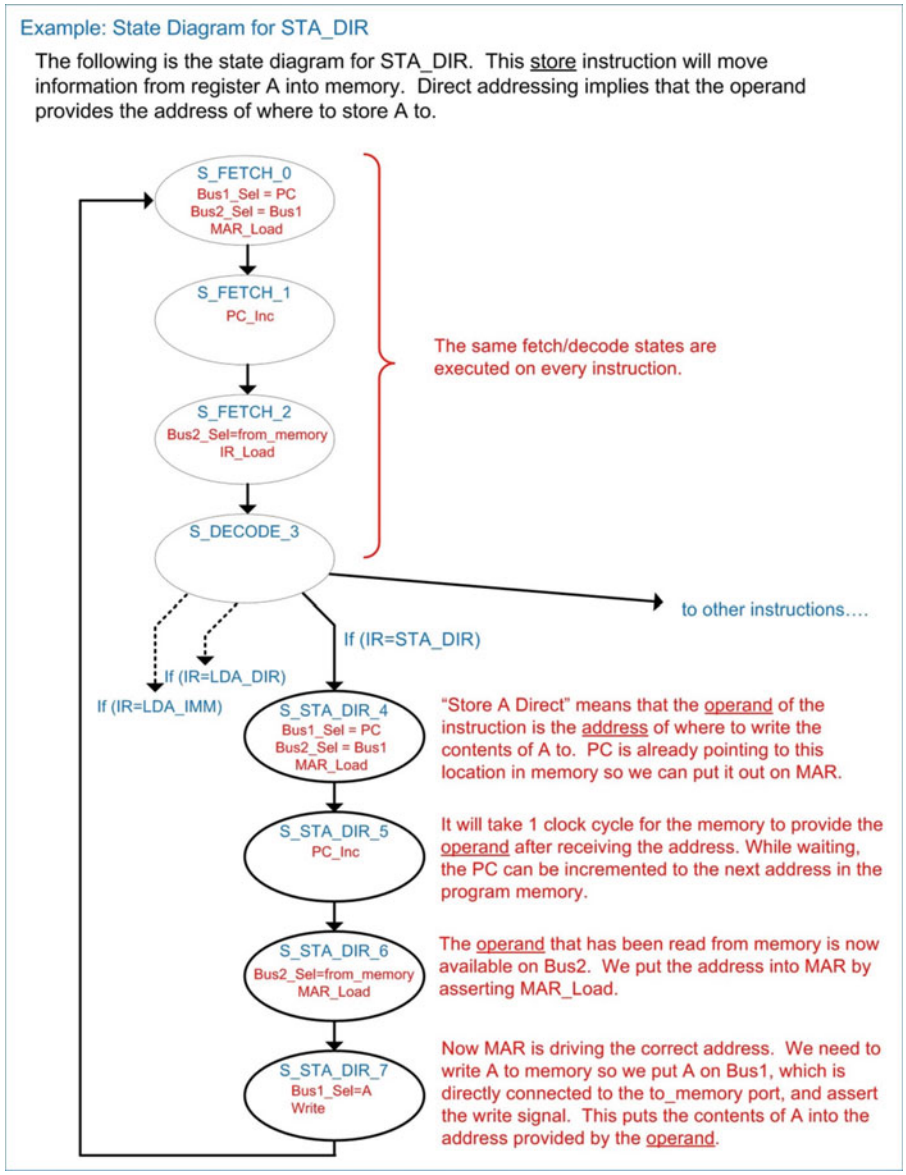
Example 11.15 shows the simulation waveform for executing LDA_DIR. In this example, register A is loaded with the contents located at address x"80", which has already been initialized to x"AA".



Example 11.15
Simulation waveform for LDA_DIR

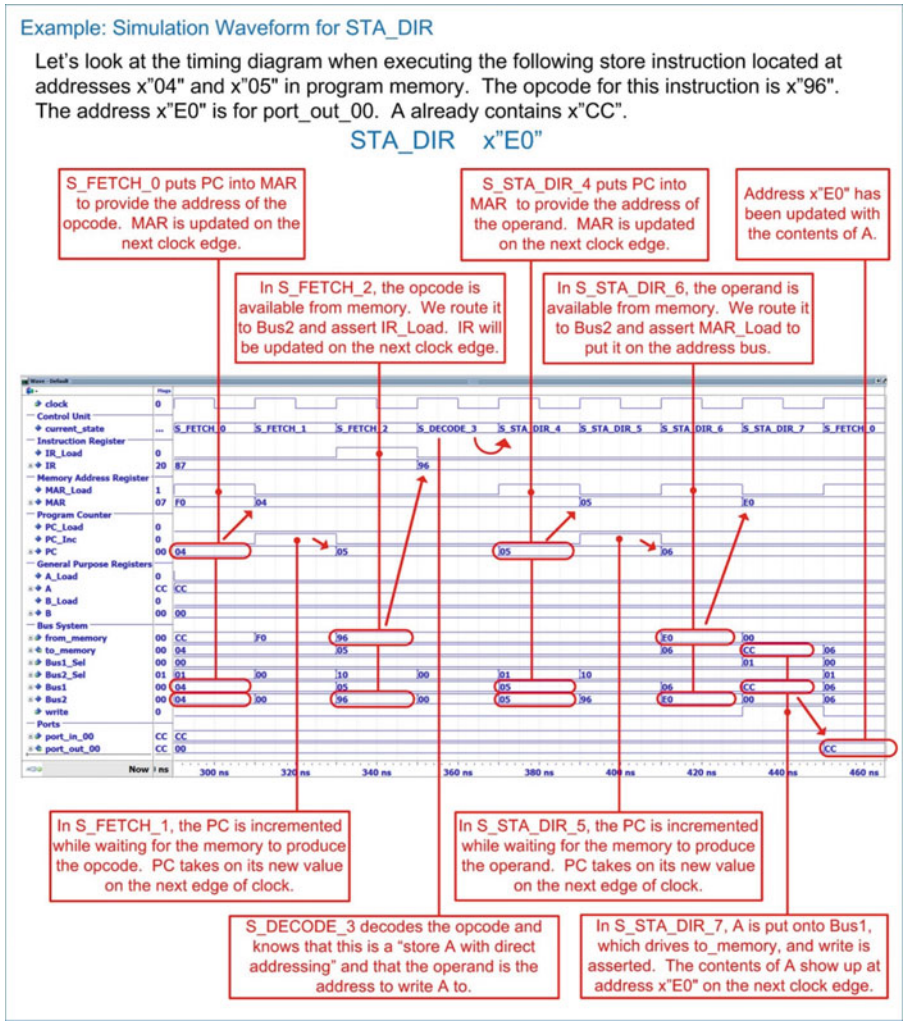
11.3.4.3.3 Detailed Execution of STA_DIR

Now let us look at the details of the instruction to store register A to memory using direct addressing (STA_DIR). Example 11.16 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine traverses four new states to execute the instruction (S_STA_DIR_4, S_STA_DIR_5, S_STA_DIR_6, S_STA_DIR_7). The purpose of these states is to read the operand and then use it as the address of where to write the contents of A to.



Example 11.16
State diagram for STA_DIR

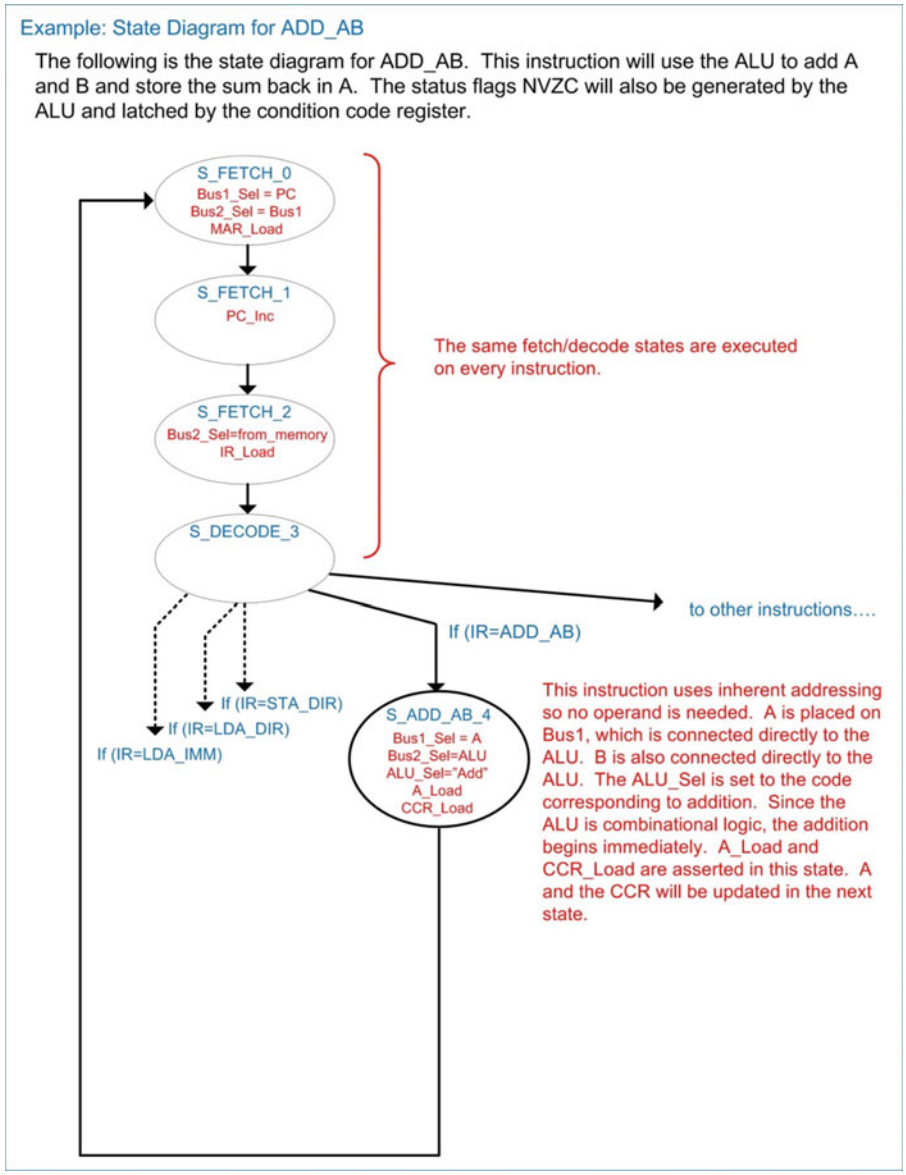
Example 11.17 shows the simulation waveform for executing STA_DIR. In this example, register A already contains the value x"CC" and will be stored to address x"E0". The address x"E0" is an output port (port_out_00) in our example computer system.



Example 11.17
Simulation waveform for STA_DIR

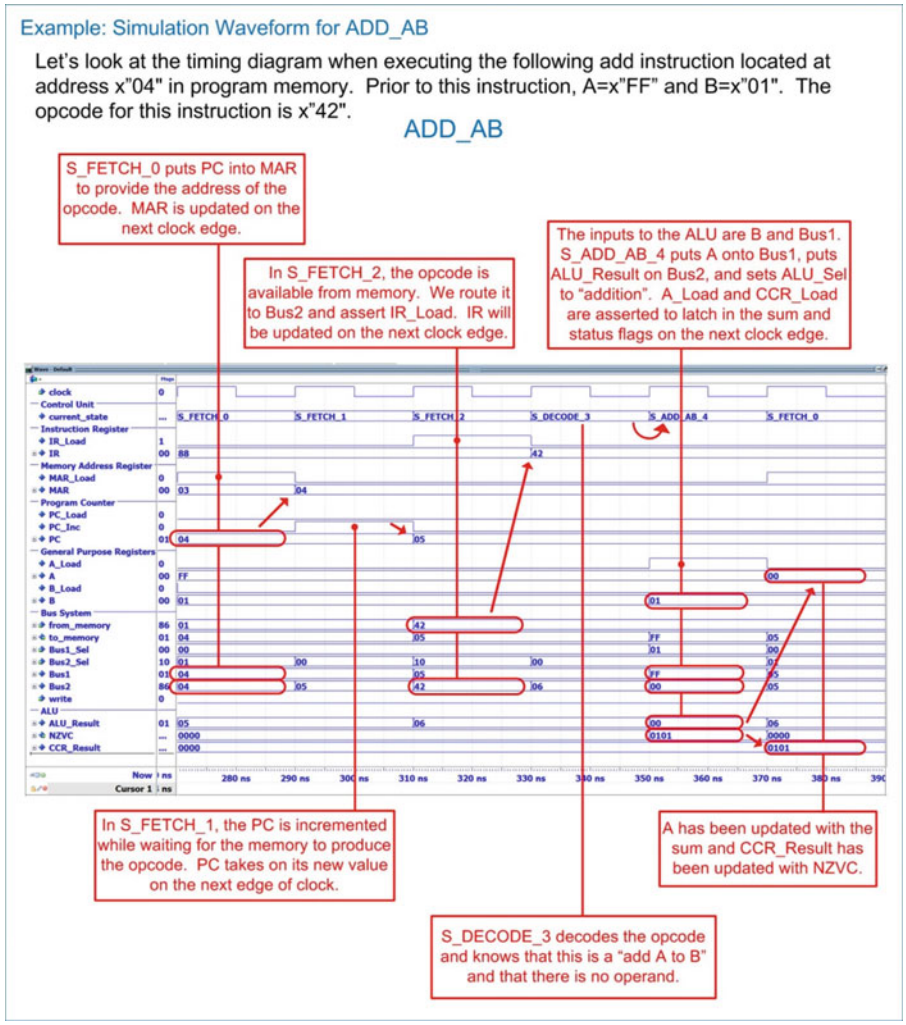
11.3.4.3.4 Detailed Execution of ADD_AB

Now let us look at the details of the instruction to add A to B and store the sum back in A (ADD_AB). Example 11.18 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine only requires one more state to complete the operation (S_ADD_AB_4). The ALU is combinational logic so it will begin to compute the sum immediately as soon as the inputs are updated. The inputs to the ALU are Bus1 and register B. Since B is directly connected to the ALU, all that is required to start the addition is to put A onto Bus1. The output of the ALU is put on Bus2 so that it can be latched into A on the next clock edge. The ALU also outputs the status flags NZVC, which are directly connected to the condition code register. A_Load and CCR_Load are asserted in this state. A and CCR_Result will be updated in the next state (i.e., S_FETCH_0).



Example 11.18
State diagram for ADD_AB

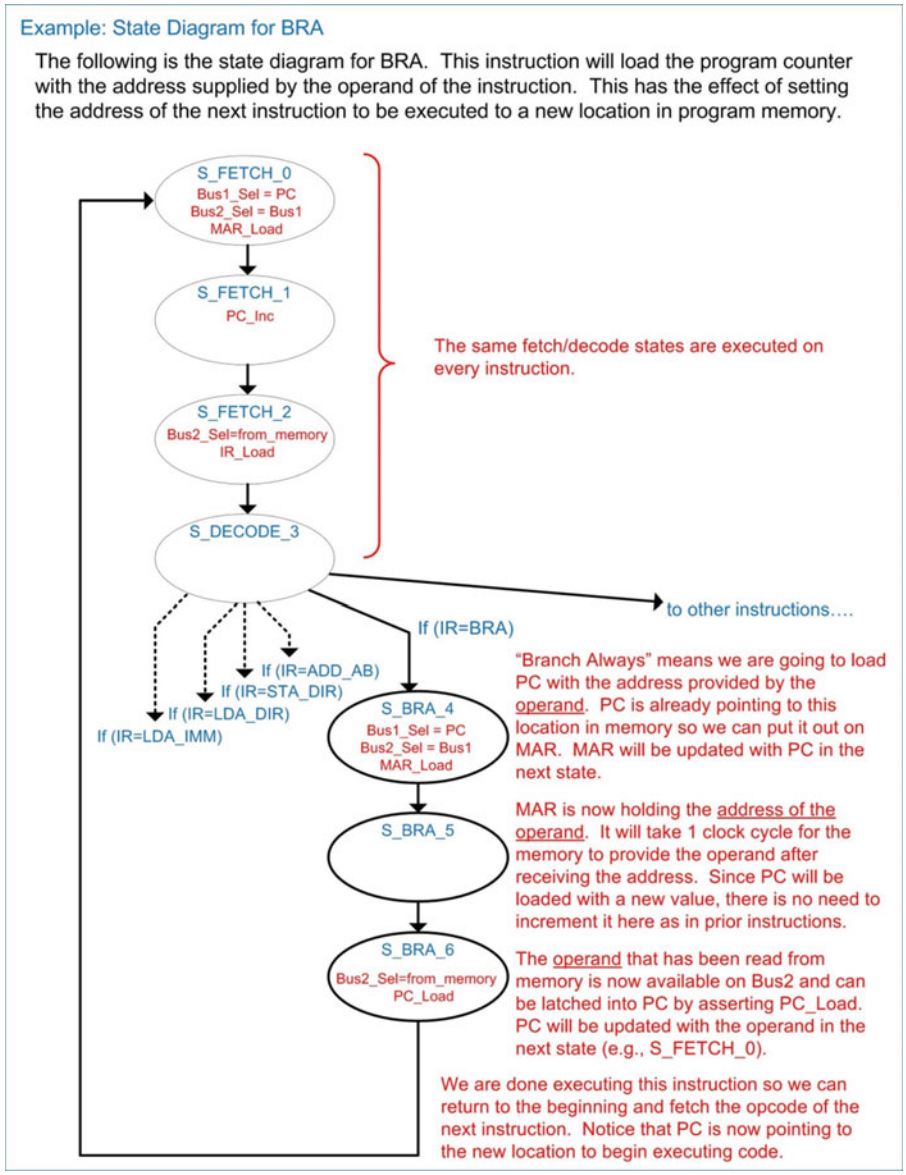
Example 11.19 shows the simulation waveform for executing ADD_AB. In this example, two load immediate instructions were used to initialize the general-purpose registers to A = x"FF" and B = x"01" prior to the addition. The addition of these values will result in a sum of x"00" and assert the carry (C) and zero (Z) flags in the condition code register.



Example 11.19
Simulation waveform for ADD_AB

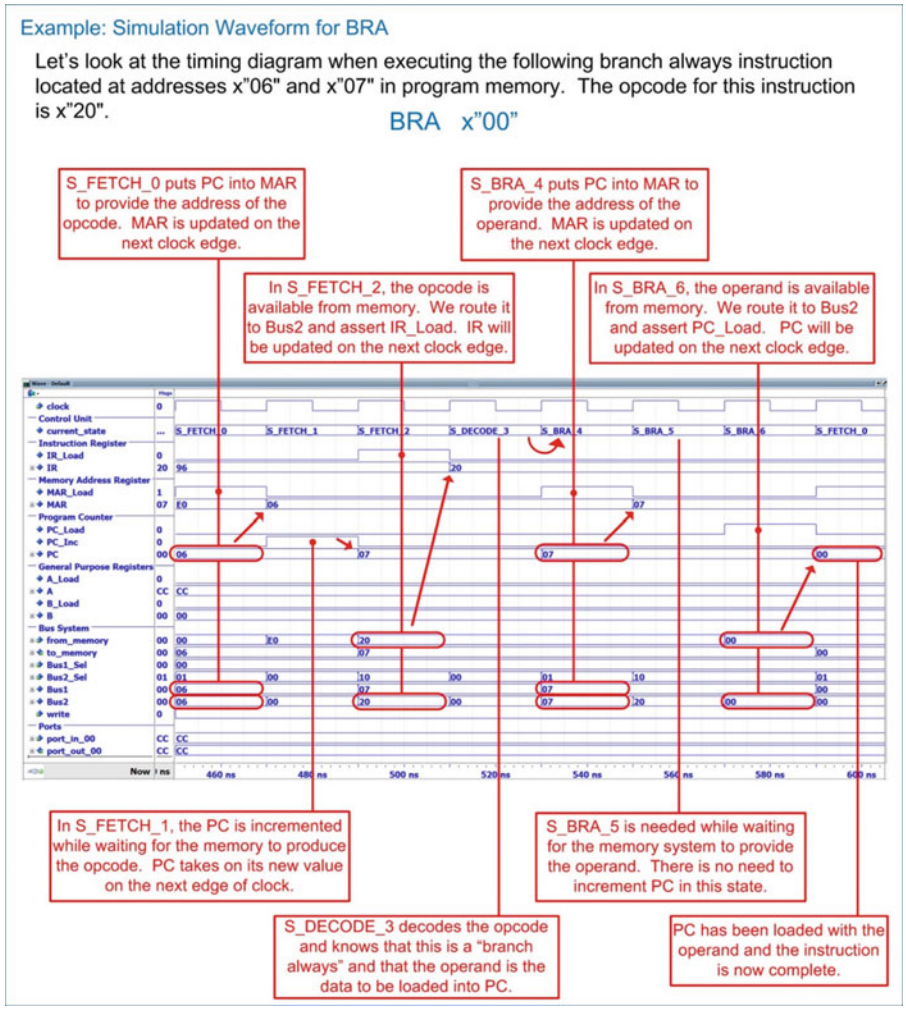
11.3.4.3.5 Detailed Execution of BRA

Now let us look at the details of the instruction to branch always (BRA). Example 11.20 shows the state diagram for this instruction. The first four states are again the same as prior instructions in order to fetch and decode the opcode. Once the opcode is decoded, the state machine traverses four new states to execute the instruction (S_BRA_4, S_BRA_5, S_BRA_6). The purpose of these states is to read the operand and put its value into PC to set the new location in program memory to execute instructions.



Example 11.20
State diagram for BRA

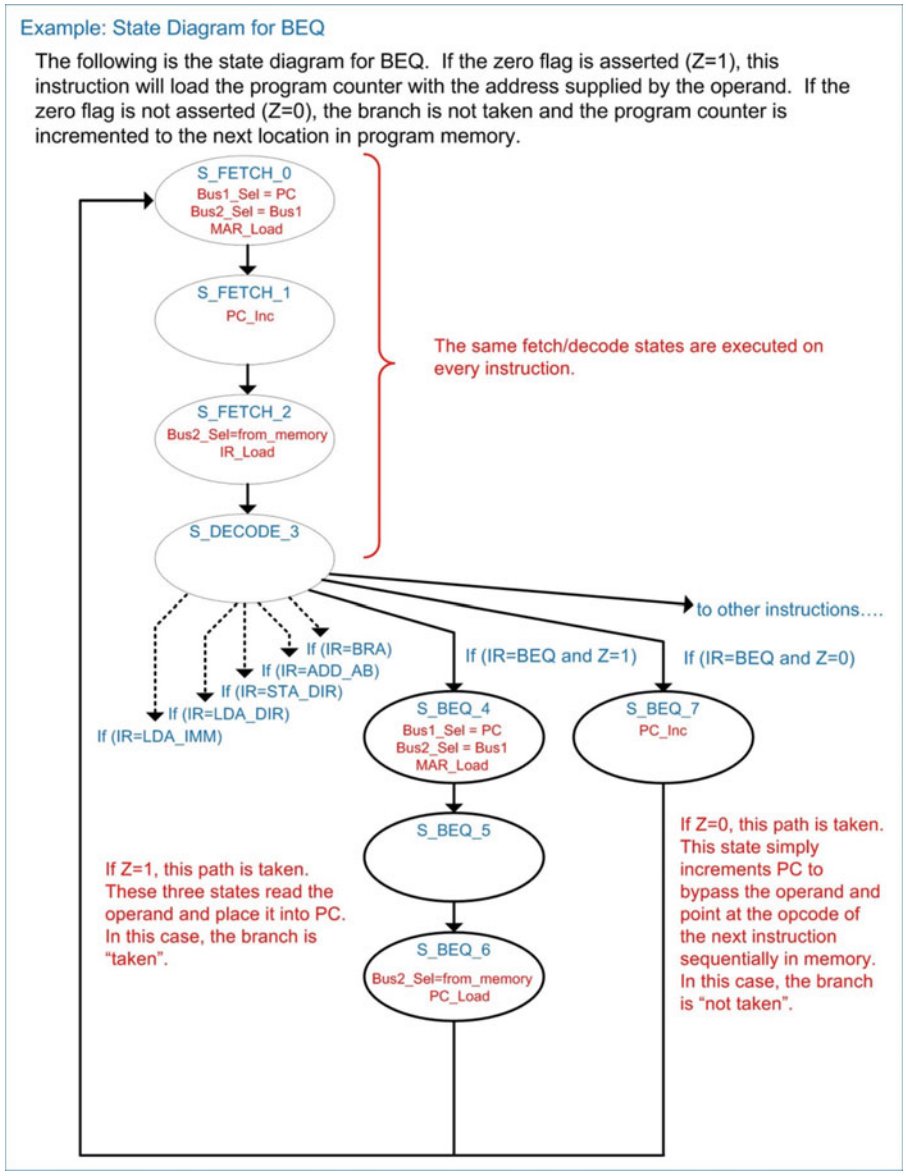
Example 11.21 shows the simulation waveform for executing BRA. In this example, PC is set back to address x"00".



