Chapter 5: Verilog (Part 1)

Based on the material presented in Chap. 4, there are a few observations about logic design that are apparent. First, the size of logic circuitry can scale quickly to the point where it is difficult to design by hand. Second, the process of moving from a high-level description of how a circuit works (e.g., a truth table) to a form that is ready to be implemented with real circuitry (e.g., a minimized logic diagram) is straightforward and well-defined. Both of these observations motivate the use of computer-aided design (CAD) tools to accomplish logic design. This chapter introduces hardware description languages (HDLs) as a means 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 that were presented in Chap. 4. The intent of this chapter is to introduce HDLs and their use in the modern digital design flow. This chapter will cover the basics of designing combinational logic in an HDL and also hierarchical design. The more advanced concepts of HDLs such as sequential logic design, high-level abstraction, and test benches are covered later so that the reader can get started quickly using HDLs to gain experience with the languages and design flow.

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 more similar to the programming language C and less strict in its type casting than VHDL. Verilog is also widely used in custom integrated circuit design so there is a great deal of documentation and examples readily available online. The goal of this chapter is to provide an understanding of the basic principles of hardware description languages.

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

5.1 Describe the role of hardware description languages in modern digital design.
5.2 Describe the fundamentals of design abstraction in modern digital design.
5.3 Describe the modern digital design flow based on hardware description languages.
5.4 Describe the fundamental constructs of Verilog.
5.5 Design a Verilog model for a combinational logic circuit using concurrent modeling techniques (continuous signal assignment with logical operators and continuous signal assignment with conditional operators).
5.6 Design a Verilog model for a combinational logic circuit using a structural design approach (gate-level primitives and user-defined primitives).
5.7 Describe the role of a Verilog test bench.

B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with Verilog*,
https://doi.org/10.1007/978-3-030-13605-5_5

5.1 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 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. Simultaneously there was a need to begin simulating the functionality of these large systems prior to fabrication to verify accuracy. Due to the complexity of these systems and the vast potential for design error, it became impractical to verify design accuracy through prototyping.

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 is 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 most 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 5.1 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.

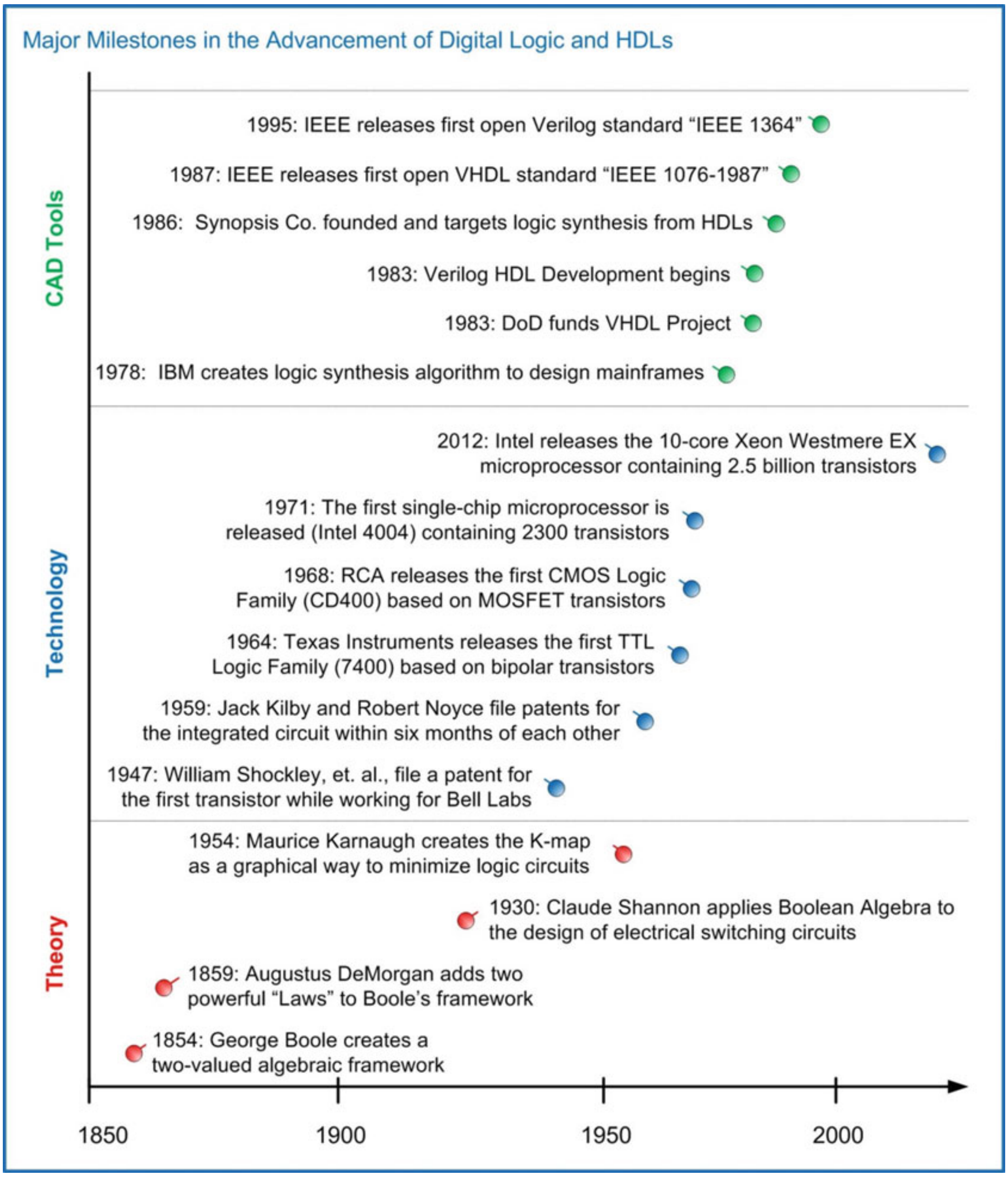

Major Milestones in the Advancement of Digital Logic and HDLs
CAD Tools
1995: IEEE releases first open Verilog standard "IEEE 1364"
1987: IEEE releases first open VHDL standard "IEEE 1076-1987"
1986: Synopsis Co. founded and targets logic synthesis from HDLs
1983: Verilog HDL Development begins
1983: DoD funds VHDL Project
1978: IBM creates logic synthesis algorithm to design mainframes
Technology
2012: Intel releases the 10-core Xeon Westmere EX microprocessor containing 2.5 billion transistors
1971: The first single-chip microprocessor is released (Intel 4004) containing 2300 transistors
1968: RCA releases the first CMOS Logic Family (CD400) based on MOSFET transistors
1964: Texas Instruments releases the first TTL Logic Family (7400) based on bipolar transistors
1959: Jack Kilby and Robert Noyce file patents for the integrated circuit within six months of each other
1947: William Shockley, et. al., file a patent for the first transistor while working for Bell Labs
Theory
1954: Maurice Karnaugh creates the K-map as a graphical way to minimize logic circuits
1930: Claude Shannon applies Boolean Algebra to the design of electrical switching circuits
1859: Augustus DeMorgan adds two powerful "Laws" to Boole's framework
1854: George Boole creates a two-valued algebraic framework
1850
1900
1950
2000

Fig. 5.1
Major milestones in the advancement of digital logic and HDLs

Concept Check

CC5.1 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.

5.2 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 5.2 shows a graphical depiction of the different layers of abstraction in digital system design.