Example 11.21
Simulation waveform for BRA

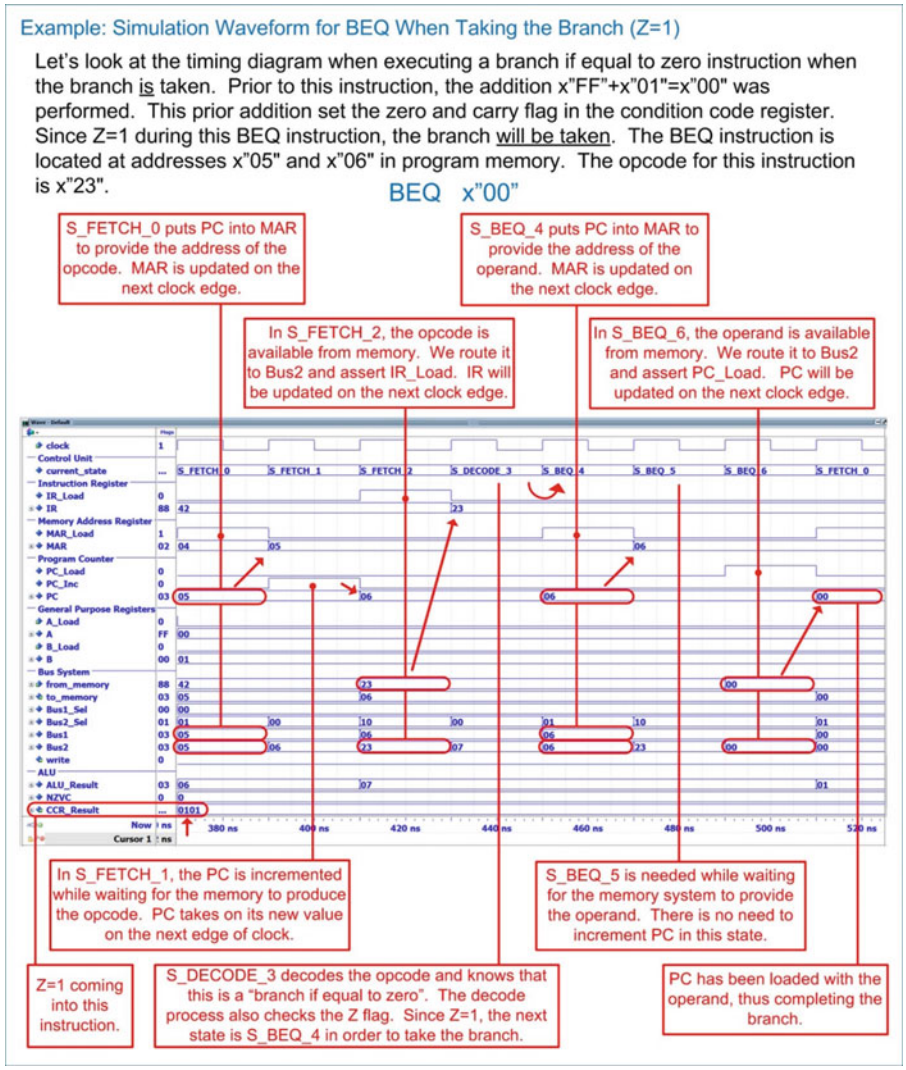
11.3.4.3.6 Detailed Execution of BEQ

Now let us look at the branch if equal to zero (BEQ) instruction. Example 11.22 shows the state diagram for this instruction. Notice that in this conditional branch, the path that is taken through the FSM depends on both IR and CCR. In the case that $Z = 1$, the branch is taken, meaning that the operand is loaded into PC. In the case that $Z = 0$, the branch is not taken, meaning that PC is simply incremented to bypass the operand and point to the beginning of the next instruction in program memory.



Example 11.22
State diagram for BEQ

Example 11.23 shows the simulation waveform for executing BEQ when the branch *is taken*. Prior to this instruction, an addition was performed on x"FF" and x"01". This resulted in a sum of x"00", which asserted the Z and C flags in the condition code register. Since $Z = 1$ when BEQ is executed, the branch is taken.



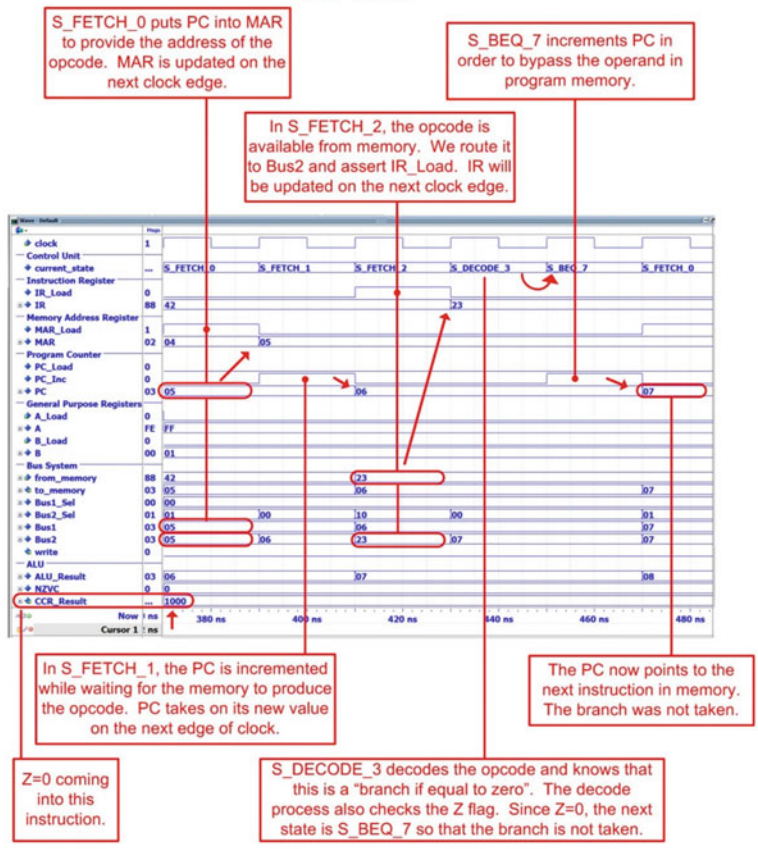
Example 11.23
Simulation waveform for BEQ when taking the branch (Z = 1)

Example 11.24 shows the simulation waveform for executing BEQ when the branch *is not taken*. Prior to this instruction, an addition was performed on x"FE" and x"01". This resulted in a sum of x"FF", which did not assert the Z flag. Since Z = 0 when BEQ is executed, the branch is not taken. When not taking the branch, PC must be incremented again in order to bypass the operand and point to the next location in program memory.

Example: Simulation Waveform for BEQ When the Branch is Not Taken (Z=0)

Let's look at the timing diagram when executing a branch if equal to zero instruction when the branch is not taken. Prior to this instruction, the addition x"FE"+x"01"=x"FF" was performed. This addition did not set the zero in the condition code register. Since this operation resulted in Z=0, the branch will not be taken. The BEQ instruction is located at addresses x"05" and x"06" in program memory. The opcode for this instruction is x"23".

BEQ x"00"



Example 11.24
Simulation waveform for BEQ when the branch is not taken (Z = 0)

Concept Check

CC11.3 The 8-bit microcomputer example presented in this section is a very simple architecture used to illustrate the basic concepts of a computer. If we wanted to keep this computer an 8-bit system but increase the depth of the memory, it would require adding more address lines to the address bus. What changes to the computer system would need to be made to accommodate the wider address bus?

A) The width of the program counter would need to be increased to support the wider address bus.

B) The size of the memory address register would need to be increased to support the wider address bus.

C) Instructions that use direct addressing would need additional bytes of operand to pass the wider address into the CPU 8-bits at a time.

D) All of the above.

Summary

- A computer is a collection of hardware components that are constructed to perform a specific set of instructions to process and store data. The main hardware components of a computer are the central processing unit (CPU), program memory, data memory, and input/output ports.
- The CPU consists of registers for fast storage, an arithmetic logic unit (ALU) for data manipulation, and a control state machine that directs all activity to execute an instruction.
- A CPU is typically organized into a *data path* and a *control unit*. The data path contains circuitry used to store and process information. The data path includes registers and the ALU. The control unit is a large state machine that sends control signals to the data path in order to facilitate instruction execution.
- The control unit performs a *fetch-decode-execute* cycle in order to complete instructions.
- The instructions that a computer is designed to execute are called its *instruction set*.
- Instructions are inserted into *program memory* in a sequence that when executed will accomplish a particular task. This sequence of instructions is called a computer *program*.
- An instruction consists of an *opcode* and a potential *operand*. The opcode is the unique binary code that tells the control state machine which instruction is being executed. An operand is additional information that may be needed for the instruction.
- An *addressing mode* refers to the way that the operand is treated. In *immediate* addressing, the operand is the actual data to be used. In *direct* addressing, the operand is the address of where the data are to be retrieved or stored. In *inherent* addressing, all of the information needed to complete the instruction is contained within the opcode, so no operand is needed.
- A computer also contains *data memory* to hold temporary variables during run time.
- A computer also contains input and output ports to interface with the outside world.
- A *memory-mapped* system is one in which the program memory, data memory, and I/O ports are all assigned a unique address. This allows the CPU to simply process information as data and addresses and allows the program to handle where the information is being sent to. A *memory map* is a graphical representation of what address ranges various components are mapped to.
- There are three primary classes of instructions. These are loads and stores, data manipulations, and branches.
- Load instructions move information from memory into a CPU register. A load instruction takes multiple read cycles. Store instructions move information from a CPU register into memory. A store instruction

takes multiple read cycles and at least one write cycle.

- Data manipulation instructions operate on information being held in CPU registers. Data manipulation instructions often use inherent addressing.
- Branch instructions alter the flow of instruction execution. *Unconditional branches* always change the location in memory of where the CPU is executing instructions. *Conditional branches* only change the location of instruction execution if a status flag is asserted.
- Status flags are held in the condition code register and are updated by certain instructions. The most commonly used flags are the negative flag (N), zero flag (Z), two's complement overflow flag (V), and carry flag (C).

Exercise Problems

Section 11.1: Computer Hardware

11.1.1 What computer hardware subsystem holds the temporary variables used by the program?

11.1.2 What computer hardware subsystem contains fast storage for holding and/or manipulating data and addresses?

11.1.3 What computer hardware subsystem allows the computer to interface to the outside world?

11.1.4 What computer hardware subsystem contains the state machine that orchestrates the fetch-decode-execute process?

11.1.5 What computer hardware subsystem contains the circuitry that performs mathematical and logic operations?

11.1.6 What computer hardware subsystem holds the instructions being executed?

Section 11.2: Computer Software

11.2.1 In computer software, what are the names of the most basic operations that a computer can perform?

11.2.2 Which element of computer software is the binary code that tells the CPU which instruction is being executed?

11.2.3 Which element of computer software is a collection of instructions that perform a desired task?

11.2.4 Which element of computer software is the supplementary information required by an instruction such as constants or which registers to use?

11.2.5 Which class of instructions handles moving information between memory and CPU registers?

11.2.6 Which class of instructions alters the flow of program execution?

11.2.7 Which class of instructions alters data using either arithmetic or logical operations?

Section 11.3: Computer Implementation—An 8-bit Computer Example

11.3.1 Design the example 8-bit computer system presented in this chapter in Verilog with the ability to execute the three instructions LDA_IMM, STA_DIR, and BRA. Simulate your computer system using the following program that will continually write the patterns x"AA" and x"BB" to output ports port_out_00 and port_out_01:

```
initial
 begin
  ROM[0]  = LDA_IMM;
  ROM[1]  = 8'hAA;
  ROM[2]  = STA_DIR;
  ROM[3]  = 8'hE0;
  ROM[4]  = STA_DIR;
  ROM[5]  = 8'hE1;
  ROM[6]  = LDB_IMM;
  ROM[7]  = 8'hBB;
  ROM[8]  = STB_DIR;
  ROM[9]  = 8'hE0;
  ROM[10] = STB_DIR;
  ROM[11] = 8'hE1;
  ROM[12] = BRA;
  ROM[13] = 8'h00;
 end
```

11.3.2 Add the functionality to the computer model from 11.3.1 the ability to perform the LDA_DIR instruction. Simulate your computer system using the following program that will continually read from port_in_00 and write its contents to port_out_00:

```
initial
 begin
  ROM[0] = LDA_DIR;
  ROM[1] = 8'hF0;
  ROM[2] = STA_DIR;
  ROM[3] = 8'hE0;
  ROM[4] = BRA;
  ROM[5] = 8'h00;
 end
```

11.3.3 Add the functionality to the computer model from 11.3.2 the ability to perform the instructions LDB_IMM, LDB_DIR, and STB_DIR. Modify the example programs given in exercise 11.3.1 and 11.3.2 to use register B in order to simulate your implementation.

11.3.4 Add the functionality to the computer model from 11.3.3 the ability to perform the addition instruction ADD_AB. Test your addition instruction by simulating the following program. The first addition instruction will perform x"FE" + x"01 = x"FF" and assert the negative (N) flag. The second addition instruction will perform x"01 + x"FF" = x"00" and assert the carry (C) and zero (Z) flags. The third addition instruction will perform x"7F" + x"7F" = x"FE" and assert the two's complement overflow (V) and negative (N) flags.

```
initial
 begin
  ROM[0]  = LDA_IMM; //-- test 1
  ROM[1]  = 8'hFE;
  ROM[2]  = LDB_IMM;
  ROM[3]  = 8'h01;
  ROM[4]  = ADD_AB;
  ROM[5]  = LDA_IMM; //-- test 2
  ROM[6]  = 8'h01;
  ROM[7]  = LDB_IMM;
  ROM[8]  = 8'hFF;
  ROM[9]  = ADD_AB;
  ROM[10] = LDA_IMM; //-- test 3
  ROM[11] = 8'h7F;
  ROM[12] = LDB_IMM;
  ROM[13] = 8'h7F;
  ROM[14] = ADD_AB;
  ROM[15] = BRA;
  ROM[16] = 8'h00;
 end
```

11.3.5 Add the functionality to the computer model from 11.3.4 the ability to perform the *branch if equal to zero* instruction BEQ. Simulate your implementation using the following program. The first addition in this program will perform x"FE" + x"01 = x"FF" (Z = 0). The subsequent BEQ instruction should NOT take the branch. The second addition in this program will perform x"FF" + x"01 = x"00" (Z = 1) and SHOULD take the branch. The final instruction in this program is a BRA that is inserted for safety. In the event that the BEQ is not operating properly, the BRA will set the program counter back to x"00" and prevent the program from running away.

```
initial
 begin
   ROM[0]  = LDA_IMM; //-- test 1
   ROM[1]  = 8'hFE;
   ROM[2]  = LDB_IMM;
   ROM[3]  = 8'h01;
   ROM[4]  = ADD_AB;
   ROM[5]  = BEQ;      //--NO branch
   ROM[6]  = 8'h00;

   ROM[7]  = LDA_IMM; //-- test 2
   ROM[8]  = 8'h01;
   ROM[9]  = LDB_IMM;
   ROM[10] = 8'hFF;
   ROM[11] = ADD_AB;
   ROM[12] = BEQ;      //-- Branch
   ROM[13] = 8'h00;

   ROM[14] = BRA;
   ROM[15] = 8'h00;
 end
```

Chapter 12: Floating-Point Systems

This chapter introduces the concept of floating-point numbers and how to build systems that use floating-point representations to perform mathematical operations. Floating-point numbers represent *real* numbers (i.e., ones that have a fractional component) using an encoding technique that first converts the original number into scientific notation (SN) and then encodes the three distinct fields of the notation with binary codes (e.g., the sign, the mantissa, and the exponent). The standard that guides the encoding/decoding of floating-point numbers is IEEE 754. This standard is used in all modern computers. This chapter begins with a detailed explanation of the IEEE 754 encoding approach including the anatomy, the algorithms for converting between decimal and floating-point formats, range, and precision. The chapter then discusses arithmetic using floating-point numbers and then moves into Verilog modeling for floating-point systems. The goal of this chapter is to provide an understanding of floating-point numbers and the basic principles of how to begin building digital systems that use floating-point numbers in Verilog.

Learning Outcomes—After completing this chapter, you will be able to:

12.1 Describe the IEEE 754 standard including the encoding algorithm, data types, range, and precision of floating-point representation
12.2 Perform conversions between decimal and IEEE 754 formats by hand
12.3 Perform basic arithmetic with floating-point numbers by hand
12.4 Design systems that use floating-point numbers for mathematical operations in Verilog

12.1 Overview of Floating-Point Numbers

12.1.1 Limitations of Fixed-Point Numbers

The first step in understanding the need for floating-point numbers is to look at the limitations of *fixed-point* representation of real numbers. In this encoding approach, a fixed number of bits are used to represent a real number, and the radix point is *fixed* within the number. The radix point can be placed anywhere within the binary number, but once this location is decided, it cannot move. If the radix point is placed directly in the middle of the bit field, then the upper half of the bits represent the whole part of the number and the lower half of the bits represent the fractional part. Fixed-point encoding is straightforward and has the advantage that it can use arithmetic circuits built for integer numbers. Figure 12.1 provides an overview of fixed-point encoding in binary.

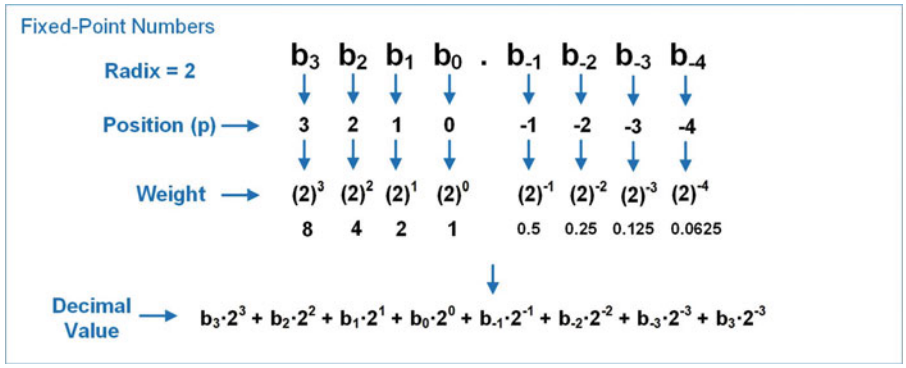


Fig. 12.1
Overview of fixed-point representation for real numbers

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2_12

The main disadvantage of fixed-point encoding is that once the location of the radix point is determined, it can never be moved. This can lead to suboptimal encoding when a system is built that operates on smaller numbers (i.e., fractional numbers that are close to zero) where it would be better to have the radix point located further to the left in the binary representation. Locating the radix point further to the left would allow more bits in the field to be used for fractional accuracy. Conversely, this also leads to suboptimal encoding when a system is built to operate on larger numbers (i.e., those that do not need a great deal of fractional accuracy) where it would be better to have the radix point located further to the right in the binary representation. Fixed-point encoding leads to a generally suboptimal use of the bits within the field, in addition to a limited range of values it can encode. As arithmetic is performed on fixed-point numbers, accuracy tends to be diminished as fractional values are truncated.

12.1.2 The Anatomy of a Floating-Point Number

An alternative to fixed-point encoding is a floating-point approach in which the radix point can be dynamically moved within the bit field based on a location value encoded into the number itself. This is the motivation for floating-point representation. The term *float* refers to the radix point *floating* within the number based on a value encoded in the word. The general concept of floating-point representation is to first convert the number into binary SN and then encode the three parts of the number separately. The first field encoded is the *sign* of the number where a 0 represents a positive number and 1 represents a negative number. The second field encoded is the *mantissa*, which is part of the SN number that is separate from the exponent multiplier portion. And finally, the *exponent* of the multiplier is encoded. The three distinct fields of the SN number are then put into a single binary word with the sign bit as the MSB followed by the exponent field and then the mantissa. Figure 12.2 shows the general overview of floating-point representation. Note that this encoding approach yields a *signed magnitude* number rather than a two's complement encoding approach.

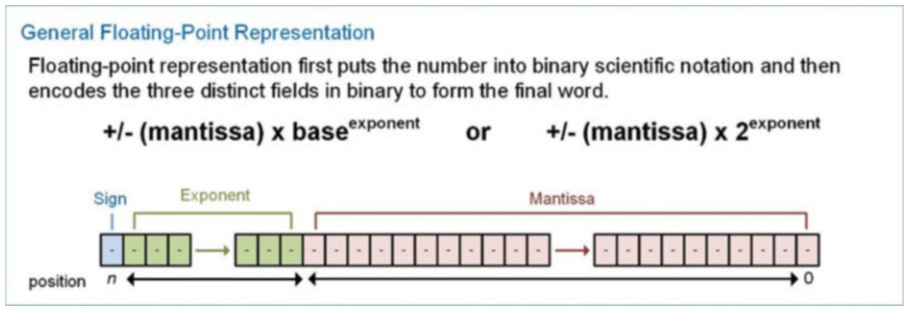


Fig. 12.2
General format of a floating-point number

12.1.3 The IEEE 754 Standard

When the adoption of integrated circuit-based microprocessors began to accelerate in the 1970s, work began on how to effectively implement floating-point numbers and arithmetic circuitry to accompany the new CPUs. As is always the case, competing ideas for how to implement floating-point numbers emerged, and incompatibility between different manufactures presented a barrier for wide-scale interoperability. In order to create a consistent approach for encoding floating-point numbers, an IEEE standard was created. The *IEEE 754 Standard for Floating-Point Arithmetic* was first released in 1985 and defined a variety of formats and special types for floating-point numbers. The standard was updated in 2008 and again in 2019 with minor revisions. The IEEE 754 standard is the most widely used approach for encoding floating-point numbers. While the current IEEE 754 standard has grown to include many different types, special functions, and operations, the most commonly used set of features are as follows:

- Encoding for a *single-precision* (32-bits) and *double-precision* (64-bits) floating-point numbers
- Mandatory mathematical operations support (add, subtract, multiply, divide, square root, fused-multiply-add)
- Conversions to/from IEEE 754 encoding to other formats (integer, character, hex character)
- Representation of non-numbers (e.g., positive and negative infinity, positive and negative zero, and not-a-number to indicate exceptions)
- Rounding procedures (round to nearest value, round toward infinity, round toward negative infinity, round toward zero)

Not every system that uses IEEE 754 implements the full set of features defined in the standard. The number of features implemented are related to the available computing hardware resources. Today, floating-point implementations range from small microcontrollers that only support limited floating-point usage all the way to high-end servers that implement the complete standard. The remaining sections in this chapter will present the most commonly used features from the IEEE standard.