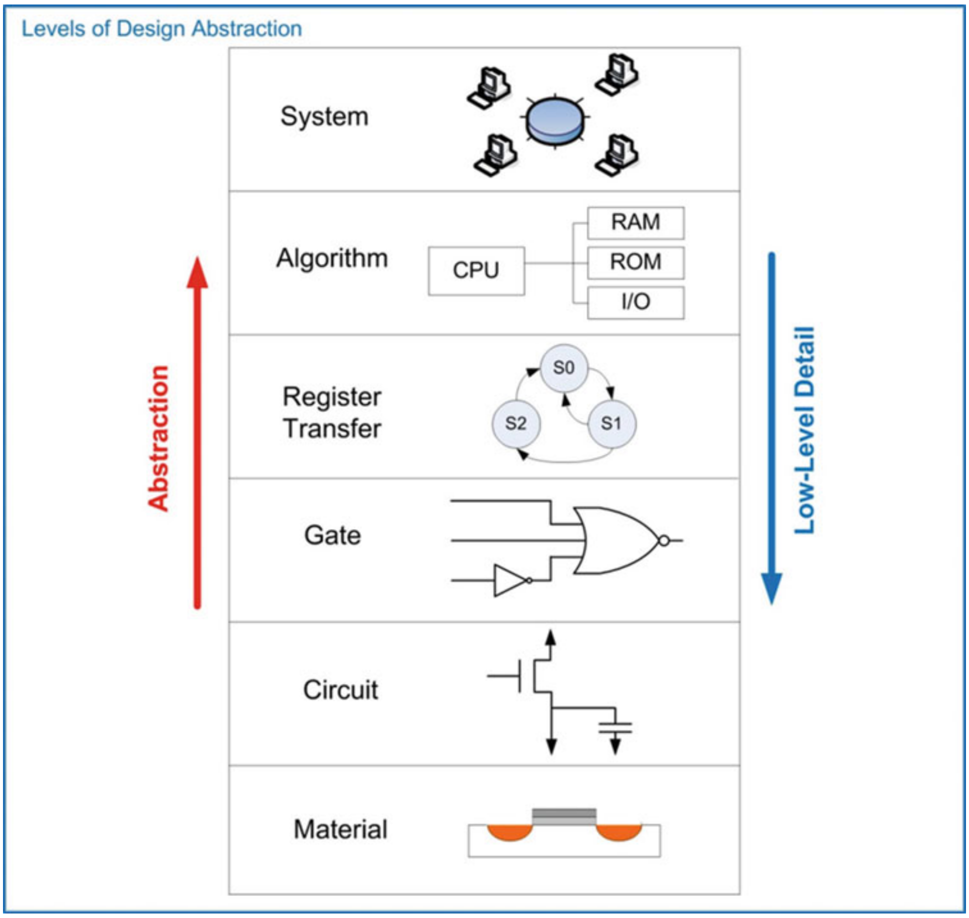


Fig. 5.2
Levels of design abstraction

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 sub-systems, 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 sub-systems 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 is moved between and within sub-systems are described in addition to how the data is 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. 5.3.

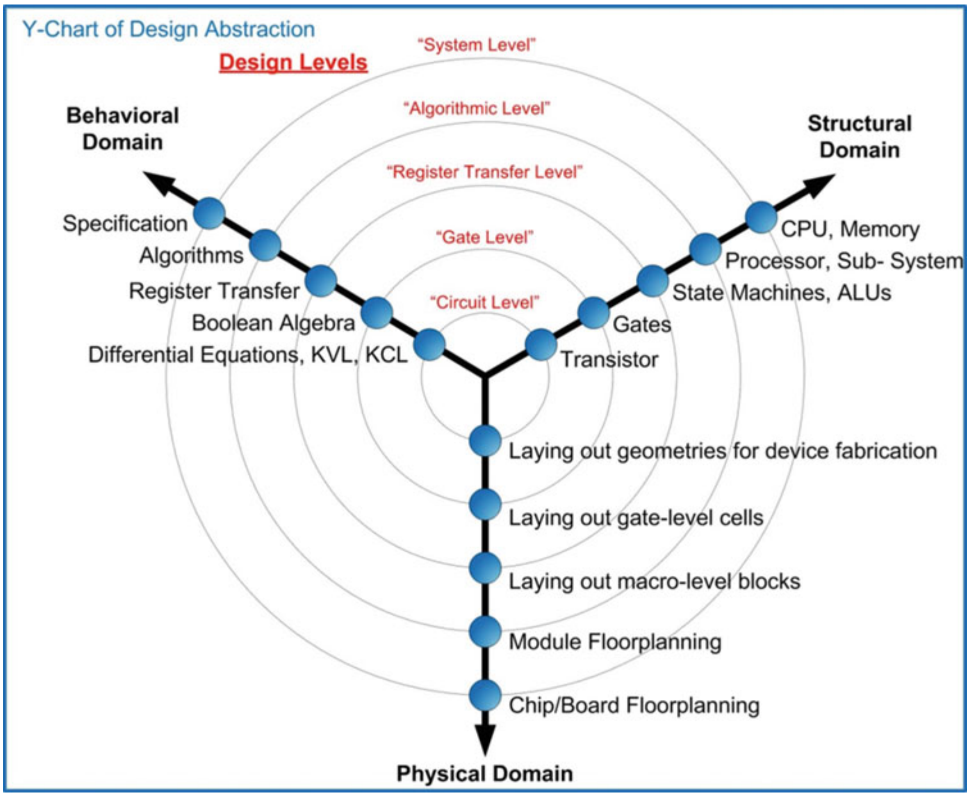


Fig. 5.3
Y-chart of design abstraction

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 5.4 shows the top-down design process depicted as an inward spiral on the Y-chart.

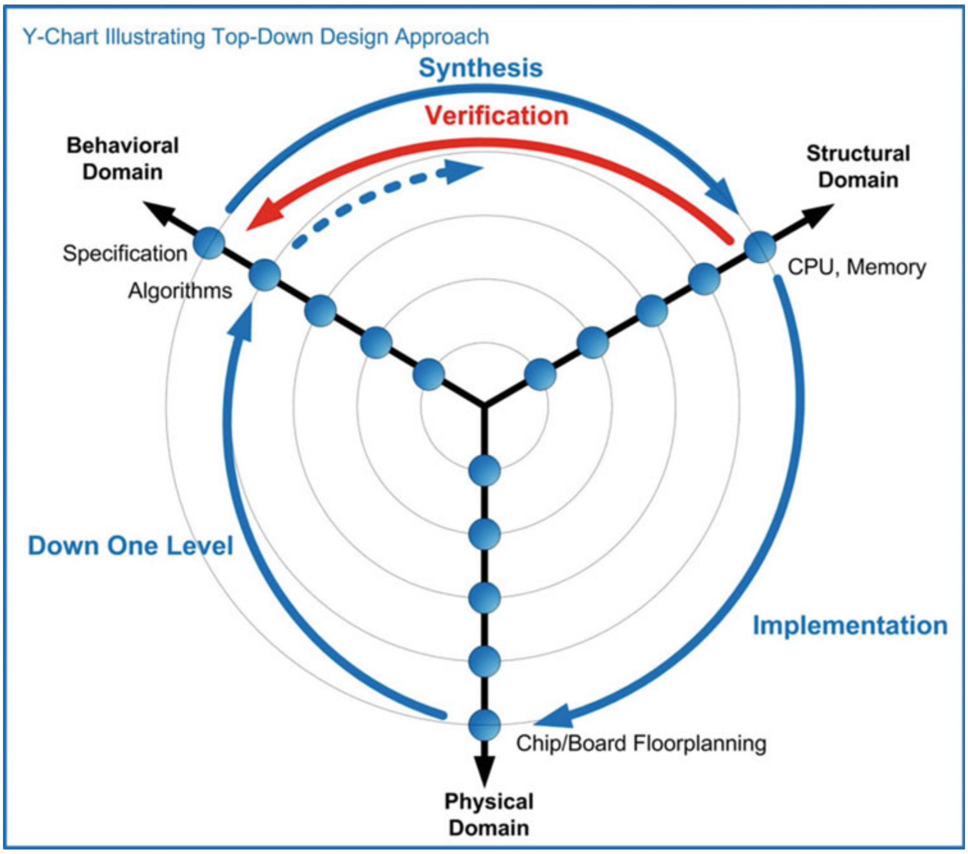


Fig. 5.4
Y-chart illustrating top-down design approach

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

CC5.2 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.

5.3 The Modern Digital Design Flow

When performing a smaller design or the design of fully contained sub-systems, the process can be broken down into individual steps. These steps are shown in Fig. 5.5. 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.

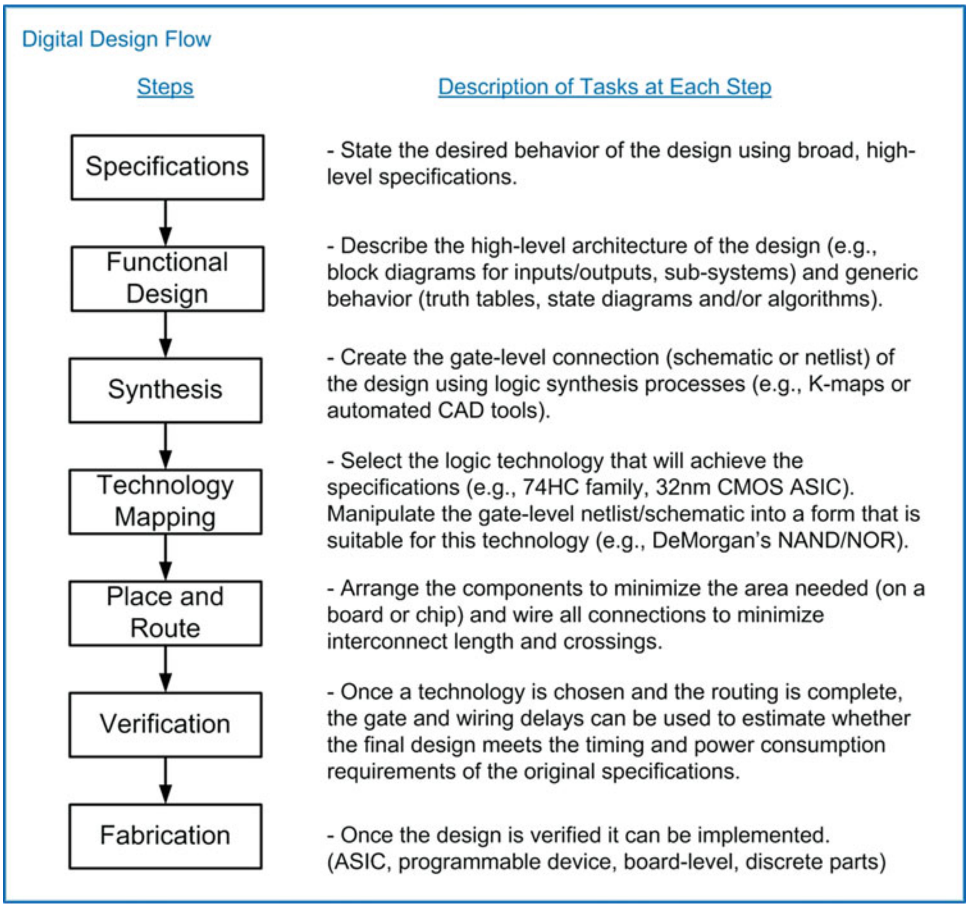

Digital Design Flow
Steps
Description of Tasks at Each Step
Specifications
- State the desired behavior of the design using broad, high-level specifications.
Functional Design
- Describe the high-level architecture of the design (e.g., block diagrams for inputs/outputs, sub-systems) and generic behavior (truth tables, state diagrams and/or algorithms).
Synthesis
- Create the gate-level connection (schematic or netlist) of the design using logic synthesis processes (e.g., K-maps or automated CAD tools).
Technology Mapping
- Select the logic technology that will achieve the specifications (e.g., 74HC family, 32nm CMOS ASIC). Manipulate the gate-level netlist/schematic into a form that is suitable for this technology (e.g., DeMorgan's NAND/NOR).
Place and Route
- Arrange the components to minimize the area needed (on a board or chip) and wire all connections to minimize interconnect length and crossings.
Verification
- Once a technology is chosen and the routing is complete, the gate and wiring delays can be used to estimate whether the final design meets the timing and power consumption requirements of the original specifications.
Fabrication
- Once the design is verified it can be implemented. (ASIC, programmable device, board-level, discrete parts)

Fig. 5.5
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 5.6 shows how this flow is used in the classical design approach of a combinational logic circuit.

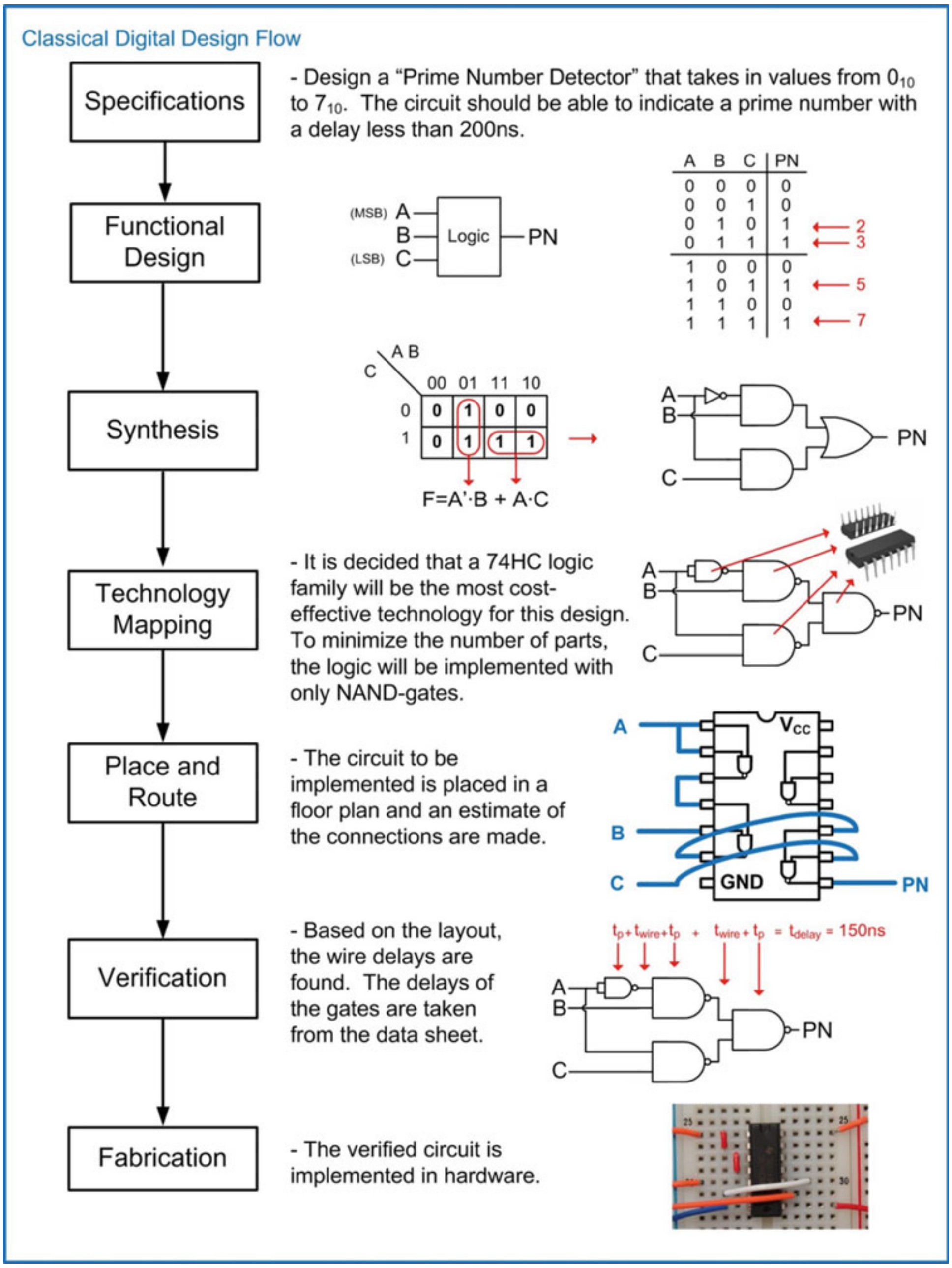

Classical Digital Design Flow
Specifications
- Design a "Prime Number Detector" that takes in values from 0₁₀ to 7₁₀. The circuit should be able to indicate a prime number with a delay less than 200ns.
Functional Design
(MSB) A
B
(LSB) C
Logic
PN
A B C | PN
0 0 0 | 0
0 0 1 | 0
0 1 0 | 1 ← 2
0 1 1 | 1 ← 3
1 0 0 | 0
1 0 1 | 1 ← 5
1 1 0 | 0
1 1 1 | 1 ← 7
Synthesis
C \ AB 00 01 11 10
0: 0 1 0 0
1: 0 1 1 1
F=A'·B + A·C
Technology Mapping
- It is decided that a 74HC logic family will be the most cost-effective technology for this design. To minimize the number of parts, the logic will be implemented with only NAND-gates.
Place and Route
- The circuit to be implemented is placed in a floor plan and an estimate of the connections are made.
Vcc
GND
Verification
- Based on the layout, the wire delays are found. The delays of the gates are taken from the data sheet.
tp+twire+tp + twire+tp = tdelay = 150ns
Fabrication
- The verified circuit is implemented in hardware.

Fig. 5.6
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 5.7 shows the modern digital design flow with the inclusion of simulation capability at each step.