12.1.4 Single-Precision Floating-Point Representation (32-Bit)

The IEEE 754 standard defines an encoding scheme that uses 32-bits and is referred to as a *single-precision* number. This is formally named **binary32** in IEEE 754, but more commonly referred to as an FP32 number, or simply a *float*. In a single-precision number, one bit is used for the sign bit (bit position 31), eight bits are allocated for the exponent (bits 30 → 23), and 23 bits are allocated for the mantissa (22 → 0). Figure 12.3 shows the anatomy of an IEEE single-precision number.

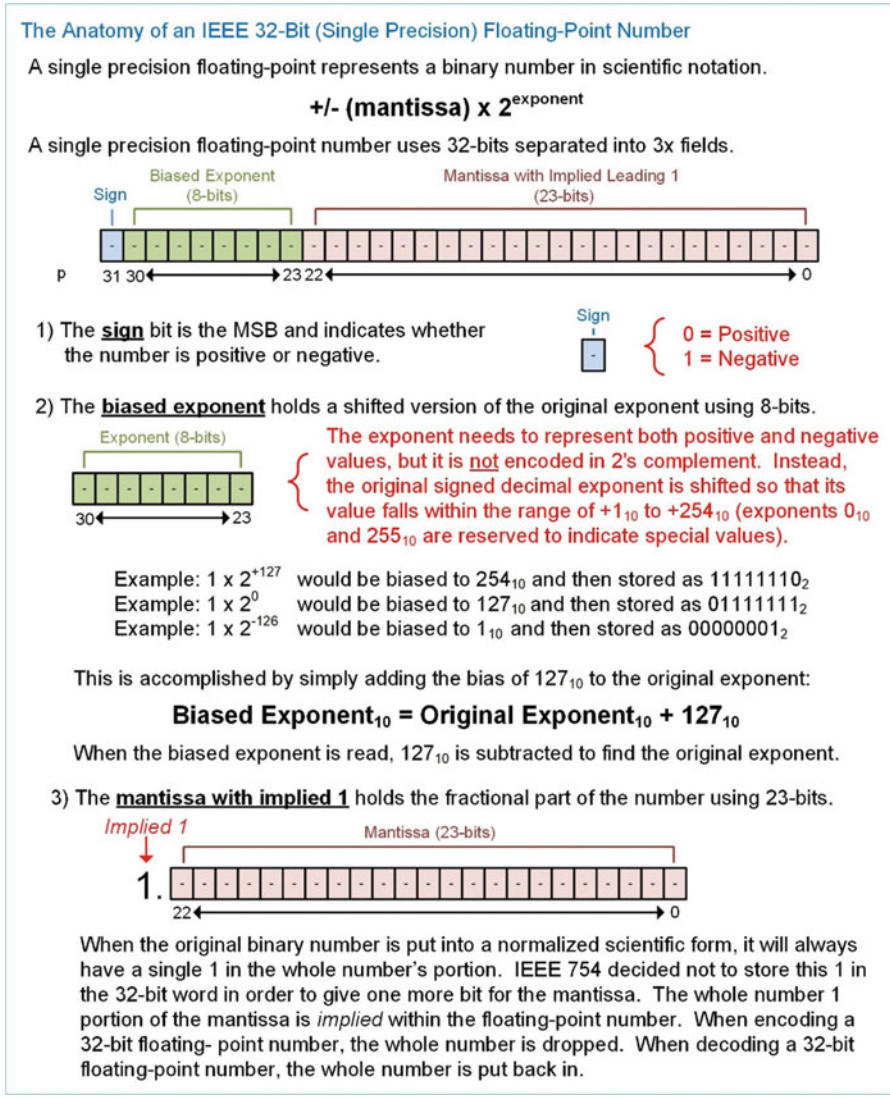


Fig. 12.3
The anatomy of an IEEE 32-bit (single-precision) floating-point number

The sign bit is encoded using the scheme of 0 = positive and 1 = negative. An important concept of single-precision numbers is that the sign bit is not treated the same as in two's complement encoded numbers where it is used as part of the encoding scheme. In single-precision numbers, the sign bit is a stand-alone indicator of whether the number is positive or negative. This is technically a *signed magnitude* representation approach.

The 23 mantissa bits represent the significant bits of the number separate from the exponent multiplier. It is common to see the mantissa referred to as the *significand* for this reason. IEEE 754 uses the concept of an *implied 1*, which takes advantage of the fact that in binary SN, a number can always be put into a form where the whole number to the left of the radix point will always be 1. Since this is always true, IEEE 754 does not include the whole number portion of the mantissa in the 23-bits of the field. This gives one additional bit of precision in the final encoded number. When encoding the

number, the implied 1 is not included in the binary32 word. When decoding the number, the implied 1 is added back to the mantissa to form the original value. When a number is put into binary SN with a single 1 in the whole number's position, it is said to be *normalized*. Since the mantissa in the IEEE 754 field represents values to the right of the radix point (without considering the exponent), it will often be called the *fractional* part of the number.

The eight exponent bits represent the decimal value of the exponent in the binary SN. IEEE 754 decided not to use two's complement encoding in the exponent bits to make some mathematical operations easier and instead only encode unsigned numbers. To encode both positive and negative exponents, the original exponent (which can be positive or negative) is *biased* by adding 127_{10} to it. This essentially *shifts* the original exponent from values that can range from -126_{10} to $+127_{10}$ to an unsigned range of 1_{10} to 254_{10}. This results in a *biased exponent* that is encoded in binary and then stored in the final field. When decoding the exponent, the bias of 127_{10} is simply subtracted from the biased exponent to find the original exponent. The IEEE 754 standard reserves the 0_{10} and 255_{10} exponent codes for special functions such as +/−infinity and +/− zero.

The *range* of a single-precision IEEE number is related to the number of bits in its exponent. The range of floating-point numbers is very large and gives it the ability to represent scientific quantities that fixed-point numbers cannot achieve with the same number of bits. Figure 12.4 shows the largest and smallest numbers possible in normal IEEE 754 single-precision encoding given the constraint that the exponent cannot use the reserved codes 0_{10} and 255_{10}.

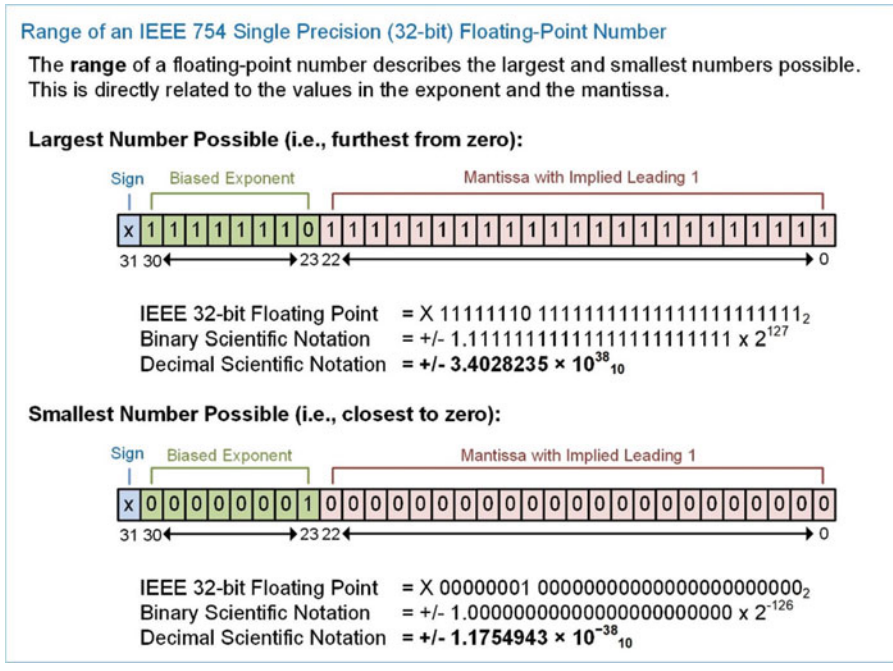


Fig. 12.4
Range of an IEEE 754 single-precision (32-bit) floating-point number

The *precision*, or accuracy, of a single-precision IEEE number is related to the number of bits in its mantissa. Precision is the biggest weakness of floating-point numbers and is most noticeable in 32-bit numbers. The precision of a binary32 value is dictated by the number of mantissa bits available to encode significant figures. The 24 mantissa bits of a binary32 value corresponds to ~7 significant figures in decimal. The loss of precision can become noticeable when performing continual mathematical operations where the error compounds due to LSBs of the result being truncated to fit into the 23-bits available in the mantissa. Floating-point numbers are not guaranteed to represent every possible number exactly. Only numbers that can be represented with the number of bits in the mantissa and exponent are guaranteed to be exact. All other numbers are representations that get as close as possible to the actual value. IEEE 754 defines rounding types that can be used for inexact values. Figure 12.5 shows the precision of an IEEE 754 single-precision floating-point number.

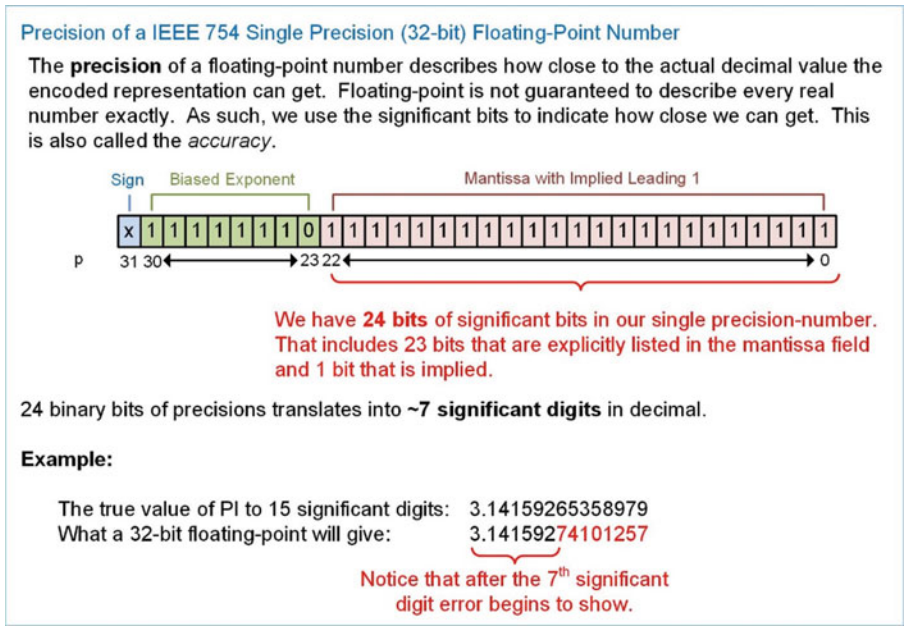


Fig. 12.5
Range of an IEEE 754 single-precision (32-bit) floating-point number

IEEE 754 also supports a special class of numbers called *subnormal*, or denormalized numbers, that allows a representation of values smaller than what can be represented using normal encoding. In the subnormal special case, the implied 1 of the mantissa is not used in the calculation of the value and is instead replaced with a 0 in the whole number position. A subnormal value is indicated by setting the exponent equal to 0. Subnormal numbers can have a positive or negative sign. Subnormal numbers can take on mantissa values from 00. . .01 to 11. . .11. The mantissa value of 00. . .00 with an exponent of 0 is reserved for a different special value that will be covered later. Supporting subnormal values is optional and often associated with how much hardware can be allocated for the floating-point system.

If a calculation results in a value that is larger than an IEEE 754 normalized code can represent, it is called *overflow*. If a calculation results in a value that is smaller than an IEEE 754 normalized code can represent, it is called *underflow*. With these terms, we now have all the definitions needed for the complete number line of a binary32 floating-point representation, which is shown in Fig. 12.6.

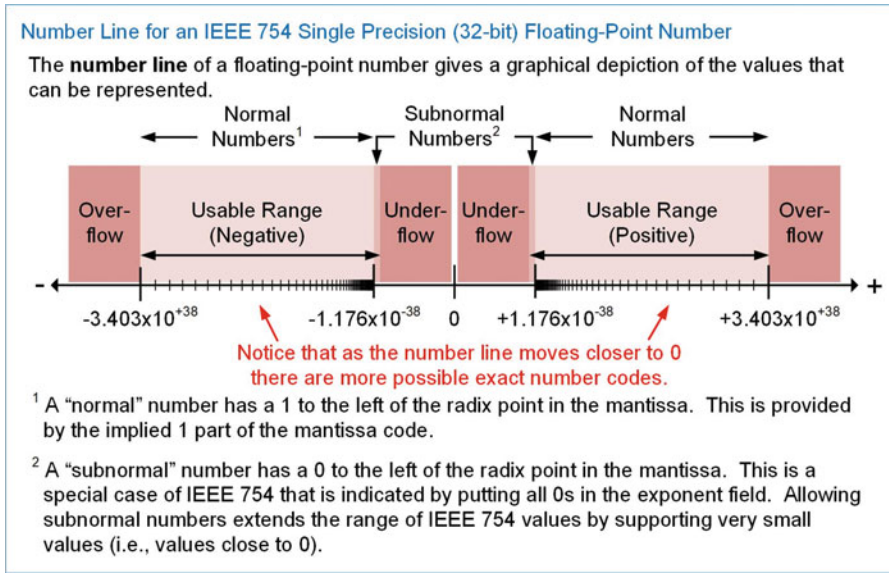


Fig. 12.6
Number line for an IEEE 754 single-precision (32-bit) floating-point number

12.1.5 Double-Precision Floating-Point Representation (64-Bit)

The IEEE 754 standard also defines an encoding scheme for a 64-bit floating-point format that is referred to as a *double-precision* number. This is formally named **binary64** in IEEE 754, but more commonly referred to as an FP64 number, or simply a *double*. In a double-precision number, one bit is used for the sign bit (bit position 63), 11 bits are allocated for the exponent (bits 62 → 52), and 52 bits are allocated for the mantissa (51 → 0). Figure 12.7 shows the anatomy of a 64-bit double-precision floating-point number in IEEE 754.

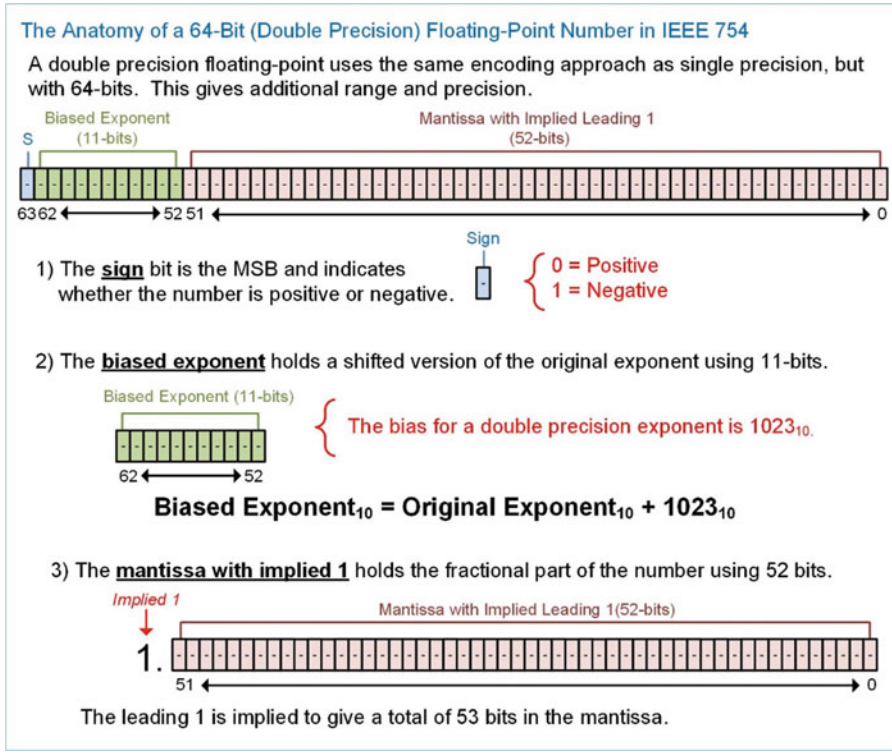


Fig. 12.7
The anatomy of an IEEE 64-bit (double-precision) floating-point number

The double-precision format gives an extremely wide range of numbers due to increasing the number of bits used in the biased exponent. Figure 12.8 shows the range of an IEEE 754 double-precision number.

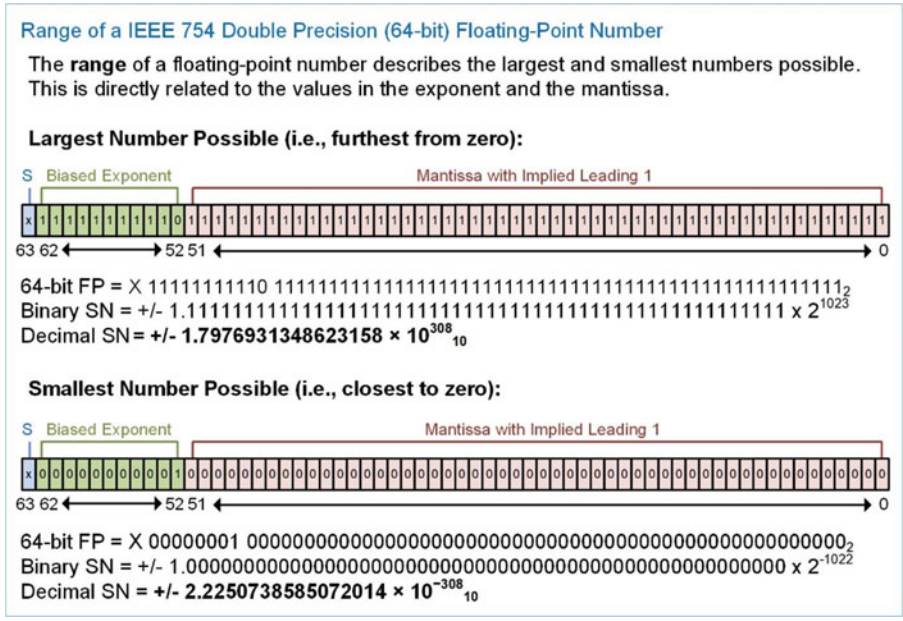


Fig. 12.8
Range of an IEEE 754 double-precision (64-bit) floating-point number

The double-precision encoding technique also addresses the precision issue that is associated with the single-precision format. By increasing the mantissa field to 52 bits, an accuracy of ~16 decimal digits is achieved. Figure 12.9 shows the precision of an IEEE 754 double-precision number.

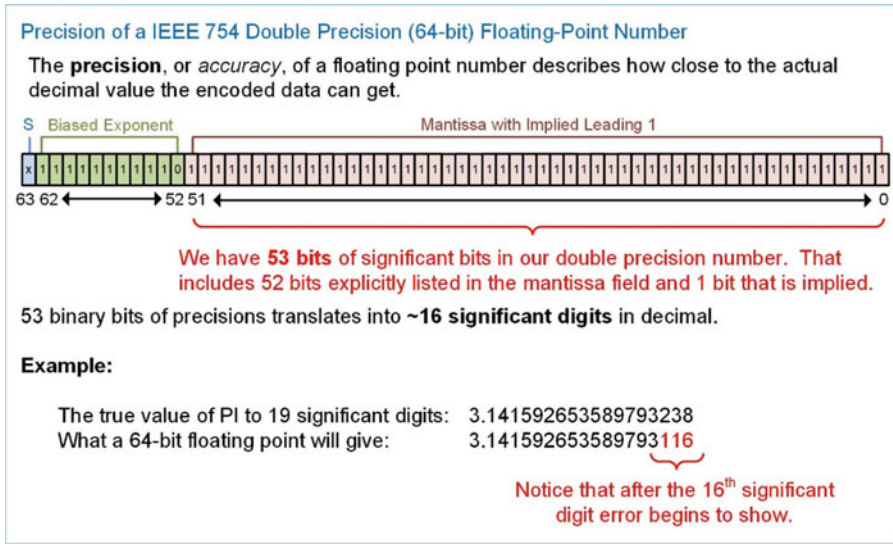


Fig. 12.9
Precision of an IEEE 754 double-precision (64-bit) floating-point number

Figure 12.10 gives the number line of a binary64 floating-point representation.

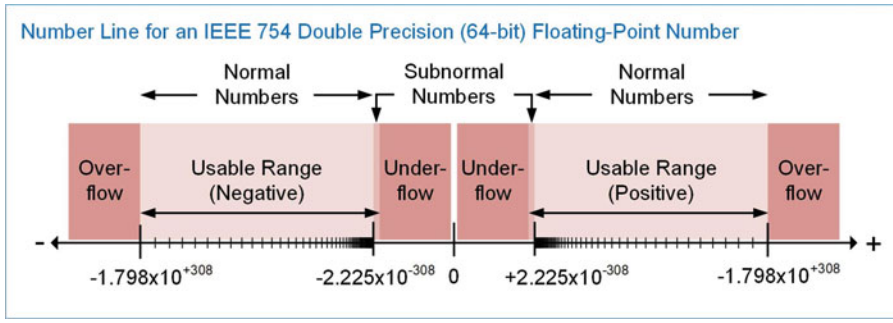


Fig. 12.10
Number line for an IEEE 754 double-precision (64-bit) floating-point number

12.1.6 IEEE 754 Special Values

IEEE 754 defines a set of other special values that are needed for implementation of a useful arithmetic system. IEEE 754 provides unique codes for **+0** and **−0**. To indicate zero, both the exponent and mantissa fields hold all 0s. The sign bit is used normally to indicate whether zero is positive or negative. The code for +0 is considered *exact zero* and used as the result of operations that result in a true zero value (i.e., n-n). While +0 and −0 have unique codes, they are evaluated as equal during compares.

IEEE 754 provides two unique codes for **+infinity** and **−infinity**. Infinity is indicated by setting the exponent to all 1s and the mantissa to all 0s. The sign bit is used normally to indicate whether infinity is positive or negative.

IEEE 754 provides an exception value called ***Not a Number*** (NaN) to indicate an invalid result. A NaN is indicated by an exponent of all 1s and a nonzero mantissa. There are two supported NaN types, a *quiet NaN* (qNaN) and a *signaling NaN* (sNaN). If the first bit of the mantissa is a 1, it indicates a qNaN. A qNaN propagates through an operation yielding a result of qNaN without signaling an exception. If the first bit of the mantissa is a 0, it indicates a sNaN. A sNaN signals an exception immediately upon use. The remaining bits in the mantissa for a sNaN are called the *payload* and hold user-defined exception diagnostic information.

Figure 12.11 gives a list of the IEEE 754 special values and their associated codes.

IEEE 754 Special Values

In IEEE 754, the exponent values of 00...00 (i.e., all 0s) and 11...11 (i.e., all 1s) are reserved to indicate *special values*. When these exponent values are used, the sign bit and mantissa provide additional information about the special value being represented.

Special Value	Sign	Exponent	Mantissa	Notes
+0	0	00...00	00...00	Also called "exact zero"
-0	1	00...00	00...00	0+ and 0- are distinct values, but are evaluated as equal during a compare.
Subnormal (Positive)	0	00...00	00...01 to 11.11	Subnormal numbers use a 0 in place of the implied 1's position in the mantissa. This provides extremely small values to be encoded surrounding 0. These are also referred to as *denormalized* numbers.
Subnormal (Negative)	1	00...00	00...01 to 11.11	
+Infinity	0	11...11	00...00	-
-Infinity	1	11...11	00...00	-
Not a Number (NaN)	x	11...11	00...01 to 01..11	If the non-zero mantissa begins with a 0, it indicates a *signaling NaN* (sNaN). The remaining mantissa bits are referred to as the *payload* and contain user-defined information about the exception.
			10...0 to 11..11	If the non-zero mantissa begins with a 1, it indicates a *quiet NaN* (qNaN). This is the default exception action.

Fig. 12.11
IEEE 754 special values

IEEE 754 also defines the results of operations on special values. If special values are supported, the codes of the inputs are checked prior to performing the operation using a dedicated decoder. If a special value is detected, it will then use the defined results in Fig. 12.12 for the result instead of using the software or circuitry designed to handle regular normal number inputs.

Results of Operations Using IEEE 754 Special Values

IEEE 754 defines the results of the following operations that use special values:

Operation	Result
n ÷ (+Infinity)[1]	+0
n ÷ (-Infinity)[1]	-0
n ÷ (+0)[1]	+Infinity
n ÷ (-0)[1]	-Infinity
(+/-Infinity) x (+/-Infinity)[2]	+/-Infinity
(+Infinity) + (+Infinity)	+Infinity
(-Infinity) + (-Infinity)	-Infinity
(+Infinity) - (+Infinity)	NaN
(-Infinity) + (+Infinity)	NaN
+/-0 x (+/-Infinity)	NaN
(+/-0) ÷ (+/-0)	NaN
(+/-Infinity) ÷ (+/-Infinity)	NaN
NaN[3]	NaN

[1] When the sign of n changes, it will change the sign of the result.

[2] The sign of the result follows the standard rules of multiplication (i.e., POSxPOS=POS, POSxNEG=NEG, NEGxPOS=NEG, NEGxNEG=POS).

[3] The most significant bit of the mantissa indicates whether the NaN is *signaling* (0) or *quiet* (1). A quiet NaN (qNaN) propagates through operations without signaling exceptions and produces a result of qNaN. A signaling NaN (sNaN) signals an exception when used.

Fig. 12.12
Results of operations using IEEE 754 special values

12.1.7 IEEE 754 Rounding Types

The IEEE 754 standard specifies a variety of rounding options for results that do not fall exactly into one of the possible floating-point codes. IEEE 754 also supports the use of *guard bits* to improve accuracy. Guard bits are additional bits added to the mantissa during an operation. After the operation, the result is rounded and then the guard bits are removed. Inexact numbers that are rounded shall have the same sign as the original unrounded number. NaNs are not rounded.

The first rounding approach is called ***round to nearest—ties to even***. This approach will round a number that falls within two exact representations to the closest numerical value. In the case that a number falls equally within two exact representations, the number is rounded to its even neighbor. The second rounding approach is called ***round to nearest—ties away from zero*** *(a.k.a., ties to away).* This approach also rounds a number that falls within two exact representations to the closest numerical value, but a number that falls equally within two exact representations will be rounded to the neighbor furthest from zero. An alternate description of this rounding approach is that a number exactly between two values will be rounded to the neighbor with the largest magnitude. The third rounding approach is called ***round toward zero*** and will always round toward the value's neighbor that is lower in magnitude (i.e., closer to zero). The fourth rounding approach is called ***round toward positive*** and will always round toward the more positive value (i.e., the neighbor to its right on the number line). The final rounding approach is called ***round toward negative*** and will always round toward the more negative value (i.e., the neighbor to its left on the number line). Figure 12.13 gives a summary of the IEEE rounding types.

IEEE 754 Rounding Types

IEEE 754 defines five rounding strategies that can be specified.

Mode	Example Values			
	+21.5	+22.5	-21.5	-22.5
Round to nearest, ties to even	+22.0	+22.0	-22.0	-22.0
Round to nearest, ties away from zero	+22.0	+23.0	-22.0	-23.0
Round toward 0	+21.0	+22.0	-21.0	-22.0
Round toward +∞	+22.0	+23.0	-21.0	-22.0
Round toward -∞	+21.0	+22.0	-22.0	-23.0

Fig. 12.13
IEEE 754 rounding types

12.1.8 Other Capabilities of the IEEE 754 Standard

The IEEE 754 standard contains other widths of floating-point numbers that are less commonly used. The **binary16** format uses 16-bits to encode the real number and is commonly referred to as a *half*. The **binary128** format uses 128-bits to encode the real number and is commonly referred to as a *quadruple*. The **binary256** format uses 256-bits to encode the real number and is commonly referred to as an *octuple*.

IEEE 754 supports three decimal-encoded floating-point formats called **decimal32**, **decimal64**, and **decimal128**. These encoding approaches are meant to match decimal rounding rules exactly, and they are primarily used in monetary calculations where even a small amount of rounding error can lead to a significant amount of money lost.

There are many nuances in the IEEE 754 standard when it comes to implementation. As such, a designer should consult the latest standard for the exact details of the encoding and decide which features are needed in their system. A designer also will need to decide whether the IEEE 754 features should be implemented in hardware, software, or a combination of both.

Concept Check

CC12.1 If using more bits for the IEEE 754 number encoding gives higher precision and wider range, why do not we just simply use the largest floating-point number allowed all the time?

A) The larger floating-point number takes more storage to hold the information.

B) It takes more circuitry to perform floating-point operations as the size of the number grows.

C) Some applications do not need the precision and range provided by larger floating-point numbers.

D) All of the above.

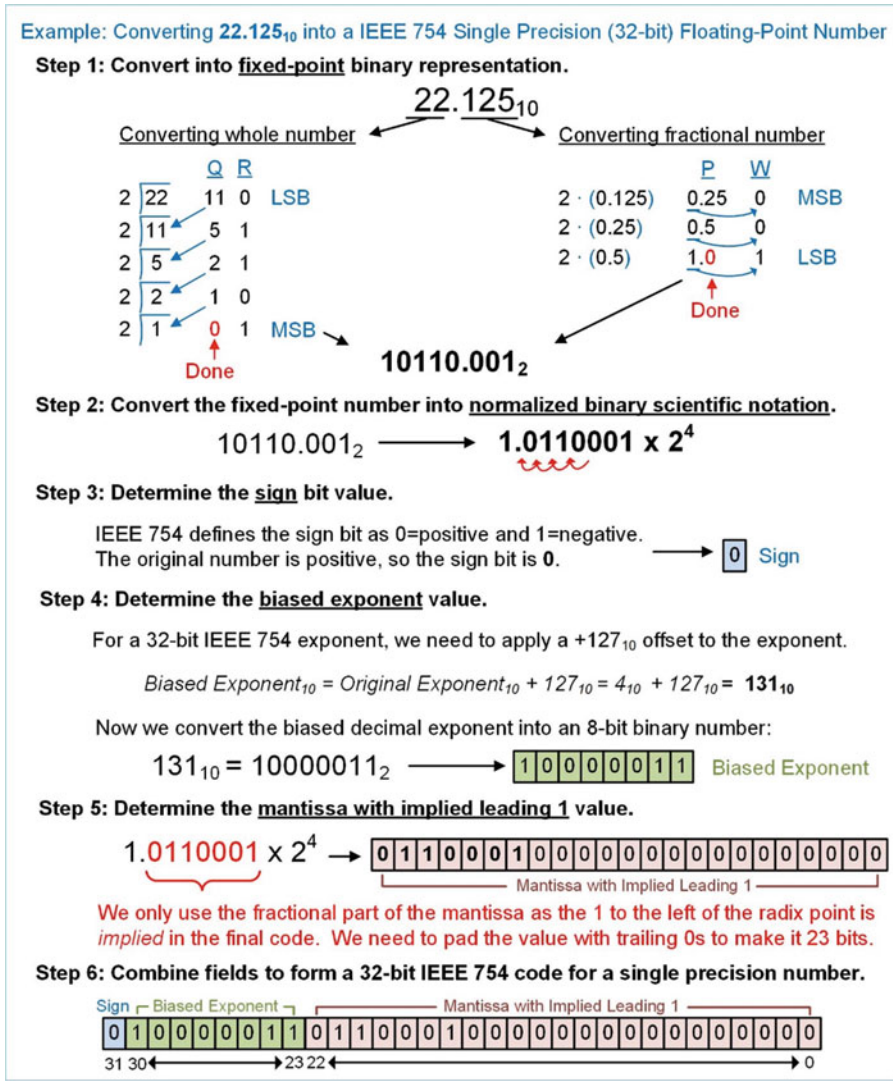
12.2 IEEE 754 Base Conversions

12.2.1 Converting from Decimal into IEEE 754 Single-Precision Numbers

Converting from a decimal number into a single-precision floating-point number by hand consists of these steps:

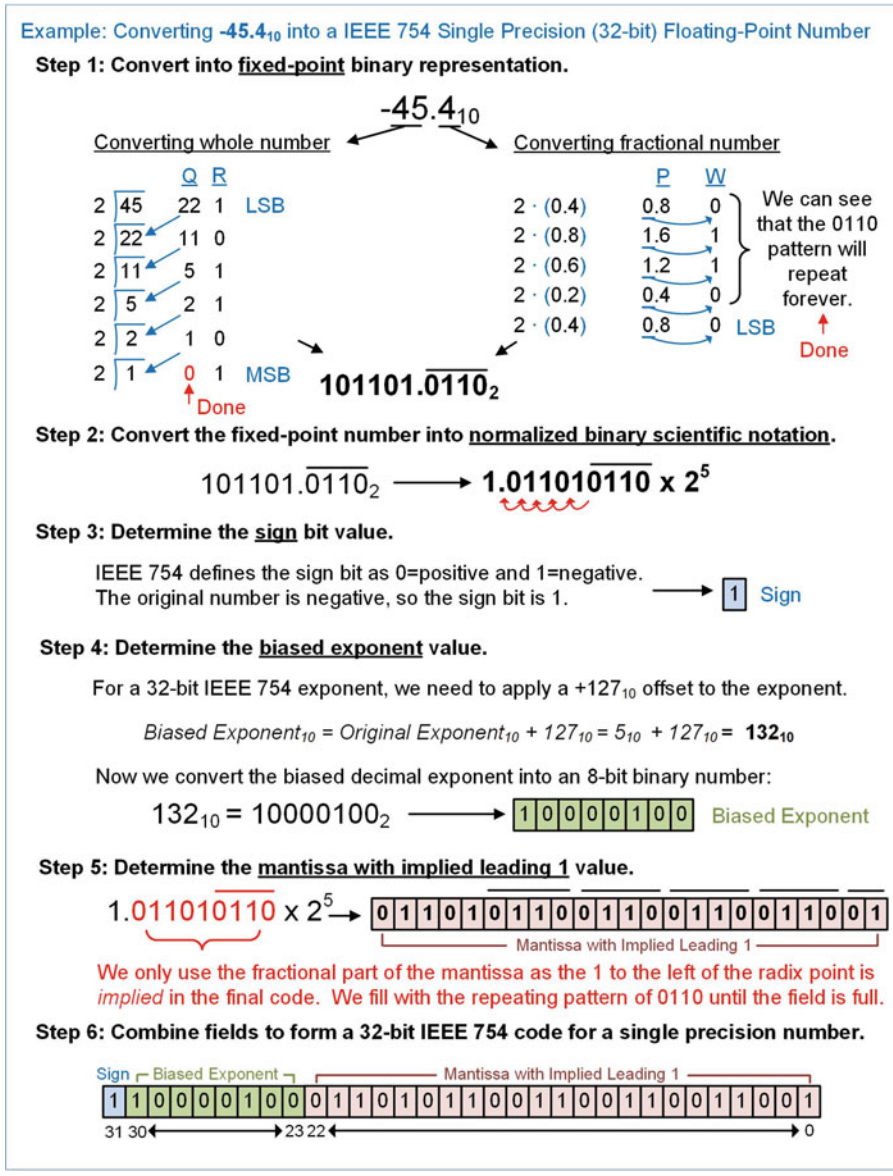
1. Convert the decimal number into a fixed-point binary representation
2. Convert the fixed-point number into normalized binary scientific notation
3. From the binary SN, determine the sign bit
4. From the binary SN, determine the biased exponent
5. From the binary SN, determined the mantissa with implied leading 1
6. Combine the three fields from steps 3–5 into the final 32-bit binary number

Example 12.1 shows the process of converting a $+22.125_{10}$ into a single-precision floating-point representation.



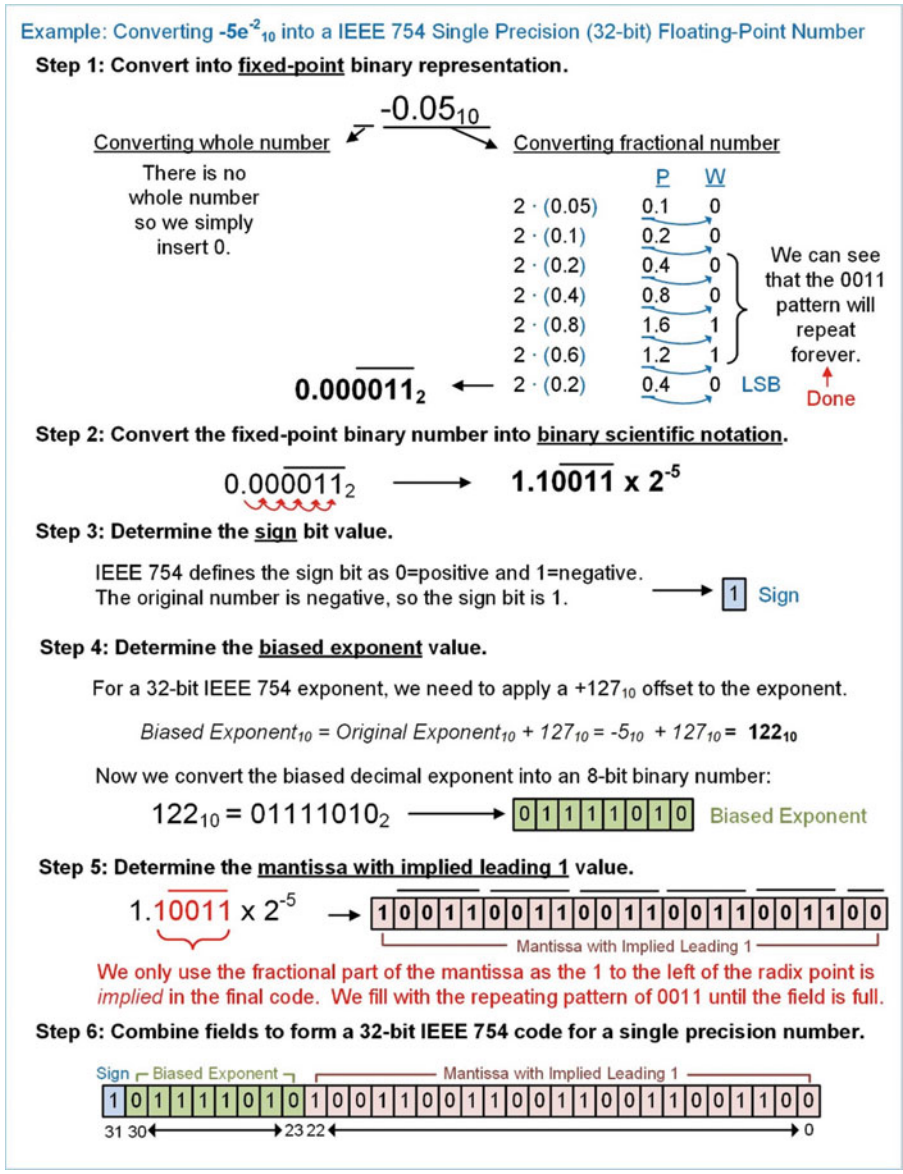
Example 12.1
Converting 22.125_{10} into IEEE 754 single-precision (32-bit) floating-point number

Example 12.2 shows another example of converting a decimal number ($-45.4._{10}$) into a single-precision floating-point representation; this time illustrating how to handle a fractional component that repeats, in addition to a number with a negative sign.



Example 12.2
Converting -45.4_{10} into IEEE 754 single-precision (32-bit) floating-point number

Example 12.3 shows another example of converting a decimal number (-0.05_{10}) into a single-precision floating-point representation; this time on a small decimal number with a repeating fractional component.



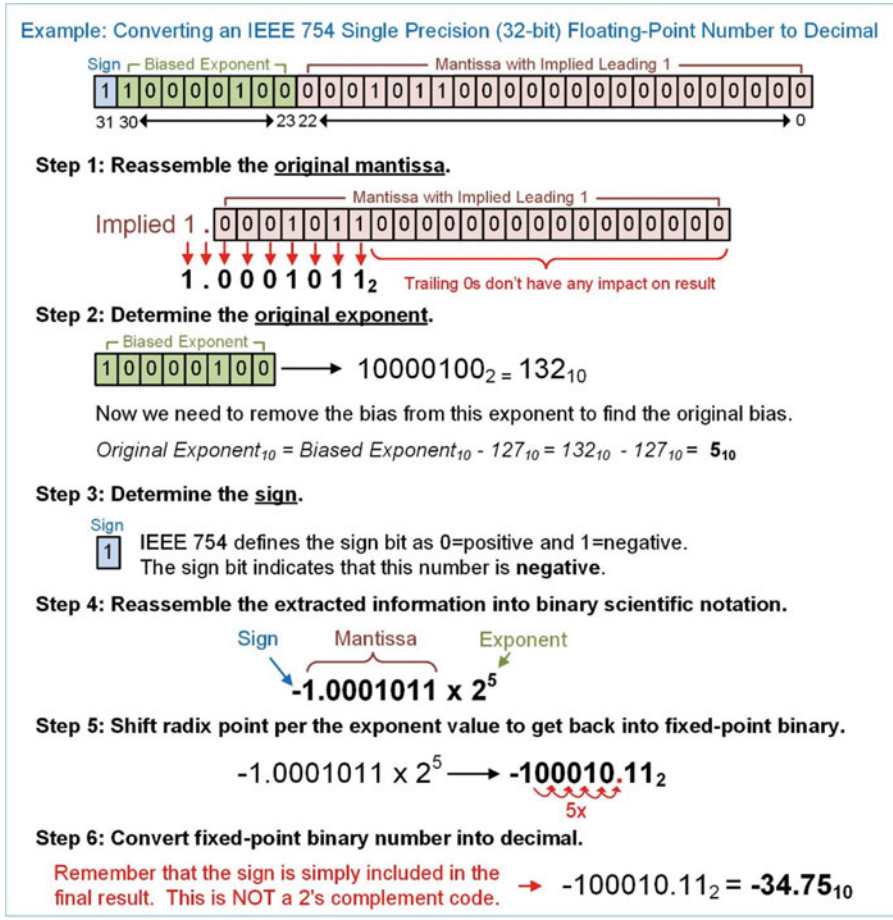
Example 12.3
Converting -0.05_{10} into IEEE 754 single-precision (32-bit) floating-point number

12.2.2 Converting from IEEE 754 Single-Precision Numbers into Decimal

Converting from an IEEE 754 single-precision number into decimal by hand is the reverse of the prior section:

1. Reassemble the original mantissa by adding back in the implied 1
2. Determine the decimal value of the original exponent from the biased exponent
3. Determine whether the number is positive or negative from sign bit
4. Assemble the extracted information from steps 1–3 into binary scientific notation
5. Shift the radix point in the binary SN per the exponent value to get back into fixed-point binary
6. Convert the fixed-point binary number to decimal

Example 12.4 shows an example of converting from an IEEE 754 single-precision floating-point number (-34.75_{10}) into decimal.



Example 12.4
Converting from an IEEE 754 single-precision number into decimal

Concept Check

CC12.2 Why is having the number in normalized binary scientific notation important for IEEE 754 encoding?

A) It avoids using engineering notation (i.e., when the exponents are multiples of 3), which has always bothered the scientists.

B) It allows the entire word to be used for the mantissa to achieve the greatest precision possible.

C) Being normalized guarantees that the bit to the left of the radix point will be a one and enables the use of the "implied 1" concept for the mantissa. This gives one extra bit for the mantissa, which leads to higher precision.

D) It minimizes the number of bits used for the exponent so that the exponent can be ignored.

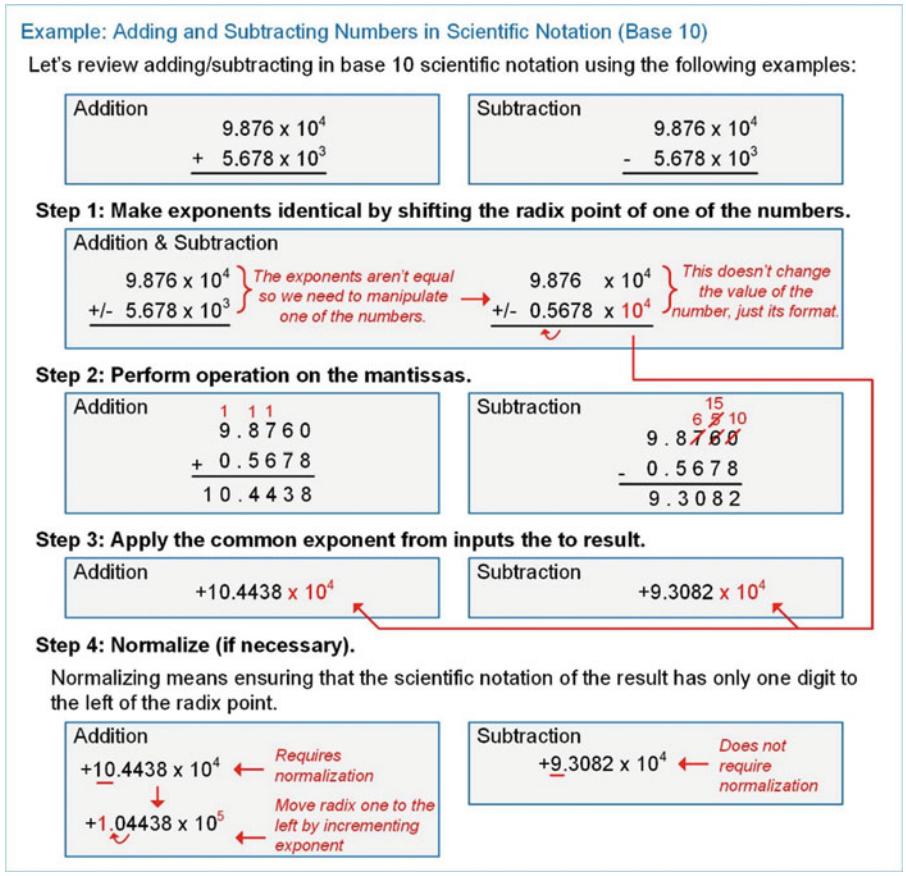
12.3 Floating-Point Arithmetic

12.3.1 Addition and Subtraction of IEEE 754 Numbers

The process of adding and subtracting IEEE 754 numbers follows a similar algorithm as when using base 10 SN. The first step is to ensure that both inputs have the same exponent by manipulating the exponent of one of the inputs until it matches the other. This has the result of moving the decimal point in the mantissa. The addition/subtraction is then performed on the mantissas of the inputs. The common exponent is then applied to the sum/difference. The final step is to *normalize* the result, which means ensuring that the result only has one nonzero digit to the left of the decimal point in the final SN. This is again accomplished by altering the exponent until the decimal point is in the desired location. These steps are summarized below:

1. Make exponents of the input arguments identical
2. Perform addition/subtraction on the mantissas
3. Apply the common exponent from step 1 to the result
4. Normalize the result (if necessary)

Example 12.5 shows an example of performing addition/subtraction on numbers in base 10 SN to illustrate these steps.



Example 12.5
Adding and subtracting numbers in scientific notation (base 10)

When doing addition/subtraction of IEEE 754 numbers, there are a few additional steps that need to be performed beyond the base 10 SN process. The first step is to convert the input arguments into normalized binary SN form. This step involves taking the original IEEE 754 encoded word and breaking it into its three distinct fields (sign, mantissa, and exponent). This step is called *unpacking*. During this step, the implied 1 is applied to the mantissa field extending its size by 1 bit and the bias is removed from the exponent.