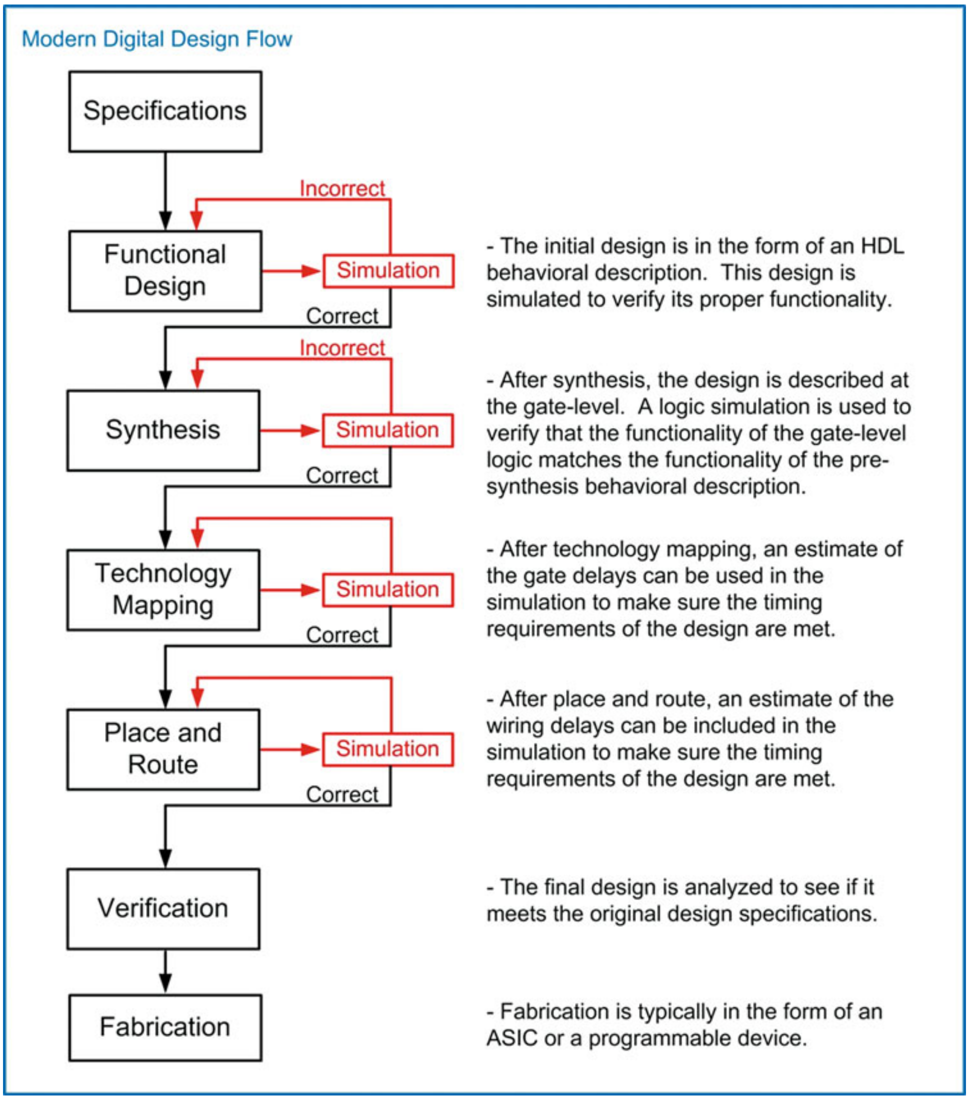

Modern Digital Design Flow
Specifications
Incorrect
Functional Design
Simulation
Correct
Incorrect
Synthesis
Simulation
Correct
Technology Mapping
Simulation
Correct
Place and Route
Simulation
Correct
Verification
Fabrication
- The initial design is in the form of an HDL behavioral description. This design is simulated to verify its proper functionality.
- After synthesis, the design is described at the gate-level. A logic simulation is used to verify that the functionality of the gate-level logic matches the functionality of the pre-synthesis behavioral description.
- After technology mapping, an estimate of the gate delays can be used in the simulation to make sure the timing requirements of the design are met.
- After place and route, an estimate of the wiring delays can be included in the simulation to make sure the timing requirements of the design are met.
- The final design is analyzed to see if it meets the original design specifications.
- Fabrication is typically in the form of an ASIC or a programmable device.

Fig. 5.7
Modern digital design flow

Concept Check

CC5.3 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.

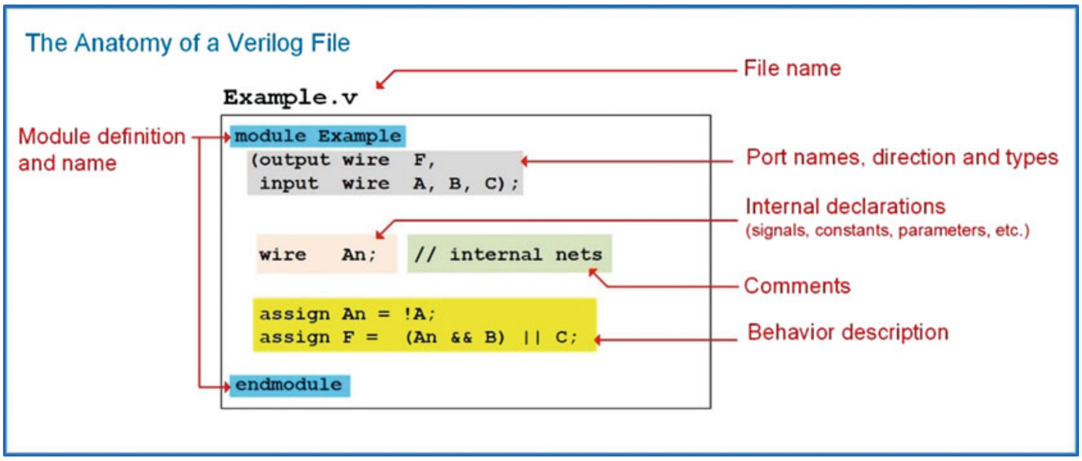
C) Schematics symbols became too small as designs became larger.

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

5.4 Verilog Constructs

Now we begin looking at the details of Verilog. 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 corrections and 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. This chapter gives an overview of the basic constructs of Verilog in order to model simple combinational logic circuits and begin gaining experience with logic simulations. The more advanced constructs of Verilog are covered in Chap. 8 with examples given throughout Chaps. 9, 10, 11, 12, and 13.

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 5.8 shows a graphical depiction of a Verilog file.


The Anatomy of a Verilog File
File name
Example.v
Module definition and name
Port names, direction and types
Internal declarations (signals, constants, parameters, etc.)
Comments
Behavior description

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

   wire   An;   // internal nets

   assign An =  !A;
   assign F =   (An && B) || C;

endmodule
```

Fig. 5.8
The anatomy of a Verilog file

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.

5.4.1 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 pre-defined 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.

5.4.1.1 Value Set

Verilog supports four basic values that a signal can take on: 0, 1, X, and Z. Most of the pre-defined 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.

5.4.1.2 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}, (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.

5.4.1.3 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, 2's complement variable representing whole numbers between $-2{,}147{,}483{,}648_{10}$ and +2,147,483,647.
real	A 64-bit, floating point variable representing real numbers between $-(2.2 \times 10^{-308})_{10}$ and $+(2.2 \times 10^{308})_{10}$.
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.

5.4.1.4 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.
```

5.4.1.5 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.
```

5.4.1.6 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., z or Z, 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.
```

5.4.1.7 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 group 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.
```

5.4.2 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
```

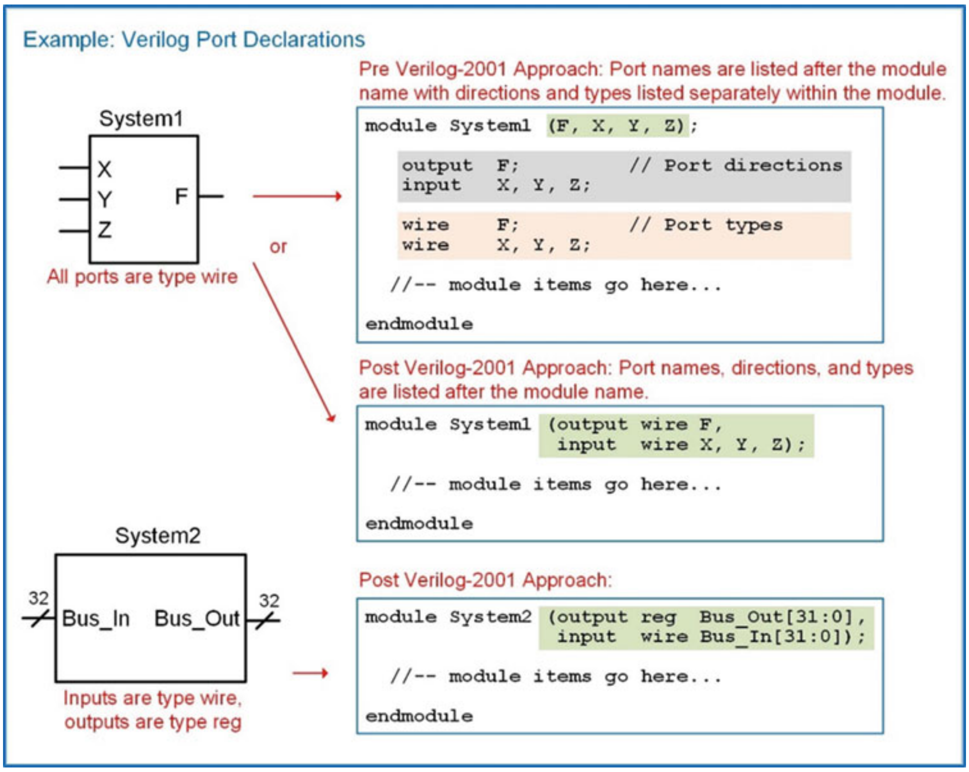
or

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

5.4.2.1 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 5.1 shows multiple approaches for defining a module and its ports.