The second step is to modify the input arguments to have the same exponents. This is accomplished by moving the radix point of one of the input arguments until it has an exponent that matches the other argument. Each time the radix point is moved, the exponent must be incremented or decremented accordingly. To minimize the loss of significant bits, this step is always performed on the input argument with the smaller exponent. A logical shift right is performed on this input argument, and the exponent is incremented accordingly until the exponents match. The logical shift right brings in 0s in the most significant position of the mantissa and can potentially shift the least significant positions out of the word. The bits lost in the lower significant position leads to *truncation error.* However, truncation error is less severe, in terms of precision, than losing bits in the most significant position as would be the case if the input argument with the larger exponent was altered using a logical shift left.

The third step involves addressing any negative inputs. Note that IEEE 754 uses a signed magnitude encoding approach for negative numbers. This means we either need to build adder/subtractor circuitry that can handle signed magnitude arithmetic, or convert the signed magnitude numbers into two's complement form and reuse arithmetic circuitry that already exists for two's complement integer operations. The most typical approach is the latter (i.e., convert the signed magnitude values into two's complement representations). To convert a signed magnitude number into two's complement representation, a 0 sign bit is first applied to the mantissa. This does not alter the value being held in the mantissa but does make the word one bit larger. If the input argument is positive as indicated by the value in the IEEE 754 sign bit, then no modifications are performed on the argument as it is already in two's complement form for a positive number. If the input argument is negative as indicated by the value in the IEEE 754 sign bit, then two's complement negation is performed to convert the input into its negative two's complement representation.

The fourth step is to perform addition/subtraction on the mantissas with the radix points aligned. Note that the radix points will already be aligned based on the logical shift rights performed in step 2. Remember that the addition/subtraction is being performed on two's complement numbers, which means if there is a resulting carry out, it is discarded.

The fifth step is to address the sign bit of the result. If the result is positive, store a 0 in the IEEE 754 sign field for the result and then remove the MSB of the result. Recall that the MSB of the result is an additional sign bit that was added in step 3 to convert the numbers from signed magnitude into two's complement representation. If the result is negative, store a 1 in the IEEE 754 sign field for the result, perform two's complement negation to convert the result into a magnitude, and then remove the MSB of the result. After the temporary sign bit is removed from the result, it will have the same number of bits as the original inputs' mantissas.

The sixth step is to apply the input arguments' exponent to the result. Recall that when we made the input arguments' exponents the same in step 2, we altered the smaller exponent to match the larger exponent. This means the larger number's exponent was not altered and can be used directly as the exponent for the result.

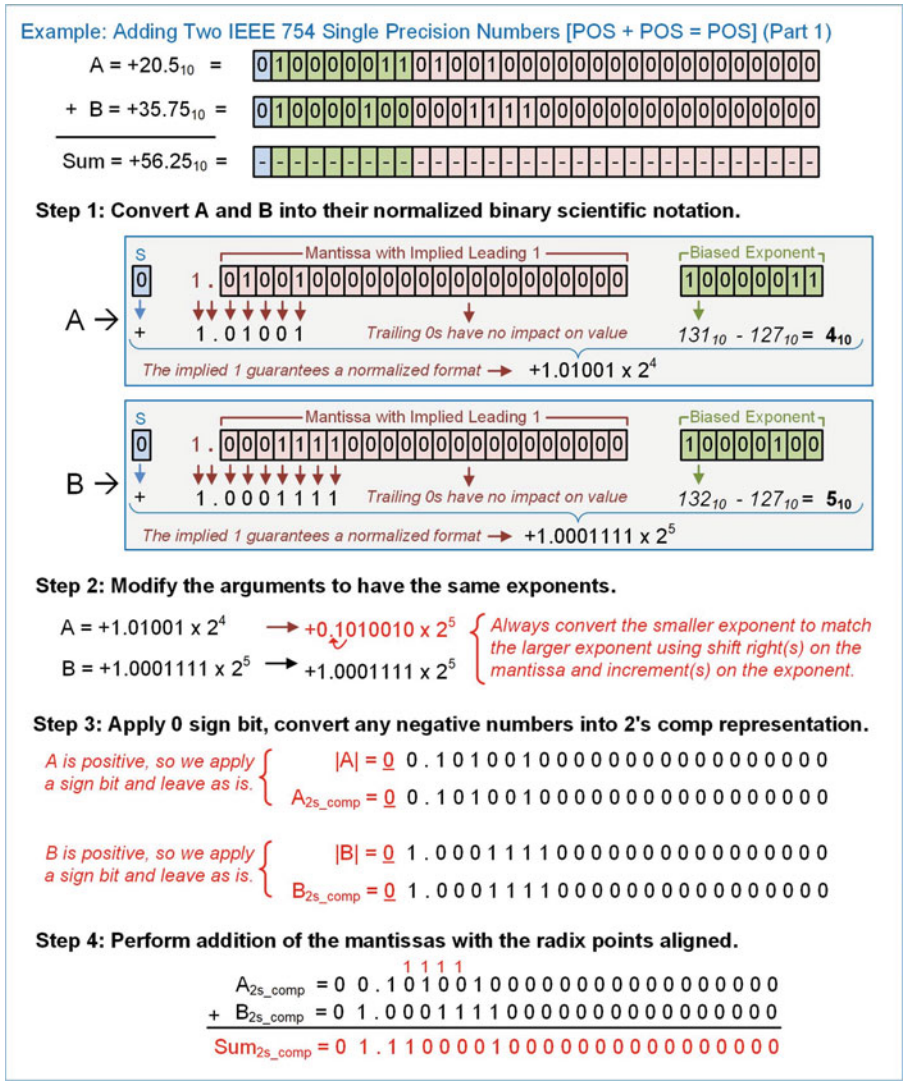
The seventh step is to normalize the result (if necessary). Recall that normalization of an IEEE 754 number means shifting the result so that there is a single one to the left of the radix point. Any shifts that are applied to the result to normalize it will be reflected by incrementing/decrementing the exponent. Note that this incrementing/decrementing can be performed on the exponent even if the bias has already been applied.

The final step is to convert the final binary SN into the binary32 encoding. This involves dropping the leading 1 of the mantissa and only storing the fractional part of the number into the 23-bits of the mantissa field. This also involves adding back in the bias to the exponent and storing it into the 8-bits of the exponent field. This final step is also called *packing*.

The steps to perform IEEE 754 addition are summarized below:

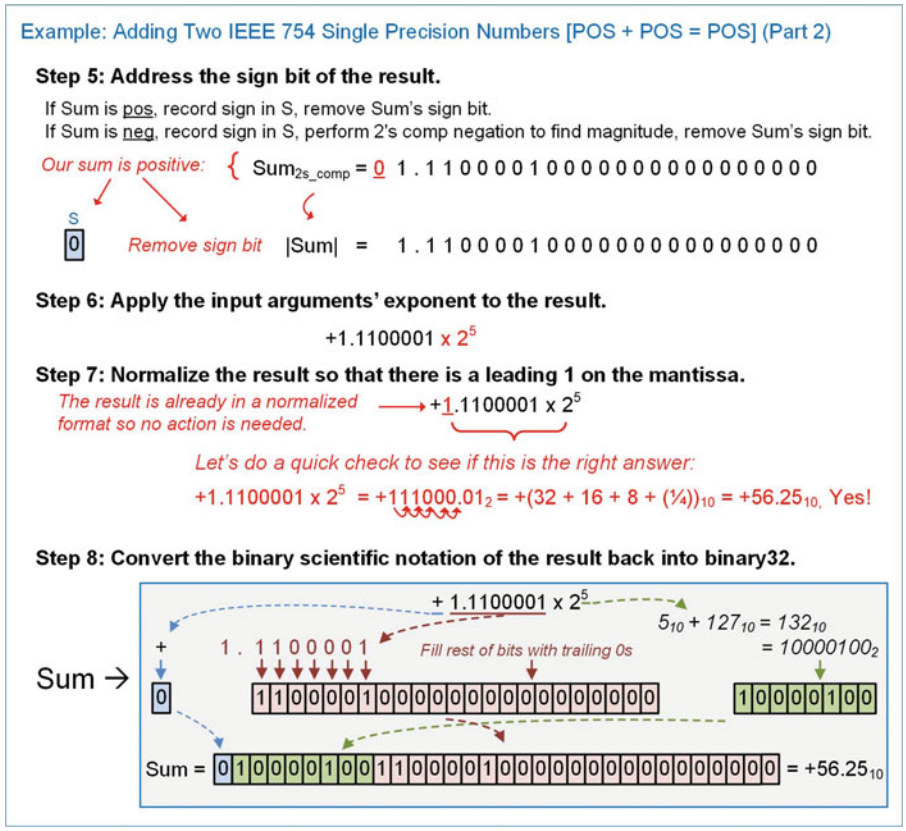
1. Convert input arguments from IEEE 754 into their normalized binary SN (also referred to as *unpacking* the original 32-bit word into its three individual parts)
2. Modify the arguments to have the same exponents
3. Apply a leading 0 sign bit to the arguments and convert any negative numbers into two's complement representation
4. Perform addition on the mantissas with the radix points aligned
5. Address the sign bit of the result
6. Apply the input arguments' exponent to the result
7. Normalize the result so that there is a leading 1 on the mantissa
8. Put all three components of the result back into binary32 format (i.e., pack the results into a 32-bit single-precision word)

Example 12.6 shows steps 1 through 4 of adding binary32 numbers. This example shows adding two positive binary32 numbers that result in a positive sum (i.e., POS + POS = POS).



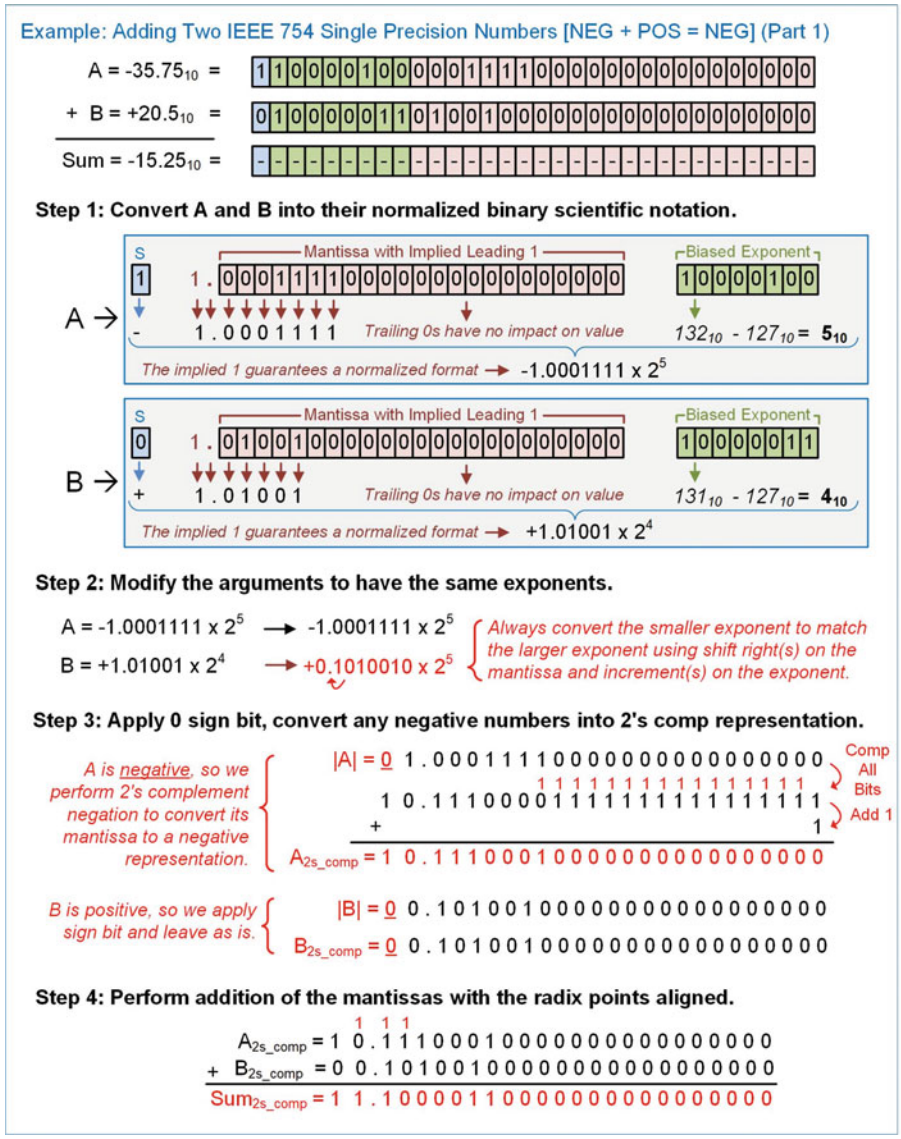
Example 12.6
Adding two IEEE 754 single-precision numbers [POS + POS = POS] (part 1)

Example 12.7 shows steps 5 through 8 of adding binary32 numbers (POS + POS = POS).



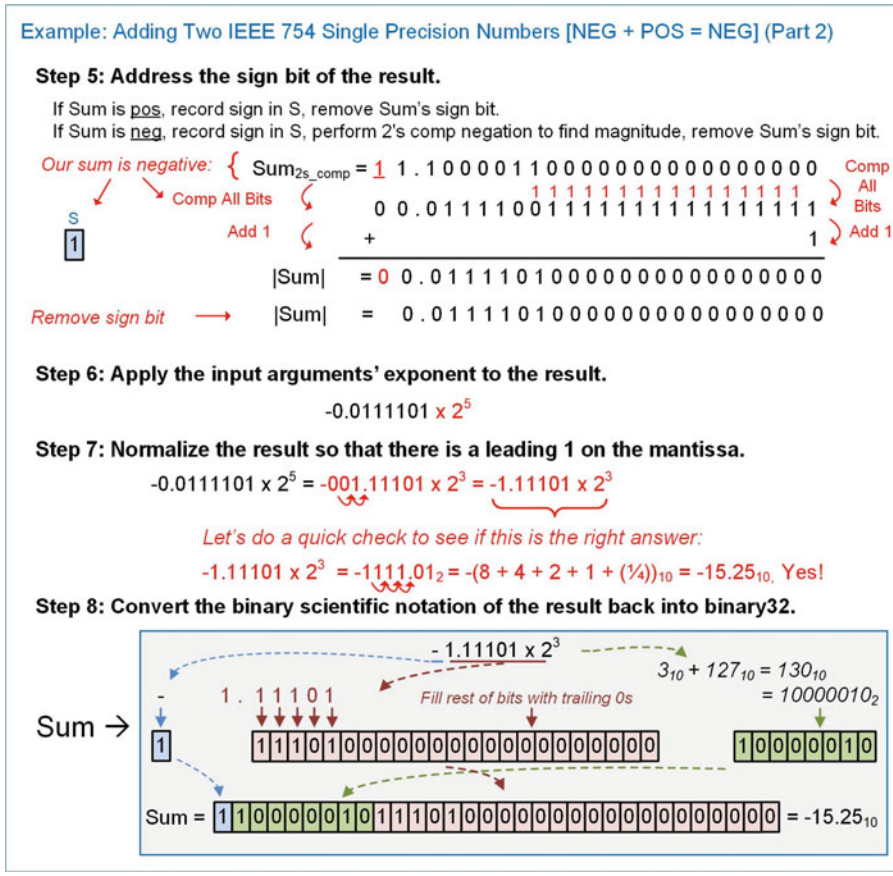
Example 12.7
Adding two IEEE 754 single-precision numbers [POS + POS = POS] (part 2)

Example 12.8 shows steps 1 through 4 of adding binary32 numbers. This example shows adding a negative number to a positive number that results in a negative sum (i.e., NEG + POS = NEG).



Example 12.8
Adding two IEEE 754 single-precision numbers [NEG + POS = NEG] (part 1)

Example 12.9 shows steps 5 through 8 of adding binary32 numbers (NEG + POS = NEG).

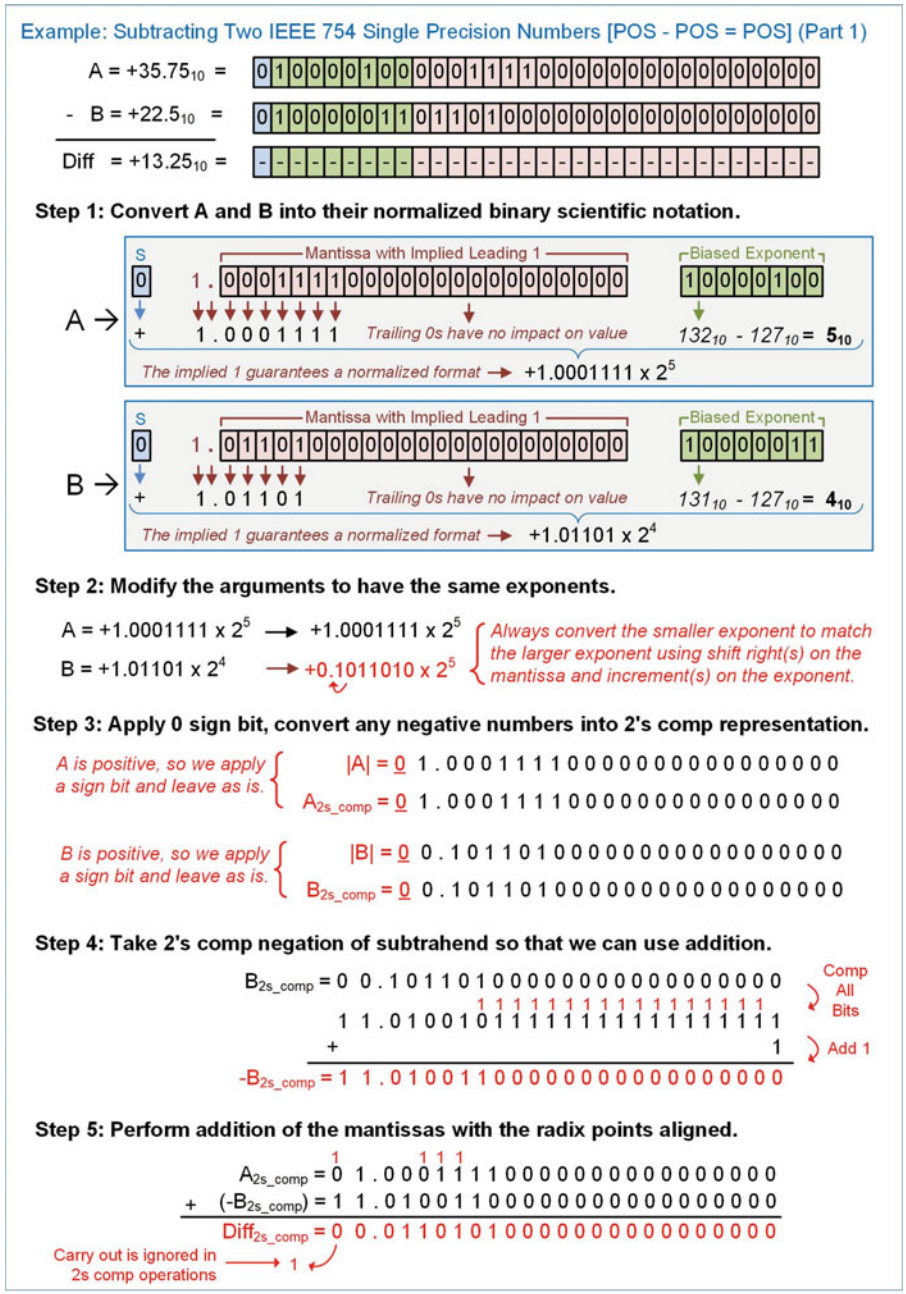


Example 12.9
Adding two IEEE 754 single-precision numbers [NEG + POS = NEG] (part 2)

Subtraction of IEEE 754 numbers can take advantage of two's complement negation of the subtrahend in order to reuse existing addition circuits within the system. This inserts an additional step into the prior addition process as follows:

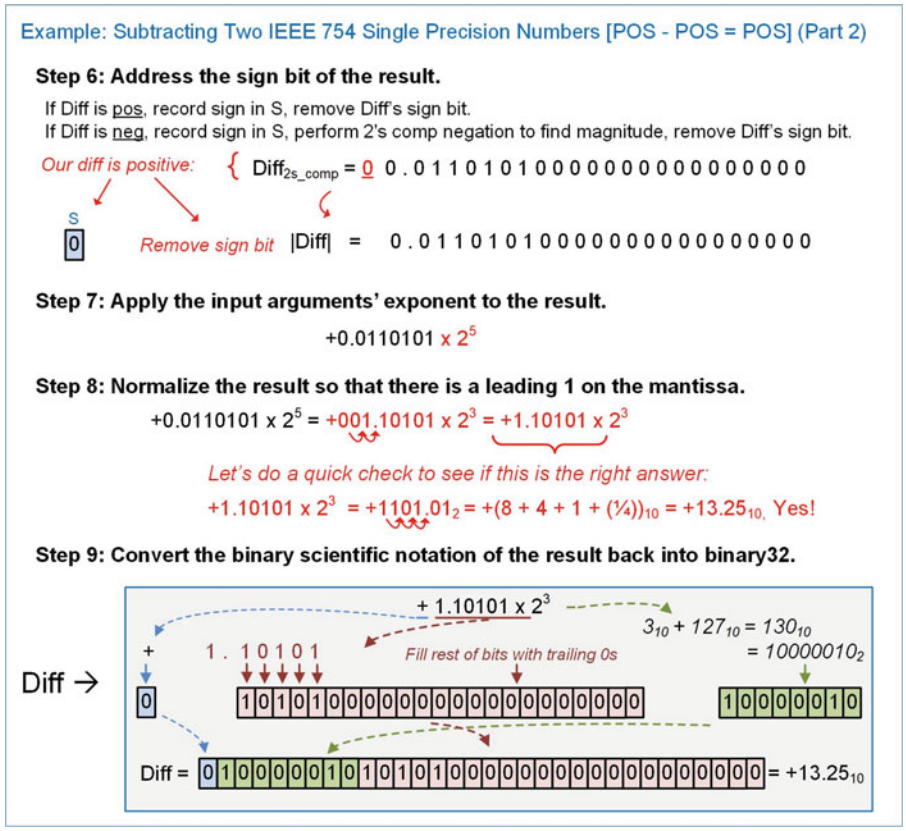
1. Convert input arguments from IEEE 754 into their normalized binary SN (i.e., unpack the inputs)
2. Modify the arguments to have the same exponents
3. Apply a leading 0 sign bit to the arguments, convert any negative numbers into two's complement representation
4. Perform two's complement negation on the subtrahend so that we can use addition *This is the extra step to perform subtraction that is included to the addition process described earlier*
5. Perform addition on the mantissas with the radix points aligned
6. Address the sign bit of the result
7. Apply the input arguments' exponent to the result
8. Normalize the result so that there is a leading 1 on the mantissa
9. Put all three components of the result back into binary32 format (i.e., pack the results into a 32-bit single-precision word)

Example 12.10 shows steps 1 through 5 of subtracting binary32 numbers. This example shows subtracting a positive number from a positive number that results in a positive difference (i.e., POS − POS = POS). This example illustrates how to use an adder and two's complement negation of the subtrahend to accomplish subtraction.



Example 12.10
Subtracting two IEEE 754 single-precision numbers [POS − POS = POS] (part 1)

Example 12.11 shows steps 6 through 9 of subtracting binary32 numbers.