Example: Verilog Port Declarations
System1
X
Y
Z
F
All ports are type wire
or
System2
32
Bus_In
Bus_Out
32
Inputs are type wire, outputs are type reg

Pre Verilog-2001 Approach: Port names are listed after the module name with directions and types listed separately within the module.

```
module System1 (F, X, Y, Z);

  output  F;            // Port directions
  input   X, Y, Z;

  wire    F;            // Port types
  wire    X, Y, Z;

 //-- module items go here...

endmodule
```

Post Verilog-2001 Approach: Port names, directions, and types are listed after the module name.

```
module System1 (output wire F,
                input  wire X, Y, Z);

 //-- module items go here...

endmodule
```

Post Verilog-2001 Approach:

```
module System2 (output reg  Bus_Out[31:0],
                input  wire Bus_In[31:0]);

 //-- module items go here...

endmodule
```

Example 5.1
Declaring Verilog module ports

5.4.2.2 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 sub-system as those at a higher level without conflict. Figure 5.9 shows an example of legal signal naming in a hierarchical design.

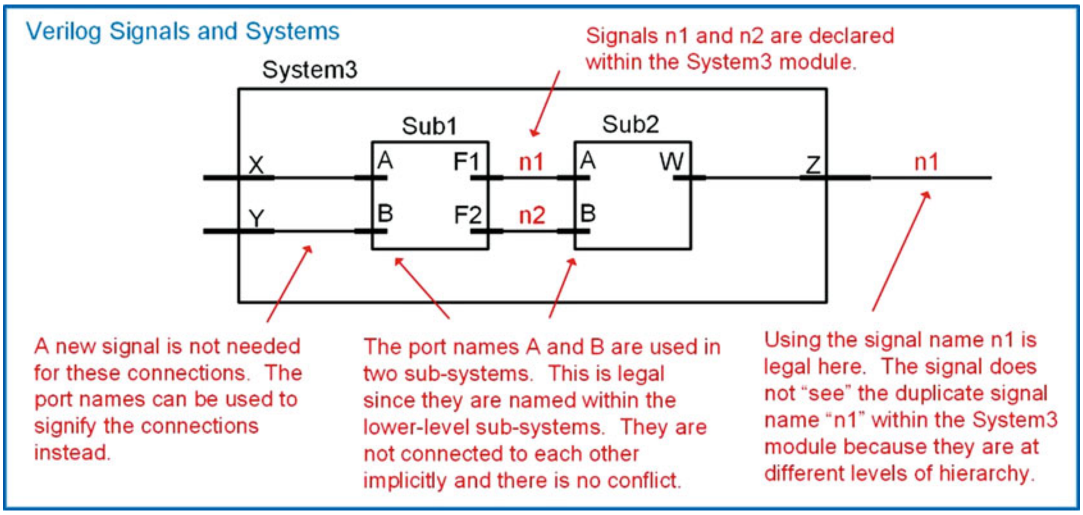

Verilog Signals and Systems
System3
Signals n1 and n2 are declared within the System3 module.
Sub1
Sub2
X
Y
A
B
F1
F2
n1
n2
A
B
W
Z
n1
A new signal is not needed for these connections. The port names can be used to signify the connections instead.
The port names A and B are used in two sub-systems. This is legal since they are named within the lower-level sub-systems. They are not connected to each other implicitly and there is no conflict.
Using the signal name n1 is legal here. The signal does not "see" the duplicate signal name "n1" within the System3 module because they are at different levels of hierarchy.

Fig. 5.9
Verilog signals and systems

5.4.2.3 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 = 64;
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
```

5.4.2.4 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.
```

5.4.3 Verilog Operators

There are a variety of pre-defined 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.

5.4.3.1 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
```

5.4.3.2 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
```

5.4.3.3 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
```

5.4.3.4 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
```

5.4.3.5 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
```

5.4.3.6 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).
```

5.4.3.7 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.
```

5.4.3.8 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};
```

5.4.3.9 Numerical Operators

Verilog also provides a set of numerical operators as follows:

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2's 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.

5.4.3.10 Operator Precedence

The following is the order of precedence of the Verilog operators:

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

CC5.4(a) What revision of Verilog added the ability to list the port names, types, and directions just once after the module name?

A) Verilog-1995.

B) Verilog-2001.

C) Verilog-2005.

D) SystemVerilog.

CC5.4(b) What is the difference between types wire and reg?

A) They are the same.

B) The type wire is a simple interconnection while reg will hold the value of its last assignment.

C) The type wire is for scalars while the type reg is for vectors.

D) Only wire is synthesizable.

5.5 Modeling Concurrent Functionality in Verilog

It is important to remember that Verilog is a hardware description language, 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.

5.5.1 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
```

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 and 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.

5.5.2 Continuous Assignment with Logical Operators

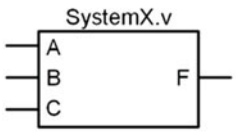
Each of the logical operators described in Sect. 5.4.3.2 can be used in conjunction with concurrent signal assignments to create individual combinational logic circuits. Example 5.2 shows how to design a Verilog model of a combinational logic circuit using this approach.

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

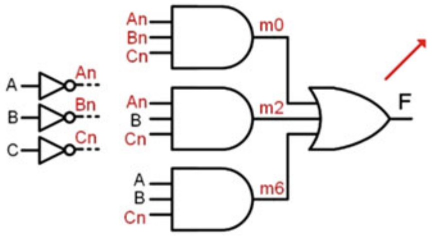
Let's call the module *SystemX*. First, let's declare the ports. The module will have three inputs (A, B, C) and one output (F). We'll use the type wire for all inputs/outputs so that this will synthesize directly into real circuitry.



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 5.2
Modeling combinational logic using continuous assignment with logical operators

5.5.3 Continuous Assignment with Conditional Operators

Logical operators are good for describing the behavior of small circuits; however, in the prior example, we still needed to create the canonical sum of products logic expression by hand before describing the functionality in Verilog. 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 affect the values of the result. In this approach, we use the conditional operator (**?**) in conjunction with the continuous assignment keyword **assign**. Example 5.3 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 5.2 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.

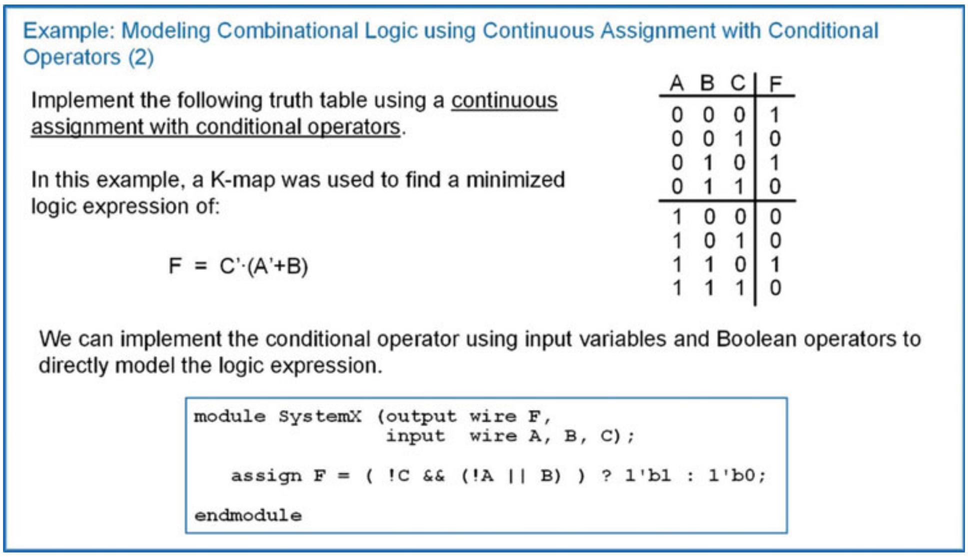
```
module SystemX (output wire F,
                input  wire A, B, C);

   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 5.3
Modeling 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 5.4 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 5.2 and 5.3 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'·(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 5.4
Modeling combinational logic using continuous assignment with conditional operators (2)

5.5.4 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. Parentheses are optional but recommended when using multiple delay parameters.

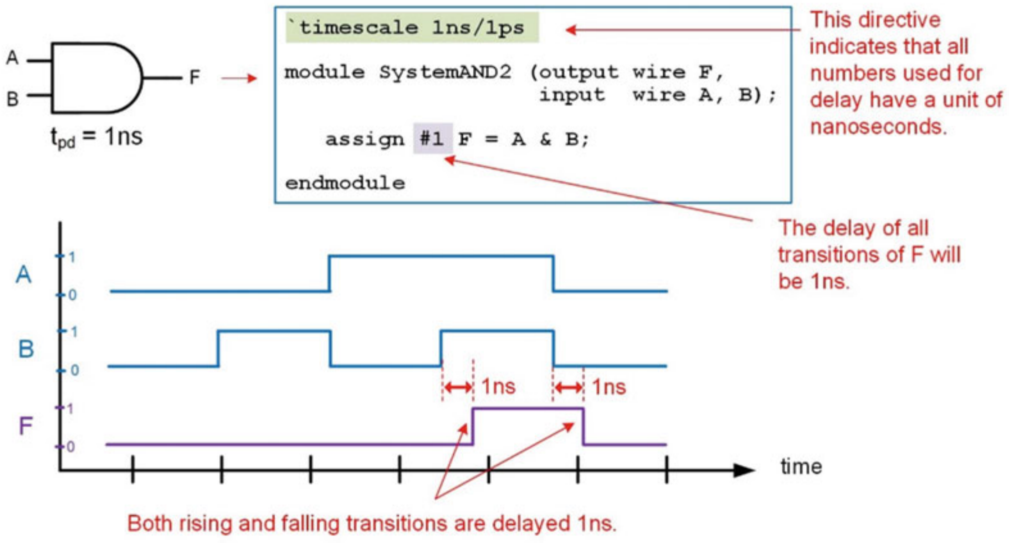
```
assign#(<del_all>)                    <target_net>=<RHS_nets,operators,etc...>;
assign#(<del_rise,del_fall>)          <target_net>=<RHS_nets,operators,etc...>;
assign#(<del_rise,del_fall,del_off>)<target_net>=<RHS_nets,operators,etc...>;
```

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
transitions.
```

When using delay, it is typical to include the **`timescale** directive to provide the units of the delay being specified. Example 5.5 shows a graphical depiction of using delay with continuous assignments when modeling combinational logic circuits.

Example: Modeling Delay in Continuous Assignments

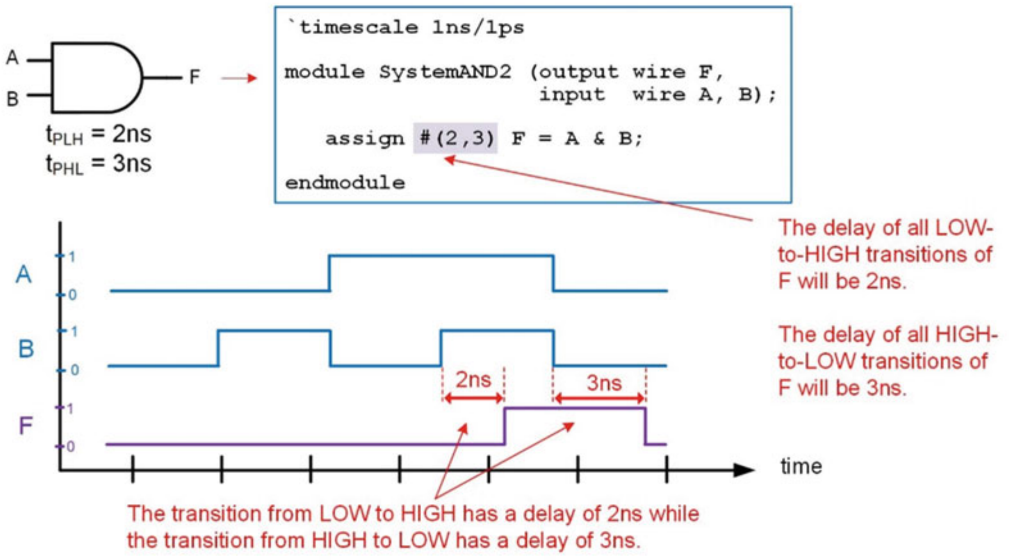

A
B
F
t_{pd} = 1ns
This directive indicates that all numbers used for delay have a unit of nanoseconds.
The delay of all transitions of F will be 1ns.
A
B
F
1
0
1ns
1ns
time
Both rising and falling transitions are delayed 1ns.

```
`timescale 1ns/1ps

module SystemAND2 (output wire F,
                   input  wire A, B);

   assign #1 F = A & B;

endmodule
```


A
B
F
t_{PLH} = 2ns
t_{PHL} = 3ns
The delay of all LOW-to-HIGH transitions of F will be 2ns.
The delay of all HIGH-to-LOW transitions of F will be 3ns.
A
B
F
1
0
2ns
3ns
time
The transition from LOW to HIGH has a delay of 2ns while the transition from HIGH to LOW has a delay of 3ns.

```
`timescale 1ns/1ps

module SystemAND2 (output wire F,
                   input  wire A, B);

   assign #(2,3) F = A & B;

endmodule
```

Example 5.5
Modeling delay in continuous assignments

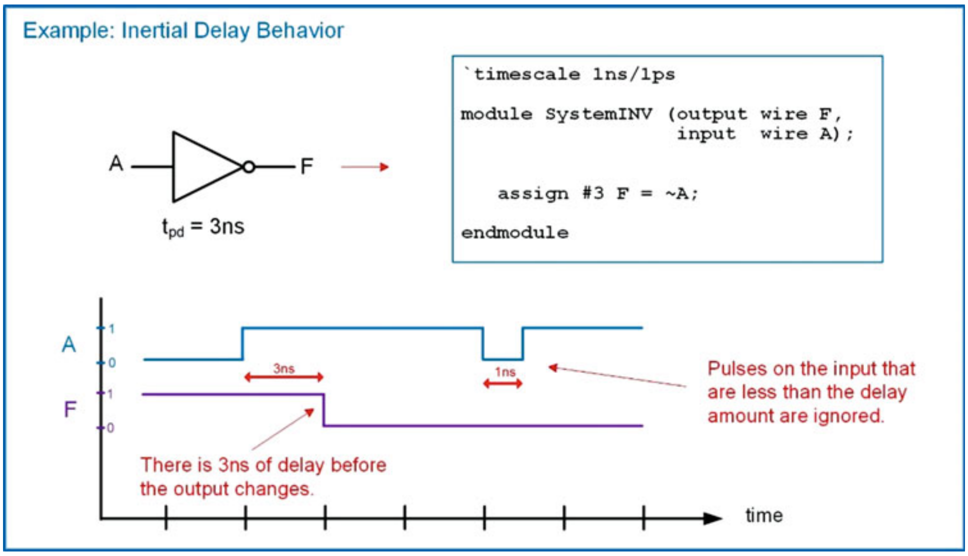
Verilog also provides a mechanism to model a range of delays that are selected by a switch set in the CAD compiler. There are three delays categories that can be specified: *minimum*, *typical*, and *maximum*. The delays are separated by a ":". The following is the syntax of how to use the delay range 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 5.6 shows a graphical depiction of inertial delay behavior in Verilog.


Example: Inertial Delay Behavior
A
F
t_{pd} = 3ns
A
F
1
0
1
0
3ns
1ns
Pulses on the input that are less than the delay amount are ignored.
There is 3ns of delay before the output changes.
time