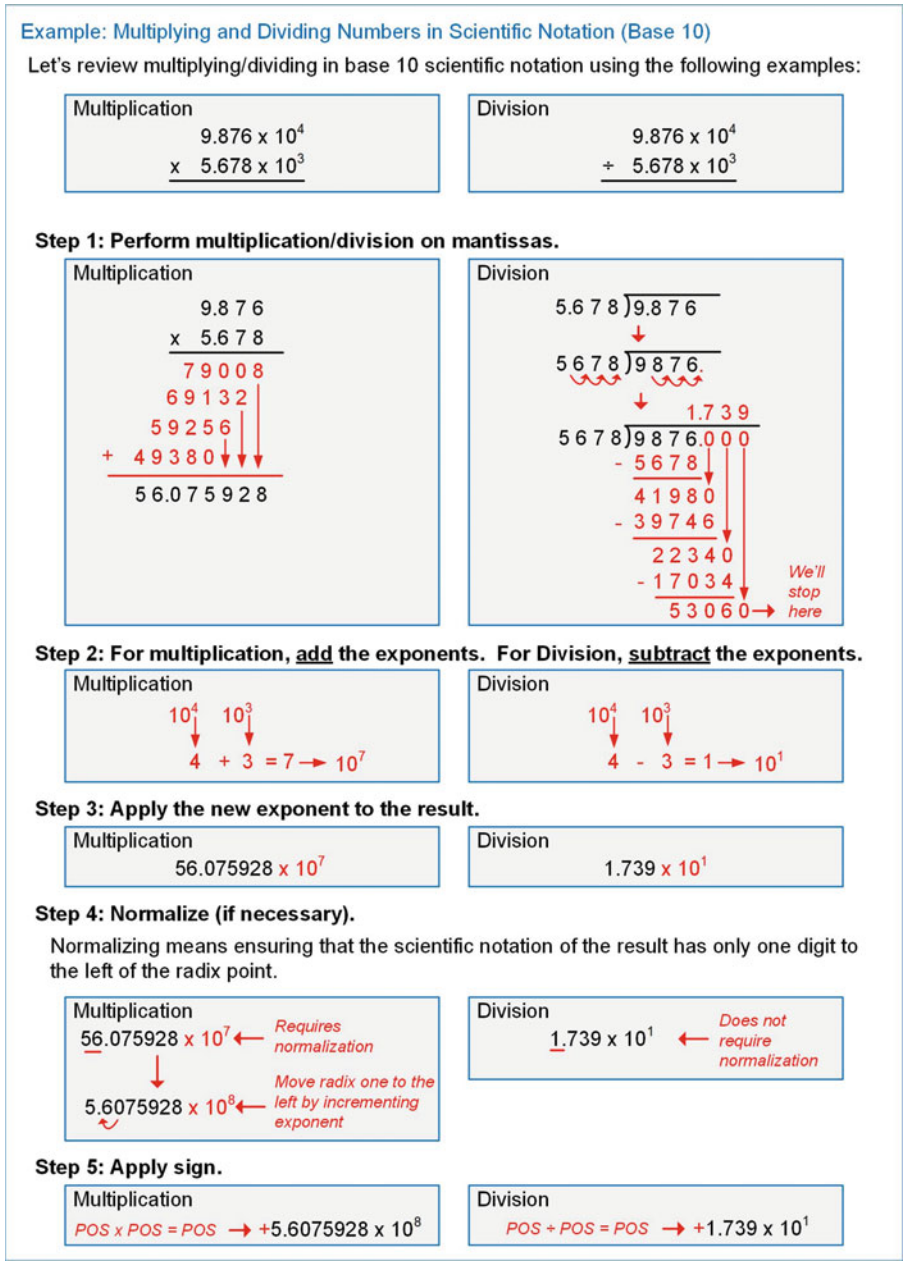
Example 12.11
Subtracting two IEEE 754 single-precision numbers [POS − POS = POS] (part 2)

12.3.2 Multiplication and Division of IEEE 754 Numbers

The process of multiplying and dividing IEEE 754 numbers follows a similar algorithm as when using base 10 SN. For multiplication, the multiplication is performed on the mantissas and then the exponents are added. The resulting exponent is applied to the product of the mantissa multiplication to get the final result. In division, the division is performed on the mantissas and then the exponents are subtracted (dividend exponent – divisor exponent). The resulting exponent is applied to the quotient of the mantissa division for the final result. The final step is to normalize the result if necessary. These steps are summarized below:

1. Perform multiplication/division on the mantissas
2. Add/subtract the exponents
3. Apply the new exponent from step 2 to the result from step 1
4. Normalize the result (if necessary)
5. Apply sign

Example 12.12 shows an example of performing multiplication/division on numbers in base 10 SN to illustrate these steps.

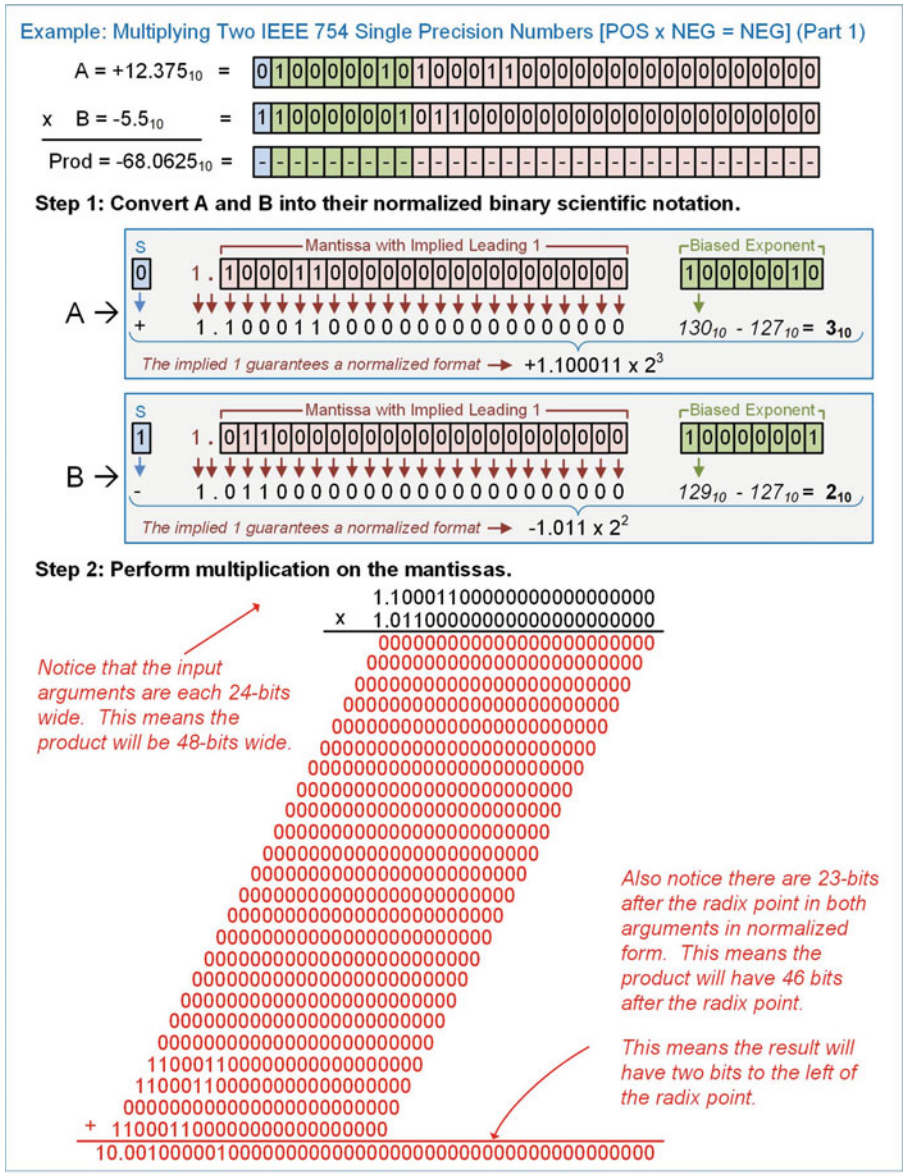


Example 12.12
Multiplying and dividing numbers in scientific notation (base 10)

When performing multiplication/division on IEEE 754 numbers, the sign of the result can be found using an exclusive-OR operation on the sign bits of the inputs. This is independent of the operations on the mantissa and exponent due to the signed magnitude-encoding approach used in IEEE 754. The multiplication/division operations can then be performed directly on the mantissa without any consideration of the sign. The steps to perform multiplication/division on IEEE 754 numbers are below:

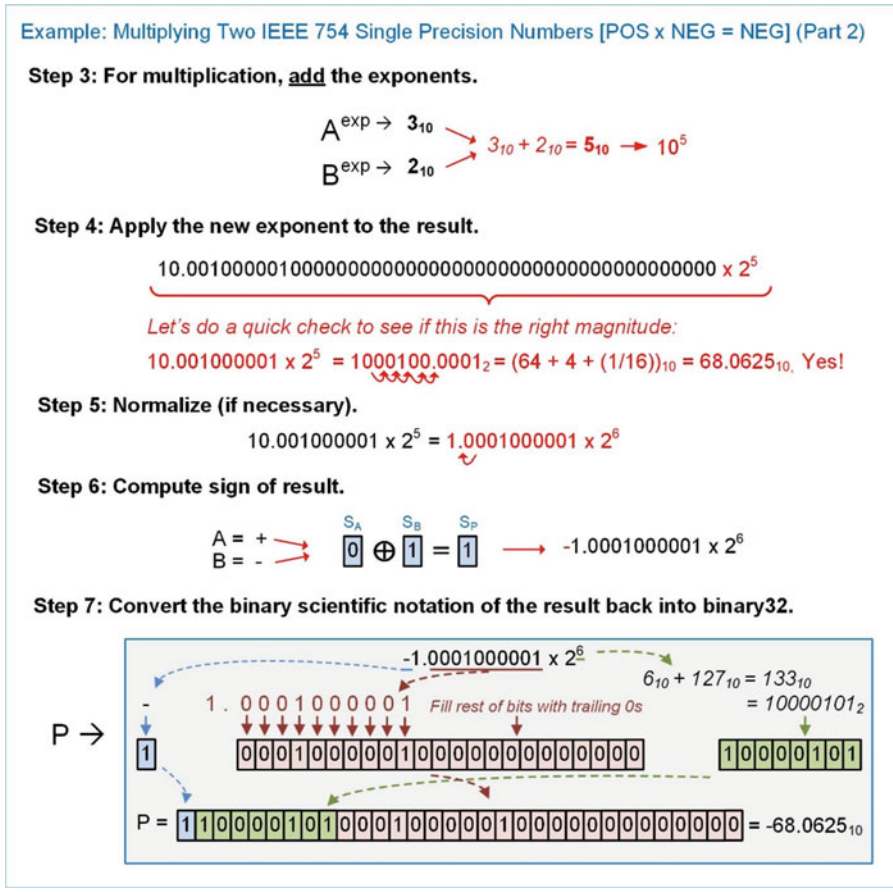
1. Convert input arguments from IEEE 754 into their normalized binary SN (i.e., unpack the inputs)
2. Perform multiplication/division on the mantissas
3. Add/subtract the exponents
4. Apply the new exponent from step 3 to the result from step 2
5. Normalize the result (if necessary)
6. Compute the sign of the result
7. Put all three components of the result back into binary32 format (i.e., pack the results into a 32-bit single-precision word)

Example 12.13 shows steps 1 through 2 of multiplying binary32 numbers.



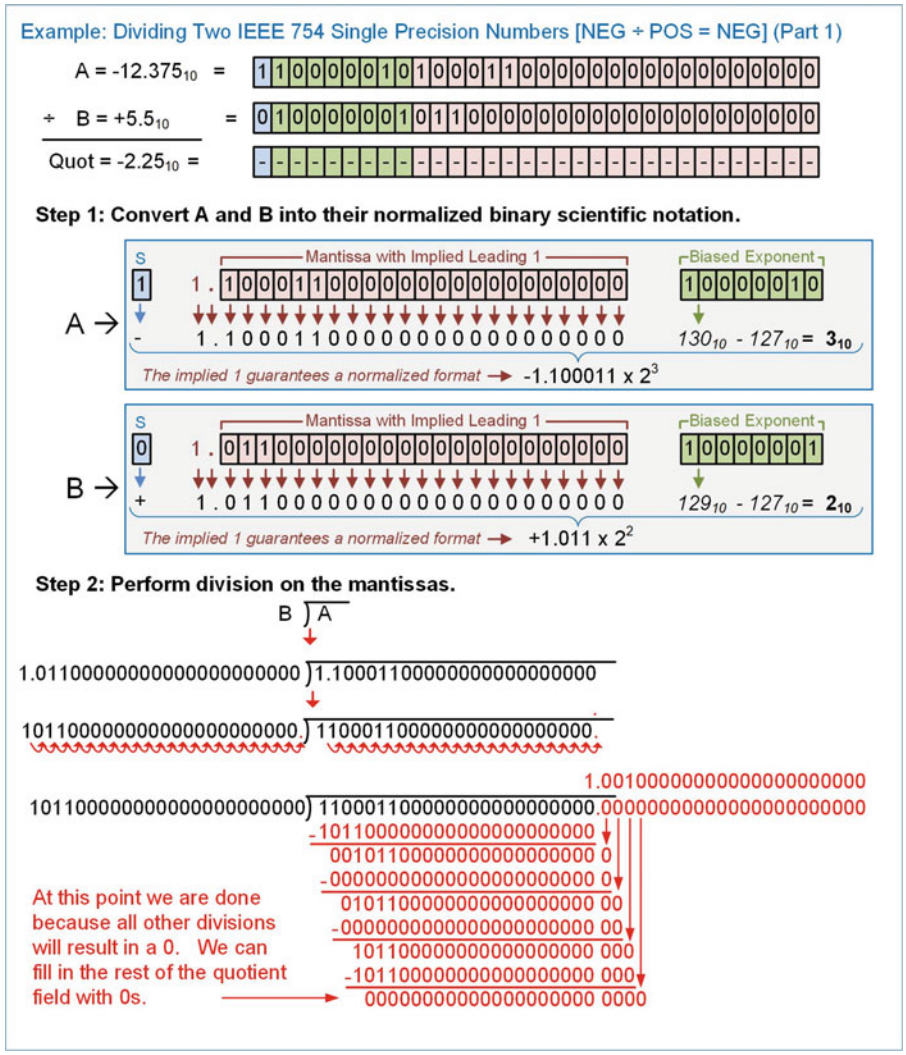
Example 12.13
Multiplying two IEEE 754 single-precision numbers [POS × NEG = NEG] (part 1)

Example 12.14 shows steps 3 through 7 of multiplying binary32 numbers.



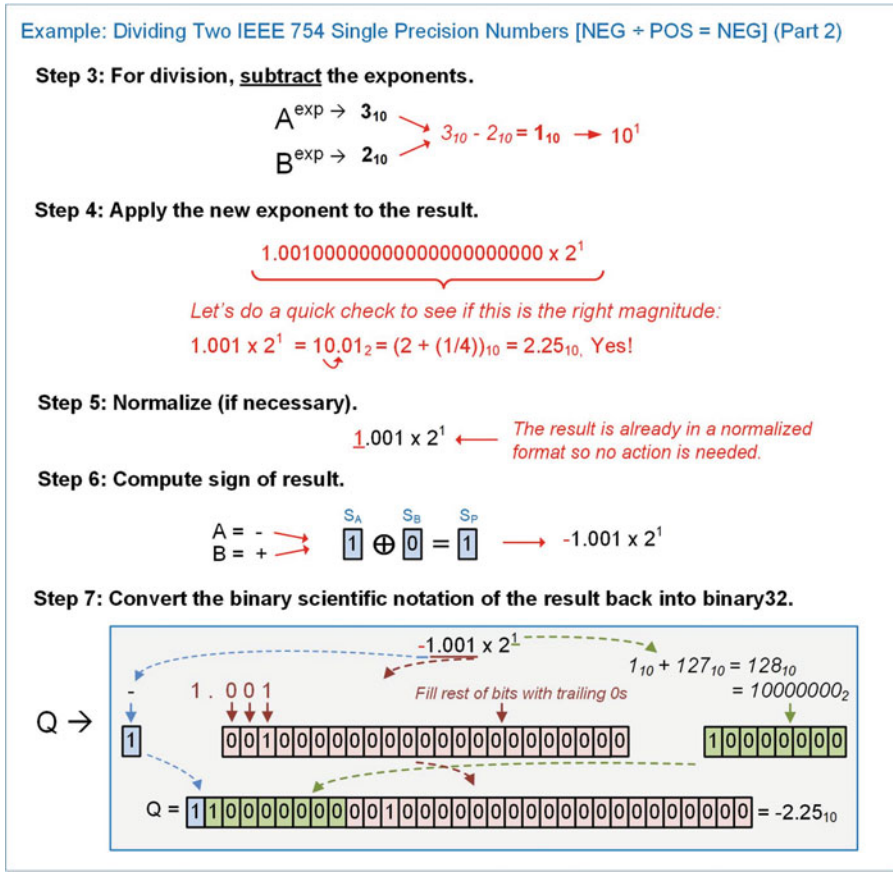
Example 12.14
Multiplying two IEEE 754 single-precision numbers [POS × NEG = NEG] (part 1)

Example 12.15 shows steps 1 through 2 of dividing binary32 numbers.



Example 12.15
Dividing two IEEE 754 single-precision numbers [NEG ÷ POS = NEG] (part 1)

Example 12.16 shows steps 3 through 7 of dividing binary32 numbers.



Example 12.16
Dividing two IEEE 754 single-precision numbers [NEG ÷ POS = NEG] (part 2)

CONCEPT CHECK

CC12.3 The process for performing floating-point arithmetic seems much more complicated than for integer and fixed-point numbers. Is this why not all computers implement floating-point operations in hardware?

A) Yes

12.4 Floating-Point Modeling in Verilog

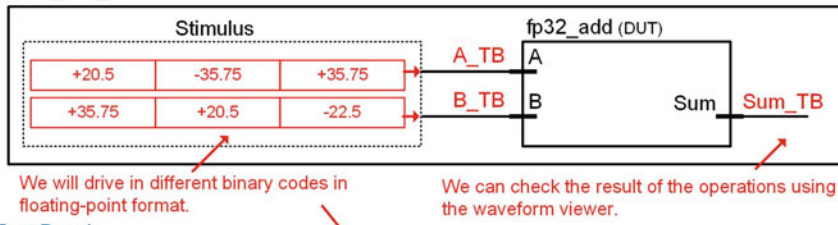
12.4.1 Modeling Floating-Point Addition in Verilog

Verilog does not have inherent library support for the IEEE 754 standard. As a result, implementing floating-point arithmetic in Verilog typically consists of manually coding the logic or using vendor-provided cores. The steps involved in arithmetic modeling mirrors the steps for performing these

operations shown in Sect. 12.3. Let us begin with modeling the addition operation. Example 12.17 shows the test bench setup that will be used to verify the model and step 1 of the operation (i.e., unpacking the inputs). Note that each step consists of defining new signals to hold the interim results and assignment statements with the appropriate logic. This approach models a fully combinational logic implementation of the operation.

Example: Floating-Point Addition in Verilog (Part 1)

fp32_add_TB



We will drive in different binary codes in floating-point format.

We can check the result of the operations using the waveform viewer.

Test Bench

```
`timescale 1ps/1fs

module fp32_add_TB ();

  reg  [31:0] A_TB, B_TB;
  wire [31:0] Sum_TB;

  fp32_add DUT (Sum_TB, A_TB, B_TB);

  initial
    begin
        A_TB = 32'b01000001101001000000000000000000; // binary32 for +20.5
        B_TB = 32'b01000010000011110000000000000000; // binary32 for +35.75

    #50 A_TB = 32'b11000010000011110000000000000000; // binary32 for -35.75
        B_TB = 32'b01000001101001000000000000000000; // binary32 for +20.5

    #50 A_TB = 32'b01000010000011110000000000000000; // binary32 for +35.75
        B_TB = 32'b11000001101101000000000000000000; // binary32 for -22.5

    end

endmodule
```

Addition Model

```
module fp32_add (output wire [31:0] Sum,
                 input  wire [31:0] A, B);

 //---------------------------------------------------------------------
 //-- Step 1: Convert A and B into Normalized Binary (Unpack)
 reg        A_sign, B_sign;          // create signals for sign fields
 reg [7:0]  A_exp,  B_exp;           // create signals for exponent fields
 reg [23:0] A_mant, B_mant;          // create signals for mantissa fields

 assign A_sign = A[31];              // unpack MSB of input as sign bit
 assign A_exp  = A[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign A_mant = {1'b1, A[22:0]};    // unpack 23x mantissa bits + implied 1

 assign B_sign = B[31];              // unpack MSB of input as sign bit
 assign B_exp  = B[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign B_mant = {1'b1, B[22:0]};    // unpack 23x mantissa bits + implied 1
```

Example 12.17
Floating-point addition in Verilog (part 1)

Example 12.18 shows steps 2–4 of the addition operation. Step 2 handles modifying the input arguments to have the same exponent by first checking which exponent is larger, finding the difference in exponents, and then shifting the input mantissa with the lesser exponent to the right the appropriate number of times needed to make its exponent match the other input's exponent. Step 3 converts the input arguments into two's complement format in order to handle negative inputs. Note that this step can be accomplished using other methods but this example uses a two's complement method to match previous arithmetic content provided in this book. Step 4 performs the addition of the modified mantissas. Note that this addition is susceptible to overflow, meaning that it is possible the result will not fit into the range of values capable in the result vector. This example model does not handle overflow checking for simplicity; however, overflow can be handled using a variety of techniques. First, an overflow checking monitor can be implemented that determines if an error has occurred by looking at the signs of the inputs and output. If overflow occurs, a common approach is to modify the input arguments to have a lesser mantissa magnitude by shifting the mantissa right and increasing the exponents. Another approach is to simply pad the magnitudes in step 3 with more leading 0s so that the result vector is large enough to accommodate larger results. The overflow capability is at the discretion of the designer and is a trade-off of functionality and model complexity.

Example: Floating-Point Addition in Verilog (Part 2)

```
//-------------------------------------------------------------------
//-- Step 2: Modify arguments to have same exponents
reg [7:0]  num_shift_rights;       // signal to hold exponent difference
reg [23:0] A_mant_mod, B_mant_mod; // signals to hold shifted mantissas
reg [7:0]  A_exp_mod, B_exp_mod;   // signals to hold modified exponent

always @*                  // adjust input with lesser exponent to match
 begin                     // other input by shifting its mantissa
  if (A_exp >= B_exp)
   begin                                      // A's exponent is largest
    num_shift_rights = A_exp - B_exp;         // determine diff in exponents
    A_mant_mod = A_mant;                      // leave A's mantissa alone
    A_exp_mod  = A_exp;                       // leave A's exponent alone
    B_mant_mod = B_mant >> num_shift_rights; // shift B's mantissa to right
    B_exp_mod  = A_exp;                       // make B's exponent match A's
   end
  else
   begin                                      // B's exponent is largest
    num_shift_rights = B_exp - A_exp;         // determine diff in exponents
    A_mant_mod = A_mant >> num_shift_rights; // shift A's mantissa to right
    A_exp_mod  = B_exp;                       // make A's exponent match B's
    B_mant_mod = B_mant;                      // leave B's mantissa alone
    B_exp_mod  = B_exp;                       // leave B's exponent alone
   end
  end

//-------------------------------------------------------------------
//-- Step 3: Apply 0 sign bit, convert any negative numbers to 2s comp
reg [24:0] A_mant_0pad, B_mant_0pad;      // signals to hold {0, mantissa}
assign A_mant_0pad = {1'b0, A_mant_mod}; // concatenate 0 to A's mantissa
assign B_mant_0pad = {1'b0, B_mant_mod}; // concatenate 0 to B's mantissa

reg [24:0] A_2s_comp, B_2s_comp;          // signals for mantissas 2s comp
                                          // if neg, perform 2s comp neg
assign A_2s_comp = (A_sign == 1'b1) ? (~A_mant_0pad + 1) : (A_mant_0pad);
assign B_2s_comp = (B_sign == 1'b1) ? (~B_mant_0pad + 1) : (B_mant_0pad);

//-------------------------------------------------------------------
//-- Step 4: Perform addition of the mantissas in 2s comp form
reg [24:0] Sum_2s_comp;                   // signal for mantissas sum
assign Sum_2s_comp = A_2s_comp + B_2s_comp; // perform addition
```

⚠Caution – Watch for Overflow

This assignment is susceptible to overflow (i.e., the result won't fit in the available range afforded by the output vector). There are a variety of design options that can be implemented to handle overflow in this step.