```
`timescale 1ns/1ps

module SystemINV (output wire F,
                  input  wire A);

   assign #3 F = ~A;

endmodule
```

Example 5.6
Inertial delay modeling when using continuous assignment

CONCEPT CHECK

CC5.5(a) Why is concurrency such an important concept in HDLs?

A) Concurrency is a feature of HDLs that can't be modeled using schematics.

B) Concurrency allows automated synthesis to be performed.

C) Concurrency allows logic simulators to display useful system information.

D) Concurrency is necessary to model real systems that operate in parallel.

CC5.5(b) Why does modeling combinational logic in its canonical form with continuous assignment with 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 doesn't 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 doesn't 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 doesn't allow an *else* clause to cover the outputs for any remaining input codes not explicitly listed.

5.6 Structural Design and Hierarchy

Structural design in Verilog refers to including lower-level sub-systems 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 sub-systems. A sub-system 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.

5.6.1 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. An 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*.

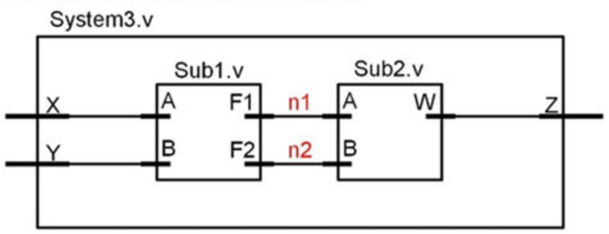
5.6.1.1 Explicit Port Mapping

In explicit port mapping, the names of the ports of the lower-level sub-system 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 parentheses. 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<instanceidentifier>(.port_name1(signal1), .port_name2(signal2),etc.);
```

Example 5.7 shows how to design a Verilog model of a hierarchical system that consists of two lower-level modules.

Verilog Structural Design using Explicit Port Mapping



```
module Sub1 (output wire F1, F2,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module Sub2 (output wire W,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module System3 (output wire Z,
                input  wire X, Y);

   wire n1, n2;

   Sub1 U0 (.F1(n1), .F2(n2), .A(X),  .B(Y));
   Sub2 U1 (.W(Z),   .A(n1),  .B(n2));

endmodule
```

The lower-level port name is explicitly listed (preceded by a period).

The signal being connected to the lower-level port is listed inside of parenthesis.

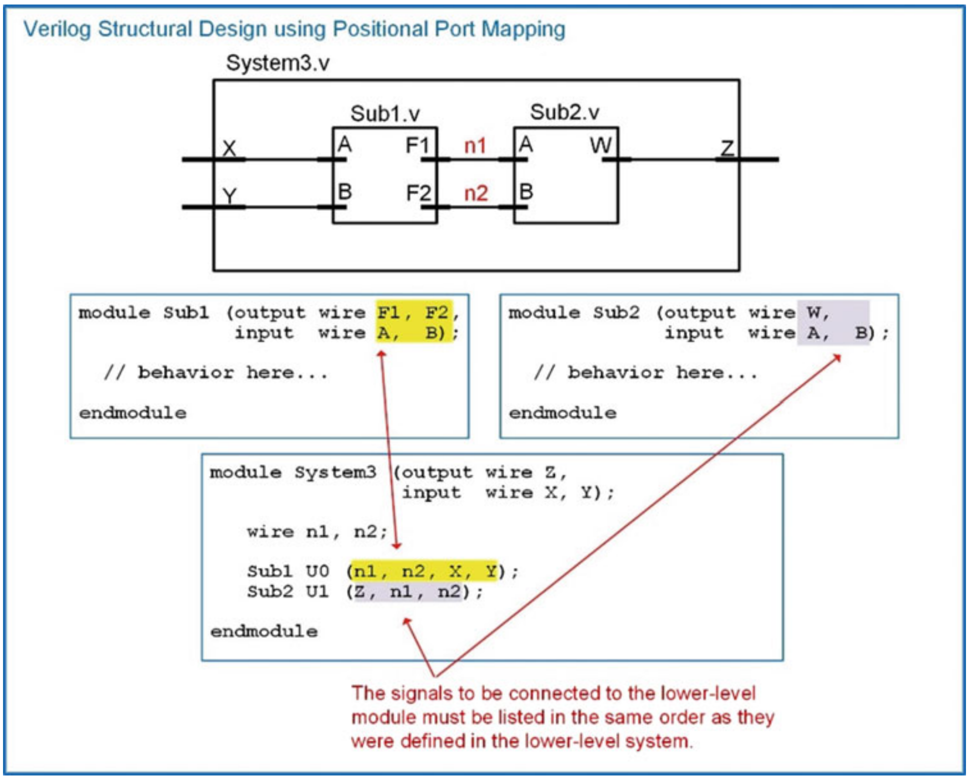
Example 5.7
Verilog structural design using explicit port mapping

5.6.1.2 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 sub-system. 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 5.8 shows how to create the same structural Verilog model as in Example 5.7, but using positional port mapping instead.



```
module Sub1 (output wire F1, F2,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module Sub2 (output wire W,
             input  wire A,  B);

  // behavior here...

endmodule
```

```
module System3 (output wire Z,
                input  wire X, Y);

   wire n1, n2;

   Sub1 U0 (n1, n2, X, Y);
   Sub2 U1 (Z, n1, n2);

endmodule
```

The signals to be connected to the lower-level module must be listed in the same order as they were defined in the lower-level system.

Example 5.8
Verilog structural design using positional port mapping

5.6.2 Gate-Level Primitives

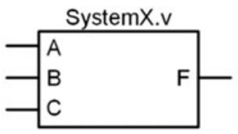
Verilog provides the ability to model basic logic functionality through the use of *primitives*. A primitive is a logic operation that is simple enough that it doesn't require explicit modeling. An example of this behavior can be a basic logic gate or even a truth table. Verilog provides a set of *gate-level primitives* to model simple logic operations. These gate-level primitives are **not()**, **and()**, **nand()**, **or()**, **nor()**, **xor()**, and **xnor()**. Each of these primitives is instantiated as lower-level sub-systems 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 is 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 5.9 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 <u>gate level primitives</u>.

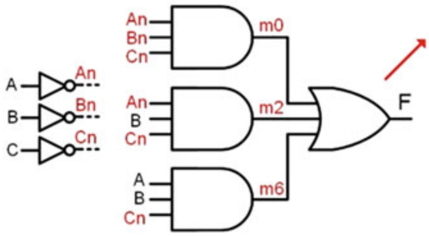
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

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.



```
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
```

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

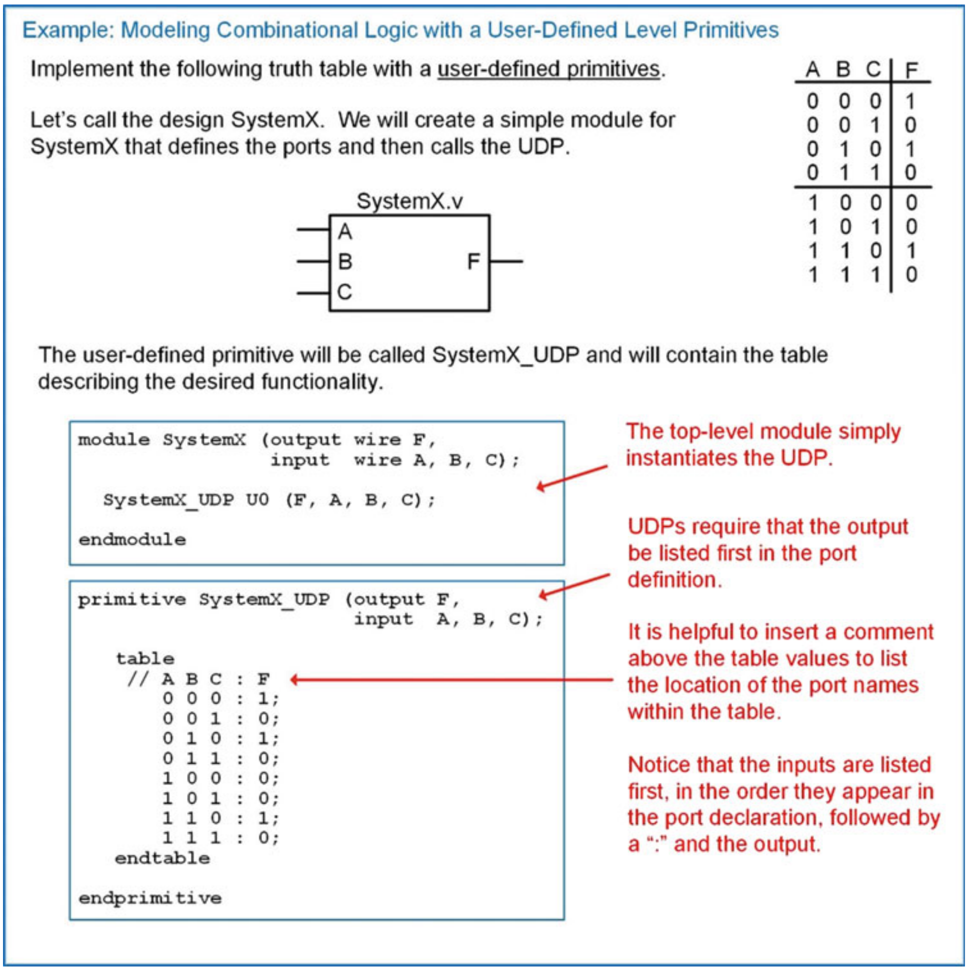
Example 5.9
Modeling combinational logic circuits using gate-level primitives

5.6.3 User-Defined Primitives

A **user-defined primitive** (UDP) is a system that describes the behavior of a low-level component 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 sub-systems that are intended to be instantiated in higher-level modules just like gate-level primitives, with the exception that the UPD 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 5.10 shows how to design a user-defined primitive to implement a combinational logic circuit.