Example 12.18
Floating-point addition in Verilog (part 2)

Example 12.19 shows steps 5–7 of the addition operation. Step 5 addresses the sign of the result. If the result is positive, it records the sign and then removes the extra leading 0 added in step 3. If the result is negative, it records the sign and then performs two's complement negation to convert the result into an absolute value before removing the extra leading 0. Step 6 is only used in the hand calculation (i.e., writing the number in binary SN). Step 7 normalizes the result by shifting the mantissa to the left so that there is a leading 1 in the most significant position and adjusting the result's exponent accordingly.

Example: Floating-Point Addition in Verilog (Part 3)

```
//-------------------------------------------------------------------
//-- Step 5: Address sign of the result
reg Sum_sign;                 // signal for final sign of sum
reg [24:0] Sum_mag_2s_comp;   // signal for 25-bit sum of the mantissas
reg [23:0] Sum_mag;           // signal for 24-bit sum after dealing w/ sign

always @*
 begin
  if(Sum_2s_comp[24] == 1'b1)
   begin                                  // if sum is negative,
    Sum_sign = 1;                         // record the sign,
    Sum_mag_2s_comp = ~Sum_2s_comp + 1; // perform 2s comp negation,
    Sum_mag = Sum_mag_2s_comp[23:0];      // strip off leading 0 to yield
   end                                    //  24-bit absolute value of sum
  else
   begin                                  // if sum is positive,
    Sum_sign = 0;                         // record the sign,
    Sum_mag_2s_comp = Sum_2s_comp;        // don't perform 2s comp negation,
    Sum_mag = Sum_mag_2s_comp[23:0];      // strip off leading 0 to yield
   end                                    //  24-bit absolute value of sum
  end

//-------------------------------------------------------------------
//-- Step 6: Apply exponents to the result (hand-calc only)

//-- Step 7: Normalize result (i.e., get one 1 to the left of radix point
reg [4:0]  msb_pos;              // signal for position of the 1st leading 1
reg [23:0] Sum_mant_normalized; // signal for mantissa after shifting
reg [7:0]  Sum_exp_normalized;  // signal for exponent after decrementing

always @*   // this block finds position of the 1st leading 1 in mantissa
 begin
  if      (Sum_mag[23] == 1'b1) msb_pos = 23;
  else if (Sum_mag[22] == 1'b1) msb_pos = 22;
  else if (Sum_mag[21] == 1'b1) msb_pos = 21;
  else if (Sum_mag[20] == 1'b1) msb_pos = 20;
  else if (Sum_mag[19] == 1'b1) msb_pos = 19;
  else if (Sum_mag[18] == 1'b1) msb_pos = 18;
  else if (Sum_mag[17] == 1'b1) msb_pos = 17;
  else if (Sum_mag[16] == 1'b1) msb_pos = 16;
  else if (Sum_mag[15] == 1'b1) msb_pos = 15;
  else if (Sum_mag[14] == 1'b1) msb_pos = 14;
  else if (Sum_mag[13] == 1'b1) msb_pos = 13;
  else if (Sum_mag[12] == 1'b1) msb_pos = 12;
  else if (Sum_mag[11] == 1'b1) msb_pos = 11;
  else if (Sum_mag[10] == 1'b1) msb_pos = 10;
  else if (Sum_mag[9] == 1'b1)  msb_pos = 9;
  else if (Sum_mag[8] == 1'b1)  msb_pos = 8;
  else if (Sum_mag[7] == 1'b1)  msb_pos = 7;
  else if (Sum_mag[6] == 1'b1)  msb_pos = 6;
  else if (Sum_mag[5] == 1'b1)  msb_pos = 5;
  else if (Sum_mag[4] == 1'b1)  msb_pos = 4;
  else if (Sum_mag[3] == 1'b1)  msb_pos = 3;
  else if (Sum_mag[2] == 1'b1)  msb_pos = 2;
  else if (Sum_mag[1] == 1'b1)  msb_pos = 1;
  else if (Sum_mag[0] == 1'b1)  msb_pos = 0;
  else                          msb_pos = 0;
 end

always @* //-- This block shifts mantissa and adjusts exponent accordingly
 begin
  Sum_mant_normalized = Sum_mag << (23 - msb_pos);       // shift mantissa
  Sum_exp_normalized = A_exp_mod - (23 - msb_pos) + 127; // dec exp + bias
 end
```

Example 12.19
Floating-point addition in Verilog (part 3)

Example 12.20 shows the final step of the addition operation (i.e., packing the results) and the functional simulation waveform verifying the correct operation of the model.

Example: Floating-Point Addition in Verilog (Part 4)

```
//-------------------------------------------------------------------------
//-- Step 8: Pack the three final fields back into binary32 format,
//--         only storing lower 23-bits of mantissa to leave off leading 1
assign Sum = {Sum_sign, Sum_exp_normalized, Sum_mant_normalized[22:0]};

endmodule
```

Simulation Waveforms Verifying Proper Results

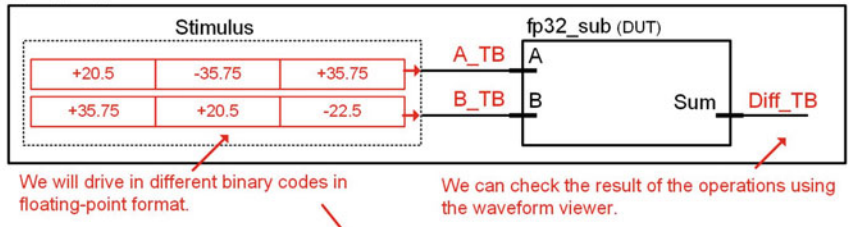
Example 12.20
Floating-point addition in Verilog (part 4)

12.4.2 Modeling Floating-Point Subtraction in Verilog

Modeling subtraction is performed using a very similar process to addition with the exception that an extra step is inserted to perform two's complement negation on the subtrahend. This allows the inputs to be added together to accomplish the subtraction in order to reuse addition circuitry that may already exist in the system. Example 12.21 shows the test bench setup and step 1 of the subtraction operation (i.e., unpacking the results).

Example: Floating-Point Subtraction in Verilog (Part 1)

fp32_sub_TB



Test Bench

```
`timescale 1ps/1fs

module fp32_sub_TB ();

  reg  [31:0] A_TB, B_TB;
  wire [31:0] Diff_TB;

  fp32_sub DUT (Diff_TB, A_TB, B_TB);

  initial
    begin
        A_TB = 32'b01000001101001000000000000000000; // binary32 for +20.5
        B_TB = 32'b01000010000011110000000000000000; // binary32 for +35.75

    #50 A_TB = 32'b11000010000011110000000000000000; // binary32 for -35.75
        B_TB = 32'b01000001101001000000000000000000; // binary32 for +20.5

    #50 A_TB = 32'b01000010000011110000000000000000; // binary32 for +35.75
        B_TB = 32'b11000001101101000000000000000000; // binary32 for -22.5

    end

endmodule
```

Subtraction Model

```
module fp32_sub (output wire [31:0] Diff,
                 input  wire [31:0] A, B);

 //------------------------------------------------------------------
 //-- Step 1: Convert A and B into Normalized Binary (Unpack)
 reg        A_sign, B_sign;          // create signals for sign fields
 reg [7:0]  A_exp,  B_exp;           // create signals for exponent fields
 reg [23:0] A_mant, B_mant;          // create signals for mantissa fields

 assign A_sign = A[31];              // unpack MSB of input as sign bit
 assign A_exp  = A[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign A_mant = {1'b1, A[22:0]};    // unpack 23x mantissa bits + implied 1

 assign B_sign = B[31];              // unpack MSB of input as sign bit
 assign B_exp  = B[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign B_mant = {1'b1, B[22:0]};    // unpack 23x mantissa bits + implied 1
```

Example 12.21
Floating-point subtraction in Verilog (part 1)

Example 12.22 shows steps 2–5 of the subtraction operation (i.e., modifying the arguments to have the same exponents, converting the inputs into two's complement representation to handle negative values, taking the two's complement negation of the subtrahend, and performing the addition. Just as was the case in addition, overflow is a concern in step 5.

Example: Floating-Point Subtraction in Verilog (Part 2)

```
//-------------------------------------------------------------------
//-- Step 2: Modify arguments to have same exponents
reg [7:0]  num_shift_rights;       // signal to hold exponent difference
reg [23:0] A_mant_mod, B_mant_mod; // signals to hold shifted mantissas
reg [7:0]  A_exp_mod, B_exp_mod;   // signals to hold modified exponents

always @*                  // adjust input with lesser exponent to match
 begin                     // other input by shifting its mantissa
  if (A_exp >= B_exp)
   begin                                      // A's exponent is largest
    num_shift_rights = A_exp - B_exp;         // determine diff in exponents
    A_mant_mod = A_mant;                      // leave A's mantissa alone
    A_exp_mod  = A_exp;                       // leave A's exponent alone
    B_mant_mod = B_mant >> num_shift_rights; // shift B's mantissa to right
    B_exp_mod  = A_exp;                       // make B's exponent match A's
   end
  else
   begin                                      // B's exponent is largest
    num_shift_rights = B_exp - A_exp;         // determine diff in exponents
    A_mant_mod = A_mant >> num_shift_rights; // shift A's mantissa to right
    A_exp_mod  = B_exp;                       // make A's exponent match B's
    B_mant_mod = B_mant;                      // leave B's mantissa alone
    B_exp_mod  = B_exp;                       // leave B's exponent alone
   end
  end

//-------------------------------------------------------------------
//-- Step 3: Apply 0 sign bit, convert any negative numbers to 2s comp
reg [24:0] A_mant_0pad, B_mant_0pad;      // signals to hold {0, mantissa}
assign A_mant_0pad = {1'b0, A_mant_mod};  // concatenate 0 to A's mantissa
assign B_mant_0pad = {1'b0, B_mant_mod};  // concatenate 0 to B's mantissa

reg [24:0] A_2s_comp, B_2s_comp;          // signals for mantissas 2s comp
                                          // if neg, perform 2s comp neg
assign A_2s_comp = (A_sign == 1'b1) ? (~A_mant_0pad + 1) : (A_mant_0pad);
assign B_2s_comp = (B_sign == 1'b1) ? (~B_mant_0pad + 1) : (B_mant_0pad);

//-------------------------------------------------------------------
//-- Step 4: Take 2s comp of subtrahend so we can use addition
reg [24:0] B_2s_comp_subtrahend;              // signal for subtrahend
assign B_2s_comp_subtrahend = ~B_2s_comp + 1; // perform 2s comp neg

//-------------------------------------------------------------------
//-- Step 5: Perform subtraction of mantissas using addition operation
//--         Note:  This model does not account for overflow
reg [24:0] Diff_2s_comp;
assign Diff_2s_comp = A_2s_comp + B_2s_comp_subtrahend; // perform sub
```

⚠ Caution – Watch for Overflow

This assignment is susceptible to overflow (i.e., the result won't fit in the available range afforded by the output vector). There are a variety of design options that can be implemented to handle overflow in this step.

Example 12.22
Floating-point subtraction in Verilog (part 2)

Example 12.23 shows steps 6–8 of the subtraction operation (i.e., addressing the sign bit of the difference andnormalizing the result).

Example: Floating-Point Subtraction in Verilog (Part 3)

```
//-----------------------------------------------------------------
//-- Step 6: Address sign of the result
reg Diff_sign;                // signal for final sign of diff
reg [24:0] Diff_mag_2s_comp; // signal for 25-bit diff of the mantissas
reg [23:0] Diff_mag;          // signal for 24-bit diff after dealing w sign

always @*
 begin
  if(Diff_2s_comp[24] == 1'b1)
   begin                                  // if diff is negative,
    Diff_sign = 1;                        // record the sign,
    Diff_mag_2s_comp = ~Diff_2s_comp + 1; // perform 2s comp negation,
    Diff_mag = Diff_mag_2s_comp[23:0];    // strip off leading 0 to yield
   end                                    //  24-bit absolute value of diff
  else
   begin                                  // if diff is positive,
    Diff_sign = 0;                        // record the sign,
    Diff_mag_2s_comp = Diff_2s_comp;      // don't perform 2s comp neg
    Diff_mag = Diff_mag_2s_comp[23:0];    // strip off leading 0 to yield
   end                                    //  24-bit absolute value of diff
  end

//-----------------------------------------------------------------
//-- Step 7: Apply exponents to the result (hand-calc only)

//-- Step 8: Normalize result (i.e., get one 1 to the left of radix point
reg [4:0]  msb_pos;              // signal for position of 1st leading 1
reg [23:0] Diff_mant_normalized; // signal for mantissa after shifting
reg [7:0]  Diff_exp_normalized;  // signal for exponent after decrementing

always @*   // this block finds position of the 1st leading 1 in mantissa
 begin
  if      (Diff_mag[23] == 1'b1) msb_pos = 23;
  else if (Diff_mag[22] == 1'b1) msb_pos = 22;
  else if (Diff_mag[21] == 1'b1) msb_pos = 21;
  else if (Diff_mag[20] == 1'b1) msb_pos = 20;
  else if (Diff_mag[19] == 1'b1) msb_pos = 19;
  else if (Diff_mag[18] == 1'b1) msb_pos = 18;
  else if (Diff_mag[17] == 1'b1) msb_pos = 17;
  else if (Diff_mag[16] == 1'b1) msb_pos = 16;
  else if (Diff_mag[15] == 1'b1) msb_pos = 15;
  else if (Diff_mag[14] == 1'b1) msb_pos = 14;
  else if (Diff_mag[13] == 1'b1) msb_pos = 13;
  else if (Diff_mag[12] == 1'b1) msb_pos = 12;
  else if (Diff_mag[11] == 1'b1) msb_pos = 11;
  else if (Diff_mag[10] == 1'b1) msb_pos = 10;
  else if (Diff_mag[9] == 1'b1)  msb_pos = 9;
  else if (Diff_mag[8] == 1'b1)  msb_pos = 8;
  else if (Diff_mag[7] == 1'b1)  msb_pos = 7;
  else if (Diff_mag[6] == 1'b1)  msb_pos = 6;
  else if (Diff_mag[5] == 1'b1)  msb_pos = 5;
  else if (Diff_mag[4] == 1'b1)  msb_pos = 4;
  else if (Diff_mag[3] == 1'b1)  msb_pos = 3;
  else if (Diff_mag[2] == 1'b1)  msb_pos = 2;
  else if (Diff_mag[1] == 1'b1)  msb_pos = 1;
  else if (Diff_mag[0] == 1'b1)  msb_pos = 0;
  else                           msb_pos = 0;
 end

always @* //-- This block shifts mantissa and adjusts exponent accordingly
 begin
  Diff_mant_normalized = Diff_mag << (23 - msb_pos);      // shift mantissa
  Diff_exp_normalized = A_exp_mod - (23 - msb_pos) + 127; // dec exp + bias
 end
```

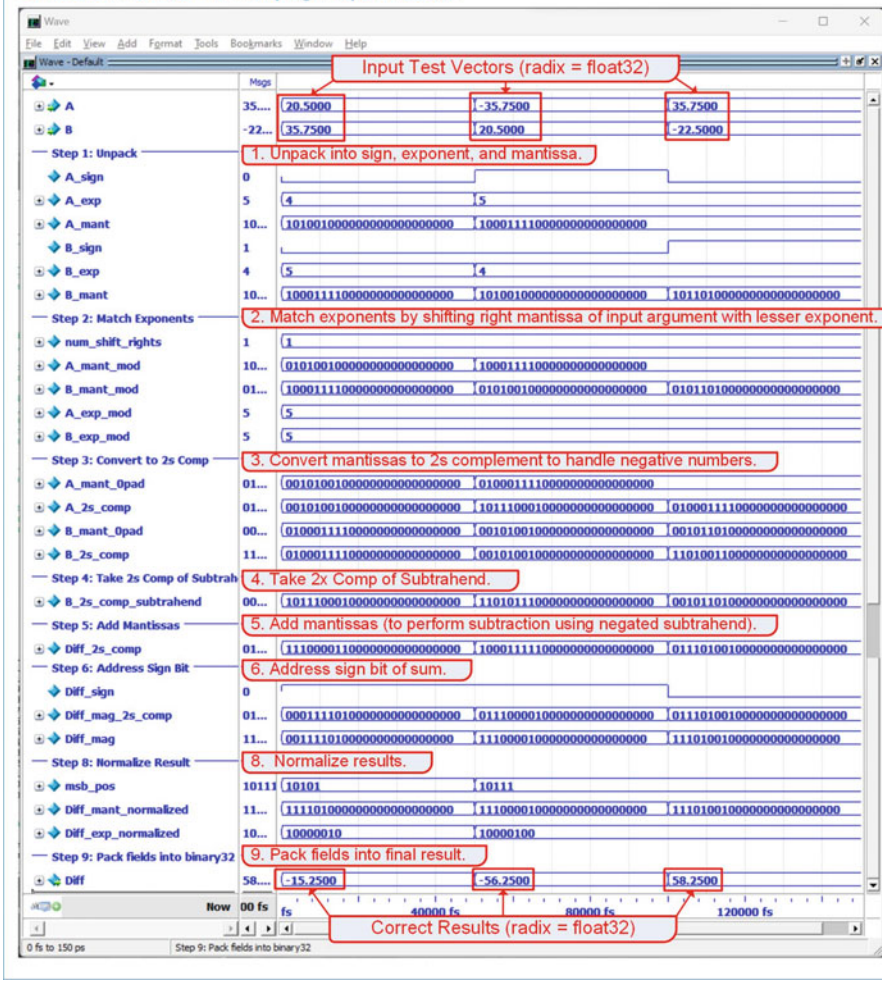
Example 12.23
Floating-point subtraction in Verilog (part 3)

Example 12.24 shows the final step of the subtraction operation (i.e., packing the results) and the functional simulation waveform verifying the correct operation of the model.

Example: Floating-Point Subtraction in Verilog (Part 4)

```
//-------------------------------------------------------------------
//-- Step 9: Pack the three final fields back into binary32 format,
//--         only storing lower 23-bits of mantissa to leave off leading 1
assign Diff = {Diff_sign, Diff_exp_normalized, Diff_mant_normalized[22:0]};

endmodule
```

Simulation Waveforms Verifying Proper Results



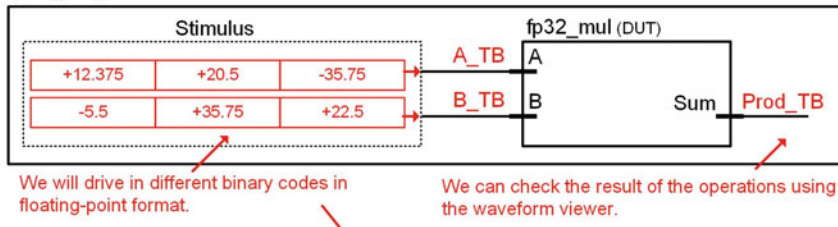
Example 12.24
Floating-point subtraction in Verilog (part 4)

12.4.3 Modeling Floating-Point Multiplication in Verilog

Multiplication modeling follows the process of multiplying binary SN by hand very closely. Example 12.25 shows the test bench setup and step 1 of the operation (i.e., unpacking the inputs).

Example: Floating-Point Multiplication in Verilog (Part 1)

fp32_mul_TB



We will drive in different binary codes in floating-point format.

We can check the result of the operations using the waveform viewer.

Test Bench

```
`timescale 1ps/1fs

module fp32_mul_TB ();

  reg  [31:0] A_TB, B_TB;
  wire [31:0] Prod_TB;

  fp32_mul DUT (Prod_TB, A_TB, B_TB);

  initial
    begin
      A_TB   = 32'b01000001010001100000000000000000; // binary32 for +12.375
      B_TB   = 32'b11000000101100000000000000000000; // binary32 for -5.5

  #50 A_TB   = 32'b01000001101001000000000000000000; // binary32 for +20.5
      B_TB   = 32'b01000010000011110000000000000000; // binary32 for +35.75

  #50 A_TB   = 32'b11000010000011110000000000000000; // binary32 for -35.75
      B_TB   = 32'b01000001101101000000000000000000; // binary32 for +22.5

    end

endmodule
```

Multiplication Model

```
module fp32_mul (output wire [31:0] Prod,
                 input  wire [31:0] A, B);

 //------------------------------------------------------------------
 //-- Step 1: Convert A and B into Normalized Binary (Unpack)
 reg        A_sign, B_sign;          // create signals for sign fields
 reg [7:0]  A_exp,  B_exp;           // create signals for exponent fields
 reg [23:0] A_mant, B_mant;          // create signals for mantissa fields

 assign A_sign = A[31];              // unpack MSB of input as sign bit
 assign A_exp  = A[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign A_mant = {1'b1, A[22:0]};    // unpack 23x mantissa bits + implied 1

 assign B_sign = B[31];              // unpack MSB of input as sign bit
 assign B_exp  = B[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign B_mant = {1'b1, B[22:0]};    // unpack 23x mantissa bits + implied 1
```

Example 12.25
Floating-point multiplication in Verilog (part 1)

Example 12.26 shows steps 2–5 of the multiplication operation (i.e., multiplying the mantissas, adding the exponents, and normalizing the result). Note that when performing the multiplication of the mantissas, the product will require 48-bits for the result. When this multiplication is performed, the product vector will have two bits to the left of the radix point. This new location must be accounted for by adjusting the exponent in the normalization step. Note that the bottom half of the 48-bit interim vector holding the product will be discarded to fit the result into the mantissa field of a single-precision number. This is considered a truncation error.