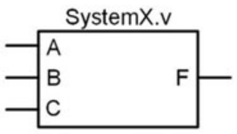


Example: Modeling Combinational Logic with a User-Defined Level Primitives

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

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

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



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 5.10
Modeling combinational logic circuits with a user-defined primitive

5.6.4 Adding Delay to Primitives

Delay can be added to primitives using the same approach as described in Sect. 5.5.4. 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

CC5.6 Does the use of lower-level sub-modules 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 sub-systems whose behavior runs in parallel just as if separate parts were placed in a design.

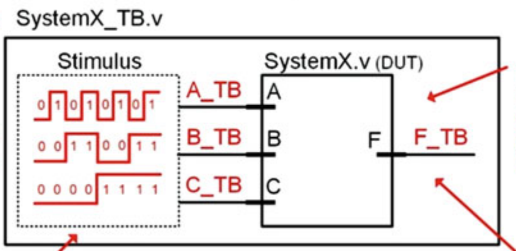
5.7 Overview of Simulation Test Benches

One of the essential components of the modern digital design flow is verifying functionality through simulation. This simulation takes place at many levels of abstraction. For a system to be tested, there needs to be a mechanism to generate input patterns to drive the system and then observe the outputs to verify correct operation. The mechanism to do this in Verilog is called a *test bench*. 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 test bench generates the input conditions and drives them into the input ports of the system being tested. Verilog contains numerous methods to generate stimulus patterns. Since a test bench will not be synthesized, very abstract behavioral modeling can be used to generate the inputs. The output of the system can be viewed as a waveform in a simulation tool. Verilog also has the ability to check the outputs against expected results and notify the user if differences occur. Figure 5.10 gives an overview of how test benches are used in Verilog. The techniques to generate the stimulus patterns are covered in Chap. 8.

Overview of Verilog Test Benches

A test bench is used to drive in signals and observe the outputs of a "device under test" or DUT. A test bench has no inputs or outputs. It calls the DUT as a lower-level module and then generates the inputs to verify its functionality.

The test bench is typically named the same as the DUT but with "_TB" at the end.



The design to be tested is instantiated in the test bench. Signals are declared to connect to the ports of the DUT.

Stimulus patterns are generated in the test bench and driven into the DUT. The patterns should cover every possible input condition.

The output of the DUT can be viewed as a waveform in a simulation tool. Verilog also has constructs to perform automated checking against a description of the expected outputs.

```
module SystemX_TB ();

  reg  A_TB, B_TB, C_TB;
  wire F_TB;

  SystemX DUT (F_TB, A_TB, B_TB, C_TB);

  // Stimulus Generation to Drive A_TB, B_TB and C_TB (covered in Ch. 8)

  // Automated Output Checking & Reporting for F_TB  (covered in Ch. 8)

endmodule
```

Fig. 5.10
Overview of Verilog test benches

Concept Check

CC5.7 How can the output of a DUT be verified when it is connected to a signal that does not go anywhere?

A) It can't. 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 can't. A signal that does not go anywhere will cause an error when the Verilog file is compiled.

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.
- ❖ In the 1980s, two major HDLs emerged, VHDL and Verilog. VHDL was sponsored by the Department of Defense, while Verilog was driven by the commercial industry. Both were later standardized by IEEE.
- ❖ 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.
- ❖ 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 in keeping track of each instantiation. The ports of the component can be connected using either *explicit* or *positional port mapping*.
- ❖ *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 systems. Complex logic circuits can be implemented by using continuous assignment with *logical operators* or *conditional operators*.
- ❖ Verilog sub-systems are also treated as concurrent sub-systems.
- ❖ Delay can be modeled in Verilog for all transitions, or for individual transitions (rise, fall, off). A range of delays can also be provided (min:typ:max). Delay can be added to continuous assignments and sub-system instantiations.
- ❖ 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 sub-system.
- ❖ User-defined primitives are supported in Verilog that allow the functionality of a circuit to be described in table form.
- ❖ A *simulation test bench* is a Verilog file that drives stimulus into a device under test (DUT). Test benches do not have inputs or outputs and are not synthesizable.

Exercise Problems

Section 5.1: History of HDLs

5.1.1 What was the original purpose of Verilog?

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

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

Section 5.2: HDL Abstraction

5.2.1 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.*

5.2.2 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.*

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

5.2.4 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*.

5.2.5 Give the level of design abstraction that the following statement relates to: *the MOSFETs need to be able to drive at least 8 other loads in this design.*

5.2.6 Give the level of design abstraction that the following statement relates to: *this system will contain 1 host computer and support up to 1000 client computers.*

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

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

5.2.9 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*.

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

Section 5.3: The Modern Digital Design Flow

5.3.1 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.*

5.3.2 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.*

5.3.3 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.*

5.3.4 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.*

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

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

5.3.7 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.*

Section 5.4: Verilog Constructs

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

5.4.2 What portion of the Verilog module describes the inputs and outputs.

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

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

5.4.5 What is the highest strength that a value can take on in Verilog.

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

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

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

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

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

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

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

Section 5.5: Modeling Concurrent Functionality in Verilog

5.5.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

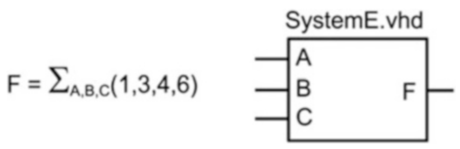


Fig. 5.11
System E functionality

5.5.2 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

5.5.3 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

$F = \prod_{A,B,C}(0,1,3,5,7)$

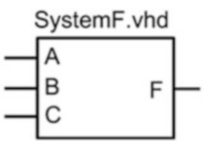


Fig. 5.12
System F functionality

5.5.4 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

5.5.5 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. 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

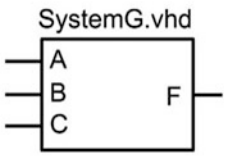


Fig. 5.13
System G functionality

5.5.6 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

5.5.7 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

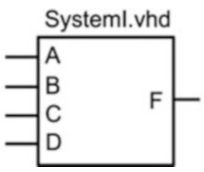


Fig. 5.14
System I functionality

5.5.8 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

5.5.9 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

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

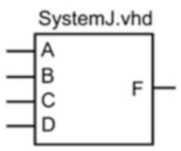


Fig. 5.15
System J functionality

5.5.10 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

5.5.11 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. 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

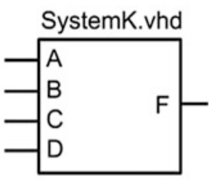


Fig. 5.16
System K functionality

5.5.12 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

Section 5.6: Structural Design in Verilog

5.6.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. 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 sub-system; however, you don't 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.

5.6.2 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. 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 sub-system. 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.

5.6.3 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. 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 sub-system; however, you don't 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.

5.6.4 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. 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 sub-system. 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.

5.6.5 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. 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 sub-system; however, you don't 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.

5.6.6 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. 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 sub-system. 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.

5.6.7 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. 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 sub-system; however, you don't 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.

5.6.8 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. 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 sub-system. 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.

5.6.9 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. 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 sub-system; however, you don't 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.

5.6.10 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. 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 sub-system. 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.

5.6.11 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. 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 sub-system; however, you don't 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.

5.6.12 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. 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 sub-system. 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.

Section 5.7: Overview of Simulation Test Benches

5.7.1 What is the purpose of a test bench?

5.7.2 Does a test bench have input and output ports?

5.7.3 Can a test bench be simulated?

5.7.4 Can a test bench be synthesized?