Example: Floating-Point Multiplication in Verilog (Part 2)

```
//-----------------------------------------------------------------
//-- Step 2: Perform multiplication of mantissas
reg [47:0] Prod_mant;                   // signal to hold mant product
assign Prod_mant = A_mant * B_mant;     // perform multiplication

//-----------------------------------------------------------------
//-- Step 3: Add the exponents
reg [7:0] Prod_exp;                     // signal to hold result exponent
assign Prod_exp = A_exp + B_exp ;       // perform addition

//-----------------------------------------------------------------
//-- Step 4: Apply exponent to the mantissa (hand-calc only)

//-- Step 5: Normalize result (i.e., get one 1 to the left of radix point
//--         The product has TWO bits to left of radix point at this point
reg [5:0]  msb_pos;                 // signal for position of 1st leading 1
reg [47:0] Prod_mant_normalized; // signal for mantissa after shifting
reg [7:0]  Prod_exp_normalized;  // signal for exponent after decrementing

always @*   // this block finds position of the 1st leading 1 in mantissa
 begin
  if      (Prod_mant[47] == 1'b1) msb_pos = 47;
  else if (Prod_mant[46] == 1'b1) msb_pos = 46;
  else if (Prod_mant[45] == 1'b1) msb_pos = 45;
  else if (Prod_mant[44] == 1'b1) msb_pos = 44;
  else if (Prod_mant[43] == 1'b1) msb_pos = 43;
  else if (Prod_mant[42] == 1'b1) msb_pos = 42;
  else if (Prod_mant[41] == 1'b1) msb_pos = 41;
  else if (Prod_mant[40] == 1'b1) msb_pos = 40;
  else if (Prod_mant[39] == 1'b1) msb_pos = 39;
  else if (Prod_mant[38] == 1'b1) msb_pos = 38;
  else if (Prod_mant[37] == 1'b1) msb_pos = 37;
  else if (Prod_mant[36] == 1'b1) msb_pos = 36;
  else if (Prod_mant[35] == 1'b1) msb_pos = 35;
  else if (Prod_mant[34] == 1'b1) msb_pos = 34;
  else if (Prod_mant[33] == 1'b1) msb_pos = 33;
  else if (Prod_mant[32] == 1'b1) msb_pos = 32;
  else if (Prod_mant[31] == 1'b1) msb_pos = 31;
  else if (Prod_mant[30] == 1'b1) msb_pos = 30;
  else if (Prod_mant[29] == 1'b1) msb_pos = 29;
  else if (Prod_mant[28] == 1'b1) msb_pos = 28;
  else if (Prod_mant[27] == 1'b1) msb_pos = 27;
  else if (Prod_mant[26] == 1'b1) msb_pos = 26;
  else if (Prod_mant[25] == 1'b1) msb_pos = 25;
  else if (Prod_mant[24] == 1'b1) msb_pos = 24;
  else                            msb_pos = 24;
 end

always @* //-- This block shifts mantissa and adjusts exponent accordingly
 begin    //--  by adding back bias & accounting for radix location in prod
  Prod_mant_normalized = Prod_mant << (47 - msb_pos);        // shift mant
  Prod_exp_normalized = Prod_exp - (47 - msb_pos) + 1 + 127; // dec exp
 end
```

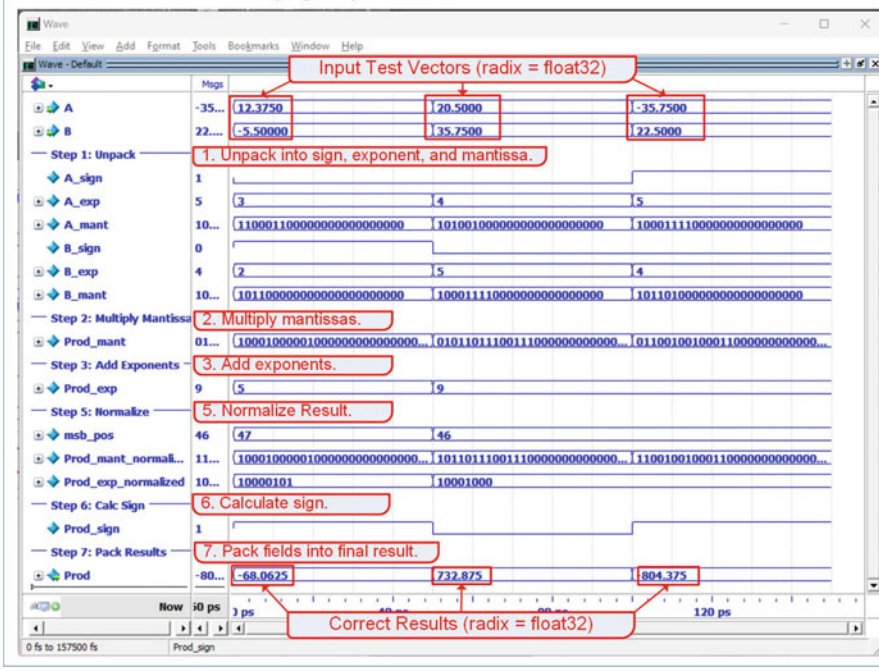
Example 12.26
Floating-point multiplication in Verilog (part 2)

Example 12.27 shows steps 6 and 7 of the multiplication operation (i.e., computing the sign and packing the results) along the functional simulation waveform verifying the correct operation of the model. The sign is found using an XOR operation. During the packing stage, the implied 1 and the lower bits of the 48-bit product are stripped off.

Example: Floating-Point Multiplication in Verilog (Part 3)

```
//-----------------------------------------------------------------
//-- Step 6: Calculate sign of the result
reg Prod_sign;                          // signal for final sign of product
assign Prod_sign = A_sign ^ B_sign; // perform XOR of input signs

//-----------------------------------------------------------------
//-- Step 7: Pack the three final fields back into binary32 format,
//--         only storing lower 23-bits of mantissa to leave off leading 1
assign Prod = {Prod_sign, Prod_exp_normalized, Prod_mant_normalized[46:24]};

endmodule
```

Simulation Waveforms Verifying Proper Results



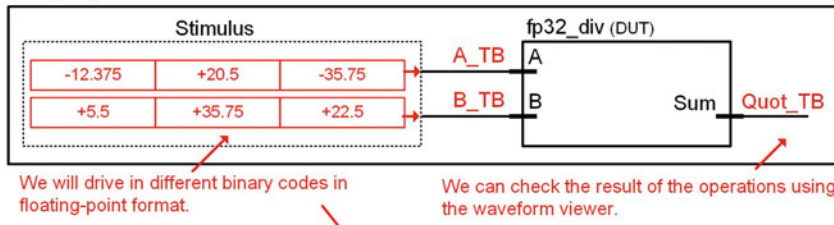
Example 12.27
Floating-point multiplication in Verilog (part 3)

12.4.4 Modeling Floating-Point Division in Verilog

Division modeling also follows the process of dividing binary SN by hand very closely. Example 12.28 shows the test bench setup and step 1 of the operation (i.e., unpacking the inputs).

Example: Floating-Point Division in Verilog (Part 1)

fp32_div_TB



Test Bench

```
`timescale 1ps/1fs

module fp32_div_TB ();

  reg  [31:0] A_TB, B_TB;
  wire [31:0] Quot_TB;

  fp32_div DUT (Quot_TB, A_TB, B_TB);

  initial
    begin
      A_TB   = 32'b11000001010001100000000000000000; // binary32 for -12.375
      B_TB   = 32'b01000000101100000000000000000000; // binary32 for +5.5

  #50 A_TB   = 32'b01000001101001000000000000000000; // binary32 for +20.5
      B_TB   = 32'b01000010000011110000000000000000; // binary32 for +35.75

  #50 A_TB   = 32'b11000010000011110000000000000000; // binary32 for -35.75
      B_TB   = 32'b01000001101101000000000000000000; // binary32 for +22.5

    end

endmodule
```

Division Model

```
module fp32_div (output wire [31:0] Quot,
                 input  wire [31:0] A, B);

 //-----------------------------------------------------------------
 //-- Step 1: Convert A and B into Normalized Binary (Unpack)
 reg        A_sign, B_sign;          // create signals for sign fields
 reg [7:0]  A_exp,  B_exp;           // create signals for exponent fields
 reg [23:0] A_mant, B_mant;          // create signals for mantissa fields

 assign A_sign = A[31];              // unpack MSB of input as sign bit
 assign A_exp  = A[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign A_mant = {1'b1, A[22:0]};    // unpack 23x mantissa bits + implied 1

 assign B_sign = B[31];              // unpack MSB of input as sign bit
 assign B_exp  = B[30:23] - 127;     // unpack 8x exponent bits, remove bias
 assign B_mant = {1'b1, B[22:0]};    // unpack 23x mantissa bits + implied 1
```

Example 12.28
Floating-point division in Verilog (part 1)

Example 12.29 shows steps 2–4 of the division operation (i.e., dividing the mantissas and subtracting the exponents). The mantissa division requires a manual implementation because the built in divide (/) and modulo (%) operations in Verilog do not produce the desired real number format that is needed in binary32. The division is accomplished using an iterative approach that closely mirrors long division by hand. The quotient is determined bit-by-bit using a for() loop starting with the MSB and moving to the LSB. To make room for numbers that may exceed the original 24-bits of the mantissas during the operation, the original mantissas are converted into a 48-bit *dividend* and a 48-bit *divisor* by concatenating 24-bits of trailing 0s. The division then performs a compare to determine if the dividend is divisible by the divisor. This compare simply asks if the dividend is greater than or equal to the divisor.

The result of this compare is either a 1 (*yes, the dividend is divisible by the divisor*) or a 0 (*no, the dividend is not divisible by the divisor*). The 1 or 0 result of this compare is stored into the current bit of the quotient being evaluated and then used to calculate the *multiple* used to determine the *remainder*. Since the result of the compare is either a 1 or a 0, the multiple will either be the current divisor value (i.e., 1 * divisor = divisor) or 0 (i.e., 0 * divisor = 0). The interim remainder is then calculated by subtracting the multiple from the dividend. This remainder will be used as the dividend the next time through the for() loop so before the loop runs again, the remainder is assigned to the dividend signal. Also prior to the loop running again, the divisor must be adjusted so that the compare is made to one lower bit position within the dividend. This can be accomplished by leaving the dividend alone and shifting the divisor one bit to the right. This process is repeated until all 24-bits of the quotient have been found. The resulting quotient will have one bit to the left of the radix point. The exponent subtraction is performed using the Verilog "-" operator.

Example: Floating-Point Division in Verilog (Part 2)

```
//-------------------------------------------------------------------
//-- Step 2: Perform division of mantissas
reg [23:0] Quot_mant;  // signal to hold mant quotient result
reg [47:0] dividend;   // signal for interim dividend
reg [47:0] divisor;    // signal for interim divisor
reg [47:0] remainder;  // signal for interim remainder
reg [47:0] multiple;   // signal for interim multiple
integer    i;          // signal for tracking current location of Quot_mant

always @* begin

 dividend = {A_mant, 24'b0}; // convert dividend & divisor to 48-bits so
 divisor  = {B_mant, 24'b0}; //  that there is room for operations

 for (i=23; i>=0; i=i-1)     // we calc each quotient bit starting with MSB
   begin

    if (dividend >= divisor) // this is the division operation
     begin
      Quot_mant[i] = 1'b1;   // divisor goes into dividend, quot=1
      multiple = divisor;    // 1 * divisor = divisor
     end
    else
     begin
      Quot_mant[i] = 1'b0;   // divisor does NOT go into dividend, quot=0
      multiple = 0;          // 0 * divisor = 0
     end

     remainder = dividend - multiple; // perform subtraction to find rem
     dividend = remainder;            // rem now serves as new dividend
     divisor = divisor >> 1;          // for the next div, we compare
                                      //  divisor to next lower 24-bits of
                                      //  dividend so we shift divisor
   end                                //  to RIGHT by one bit.
 end

//-------------------------------------------------------------------
//-- Step 3: Subtract the exponents
reg [7:0] Quot_exp;                // signal to hold result exponent
assign Quot_exp = A_exp - B_exp ;  // perform subtraction

//-------------------------------------------------------------------
//-- Step 4: Apply exponent to the mantissa (hand-calc only)
```

Example 12.29
Floating-point division in Verilog (part 2)

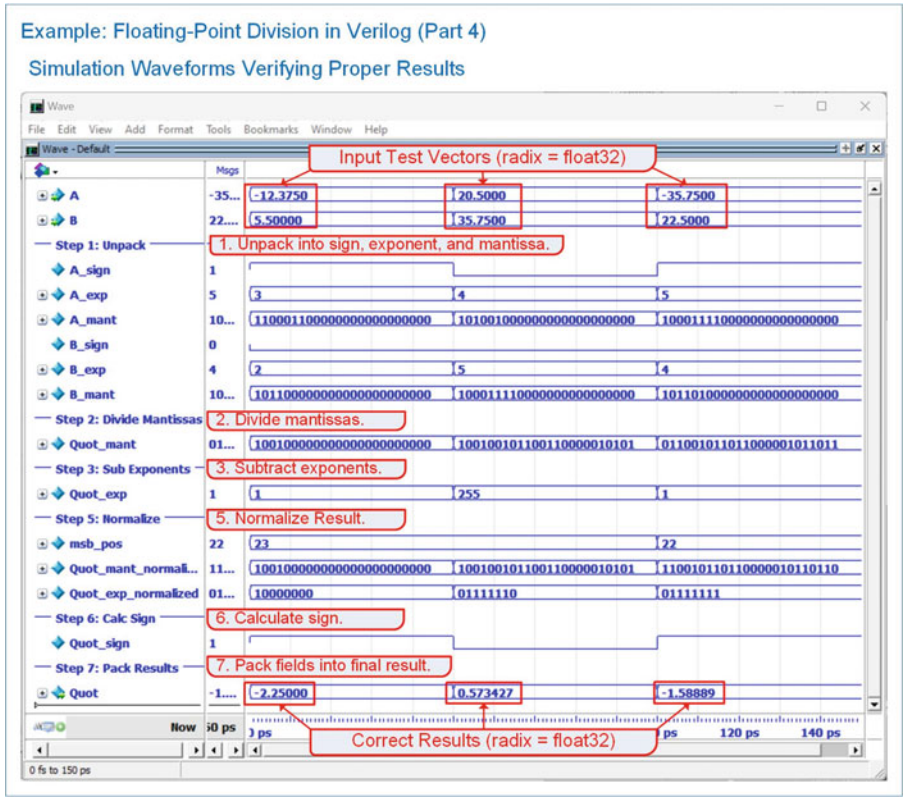
Example 12.30 shows steps 5 and 7 of the division operation (i.e., normalization, computing the sign, and packing the results). The normalization step finds the position of the leading 1 in the mantissa and then shifts the mantissa to the left and decrementing the exponent accordingly. The sign bit is found using an XOR operation. The final packing step only stores the lower 23-bits of the mantissa to strip off the implied leading 1.

Example: Floating-Point Division in Verilog (Part 3)

```
//-----------------------------------------------------------------
//-- Step 5: Normalize result (i.e., get one 1 to the left of radix point
reg [5:0]  msb_pos;              // signal for position of 1st leading 1
reg [23:0] Quot_mant_normalized; // signal for mantissa after shifting
reg [7:0]  Quot_exp_normalized;  // signal for exponent after decrementing

always @*   // this block finds position of the 1st leading 1 in mantissa
 begin
  if      (Quot_mant[23] == 1'b1) msb_pos = 23;
  else if (Quot_mant[22] == 1'b1) msb_pos = 22;
  else if (Quot_mant[21] == 1'b1) msb_pos = 21;
  else if (Quot_mant[20] == 1'b1) msb_pos = 20;
  else if (Quot_mant[19] == 1'b1) msb_pos = 19;
  else if (Quot_mant[18] == 1'b1) msb_pos = 18;
  else if (Quot_mant[17] == 1'b1) msb_pos = 17;
  else if (Quot_mant[16] == 1'b1) msb_pos = 16;
  else if (Quot_mant[15] == 1'b1) msb_pos = 15;
  else if (Quot_mant[14] == 1'b1) msb_pos = 14;
  else if (Quot_mant[13] == 1'b1) msb_pos = 13;
  else if (Quot_mant[12] == 1'b1) msb_pos = 12;
  else if (Quot_mant[11] == 1'b1) msb_pos = 11;
  else if (Quot_mant[10] == 1'b1) msb_pos = 10;
  else if (Quot_mant[9] == 1'b1)  msb_pos = 9;
  else if (Quot_mant[8] == 1'b1)  msb_pos = 8;
  else if (Quot_mant[7] == 1'b1)  msb_pos = 7;
  else if (Quot_mant[6] == 1'b1)  msb_pos = 6;
  else if (Quot_mant[5] == 1'b1)  msb_pos = 5;
  else if (Quot_mant[4] == 1'b1)  msb_pos = 4;
  else if (Quot_mant[3] == 1'b1)  msb_pos = 3;
  else if (Quot_mant[2] == 1'b1)  msb_pos = 2;
  else if (Quot_mant[1] == 1'b1)  msb_pos = 1;
  else if (Quot_mant[0] == 1'b1)  msb_pos = 0;
  else                            msb_pos = 0;
 end

always @* //-- This block shifts mantissa and adjusts exponent accordingly
 begin
  Quot_mant_normalized = Quot_mant << (23 - msb_pos);    // shift mantissa
  Quot_exp_normalized = Quot_exp - (23 - msb_pos) + 127; // dec exp + bias
 end

//-----------------------------------------------------------------
//-- Step 6: Calculate sign of the result
reg Quot_sign;                    // signal for final sign of product
assign Quot_sign = A_sign ^ B_sign; // perform XOR of input signs

//-----------------------------------------------------------------
//-- Step 7: Pack the three final fields back into binary32 format,
//--         only storing lower 23-bits of mantissa to leave off leading 1
assign Quot = {Quot_sign, Quot_exp_normalized, Quot_mant_normalized[22:0]};

endmodule
```

Example 12.30
Floating-point division in Verilog (part 3)

Example 12.31 shows the simulation waveform for the division model verifying the correct outputs.



Example 12.31
Floating-point division in Verilog (part 4)

Concept Check

CC12.4 Why would a designer ever want to create a custom floating-point model instead of using a fully IEEE 754 compliant, prebuilt core provided by a device vendor?

A) A fully compliant IEEE 754 core maybe impractically large for most embedded designs.

B) Designers often want to customize certain floating-point functions to achieve greater speed or reduced power consumption.

C) Designers often only want to implement a subset of IEEE 754 in hardware to reduce the size of the final implementation.

D) All of the above.

Summary

- Floating-point number encoding provides a way to represent very large and very small numbers by expressing the value in binary scientific notation with the ability to move the radix point.
- IEEE 754 is a standard that defines the encoding for specific floating-point types, rounding techniques, and functions that should be supported.

- An IEEE 754 floating-point number consists of a sign bit, an exponent field, and a mantissa field.
- In IEEE 754, a sign bit of 0 indicates a positive number and a sign bit of 1 indicates a negative number. This is a signed magnitude approach to representing the polarity of the number.
- IEEE 754 uses a biased exponent, which shifts the exponent's center point from 0 to a predetermined bias. This allows the exponent to always be positive instead of having both positive and negative values.
- IEEE 754 uses an implied 1 when storing the mantissa. Since in normalized binary scientific form, there is always a single 1 to the left of the radix point, this 1 is not stored in the mantissa field but is rather *implied* in the number. This gives one more bit for precision.
- The exponent of an IEEE floating-point number dictates its range.
- The mantissa of an IEEE floating-point number dictates its precision.
- A single-precision IEEE 754 number uses 32-bits that consist of one sign bit, 8-bits of exponent, and 23-bits of mantissa.
- A single-precision IEEE 754 number has a range of $+/-3.4028235 \times 10^{38}{}_{10}$ and a precision of ~7 significant digits.
- A double-precision IEEE 754 number uses 64-bits that consist of one sign bit, 11-bits of exponent, and 52-bits of mantissa.
- A double-precision IEEE 754 number has a range of $+/-1.7976931348623158 \times 10^{308}{}_{10}$ and a precision of ~16 significant digits.
- The steps to convert from a decimal number into IEEE 754 floating-point format include: (1) convert the decimal number into a fixed-point binary representation; (2) convert the fixed-point number into normalized binary scientific notation; (3) from the binary SN, determine the sign bit; (4) from the binary SN, determine the biased exponent; (5) from the binary SN, determine the mantissa with implied leading 1; and (6) combine the three fields from steps 3–5 into the final 32-bit binary number.
- The steps to convert from an IEEE 754 floating-point format into decimal include: (1) reassemble the original mantissa by adding back in the implied 1; (2) determine the decimal value of the original exponent from the biased exponent; (3) determine whether the number is positive or negative from sign bit; (4) assemble the extracted information from steps 1–3 into binary scientific notation; (5) shift the radix point in the binary SN per the exponent value to get back into fixed-point binary; and (6) convert the fixed-point binary number to decimal.
- The steps to perform addition and subtraction on IEEE 754 numbers follow a similar algorithm as when doing these operations on decimal scientific notation. The numbers are reformatted until the exponents are the same. Then the operations are performed on the mantissas. Then the common exponent is applied to the result and the final value is normalized.
- The steps to perform multiplication and division on IEEE 754 numbers follow a similar algorithm as when doing these operations on decimal scientific notation. The operations are performed on the mantissas. For multiplication, the exponents are added. For division, the exponents are subtracted. The new exponent is then applied to the resulting mantissa and the final value is normalized.
- The standard Verilog language supports operations on the type *real;* however, this type is not synthesizable so it cannot be used for floating-point modeling.
- Most floating-point implementations in Verilog are done using a manual approach where each step is coded at a low level of abstraction.
- Verilog models for floating-point arithmetic can be implemented in steps that follow the algorithms for arithmetic calculations by hand.

Exercise Problems

Section 12.1: Overview of Floating-Point Numbers

12.1.1 How many bits are used to encode an IEEE 754 single-precision number?

12.1.2 How many bits does an IEEE 754 single-precision number use to hold the exponent?

12.1.3 How many bits does an IEEE 754 single-precision number use to hold the mantissa?

12.1.4 What is the smallest normal, positive number that an IEEE 754 single-precision number can hold?

12.1.5 What is the largest normal, positive number that an IEEE 754 single-precision number can hold?

12.1.6 How many bits are used to encode an IEEE 754 double-precision number?

12.1.7 How many bits does an IEEE 754 double-precision number use to hold the exponent?

12.1.8 How many bits does an IEEE 754 double-precision number use to hold the mantissa?

12.1.9 What is the smallest normal, positive number that an IEEE 754 double-precision number can hold?

12.1.10 What is the largest normal, positive number that an IEEE 754 double-precision number can hold?

Section 12.2: IEEE 754 Base Conversions

12.2.1 Convert 10.5_{10} into its IEEE 754 single-precision binary representation.

12.2.2 Convert -55.25_{10} into its IEEE 754 single-precision binary representation.

12.2.3 Convert $7.75 \times 10^{-5}{}_{10}$ into its IEEE 754 single-precision binary representation.

12.2.4 Convert 10.5_{10} into its IEEE 754 double-precision binary representation.

12.2.5 Convert -55.25_{10} into its IEEE 754 double-precision binary representation.

12.2.6 Convert the IEEE 754 single-precision code $42B1C000_{16}$ into decimal.

12.2.7 Convert the IEEE 754 single-precision code $44E90000_{16}$ into decimal.

12.2.8 Convert the IEEE 754 single-precision code $44ECA000_{16}$ into decimal.

12.2.9 Convert the IEEE 754 single-precision code $4382A168_{16}$ into decimal.

12.2.10 Convert the IEEE 754 single-precision code $43B7AEF9_{16}$ into decimal.

Section 12.3: Floating-Point Arithmetic

12.3.1 Calculate the sum of 101_{10} + 261_{10} by hand using IEEE 754 single-precision encoding. Show your work.

12.3.2 Calculate the sum of 261_{10} + 367_{10} by hand using IEEE 754 single-precision encoding. Show your work.

12.3.3 Calculate the sum of (-367_{10}) + 101_{10} by hand using IEEE 754 single-precision encoding. Show your work.

12.3.4 Calculate the difference of 367_{10} – 261_{10} by hand using IEEE 754 single-precision encoding. Show your work.

12.3.5 Calculate the difference of 261_{10} – 101_{10} by hand using IEEE 754 single-precision encoding. Show your work.

12.3.6 Calculate the difference of $(-367_{10}) - 261_{10}$ by hand using IEEE 754 single-precision encoding. Show your work.

12.3.7 Calculate the product of $101_{10} \times 101_{10}$ by hand using IEEE 754 single-precision encoding. Show your work.

12.3.8 Calculate the product of $(-261_{10}) \times 367_{10}$ by hand using IEEE 754 single-precision encoding. Show your work.

12.3.9 Calculate the quotient of $367_{10} \div 261_{10}$ by hand using IEEE 754 single-precision encoding. Show your work.

12.3.10 Calculate the quotient of $(-367_{10}) \div 101_{10}$ by hand using IEEE 754 single-precision encoding. Show your work.

Section 12.4: Floating-Point Modeling in Verilog

12.4.1 When modeling addition and subtraction of floating-point numbers in Verilog, overflow is a concern when operating on the mantissas. What are two ways to address this concern?

12.4.2 Design and simulate a Verilog model for a double-precision floating-point *addition* circuit.

12.4.3 Design and simulate a Verilog model for a double-precision floating-point *subtraction* circuit.

12.4.4 Design and simulate a Verilog model for a double- precision floating-point *multiplication* circuit.

12.4.5 Design and simulate a Verilog model for a double-precision floating-point *division* circuit.

Appendix A: List of Worked Examples

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2

Index

B.J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-031-44104-2

T

V

W

Y