Chapter 5: VHDL (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 covers 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 adding functionality to an HDL through additional libraries and packages 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 use VHDL to learn the concepts of an HDL. VHDL is stricter in its syntax and typecasting than Verilog, so it is a good platform for beginners as it provides more of a scaffold for the description of circuits. This helps avoid some of the common pitfalls that beginners typically encounter. 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 VHDL.
5.5 Design a VHDL model for a combinational logic circuit using concurrent modeling techniques (signal assignments and logical operators, conditional signal assignments, and selected signal assignments).
5.6 Design a VHDL model for a combinational logic circuit using a structural design approach.
5.7 Describe the role of a VHDL test bench.

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

© Springer International Publishing Switzerland 2017
B.J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*,
DOI 10.1007/978-3-319-34195-8_5

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 that their parts would be compatible with other manufacturers at the physical layer (e.g., voltage and current); however, one challenge that was encountered by the industry was a way to document the complex behavior of larger systems. The use of schematics to document large digital designs became too cumbersome and difficult to understand by anyone besides the designer. Word descriptions of the behavior were easier to understand, but even this form of documentation became too voluminous to be effective for the size of designs that were emerging.

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

Also in 1983, the Verilog HDL was developed by *Automated Integrated Design Systems* as a logic simulation language. The development of Verilog took place completely independent from the VHDL project. Automated Integrated Design Systems (renamed *Gateway Design Automation* in 1985) was acquired by CAD tool vendor *Cadence Design Systems* in 1990. In response to the rapid adoption of the open VHDL standard, Cadence made the Verilog HDL open to the public in order to stay competitive. IEEE once again developed the open standard for this HDL, and in 1995 released the Verilog standard titled IEEE 1364.

The development of CAD tools to accomplish automated logic synthesis can be dated back to the 1970s when IBM began developing a series of practical synthesis engines that were used in the design of their mainframe computers; however, the main advancement in logic synthesis came with the founding of a company called *Synopsis* in 1986. Synopsis was the first company to focus on logic synthesis directly from HDLs. This was a major contribution because designers were already using HDLs to describe and simulate their digital systems, and now logic synthesis became integrated in the same design flow. Due to the complexity of synthesizing highly abstract functional descriptions, only lower levels of abstraction that were thoroughly elaborated were initially able to be synthesized. As CAD tool capability evolved, synthesis of higher levels of abstraction became possible, but even today not all functionality that can be described in an HDL can be synthesized.

The history of HDLs, their standardization, and the creation of the associated logic synthesis tools are key to understanding the use and limitations of HDLs. HDLs were originally designed for documentation and behavioral simulation. Logic synthesis tools were developed independently and modified later to work with HDLs. This history provides some background into the most common pitfalls that beginning digital designers encounter, that being that 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 VHDL 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.

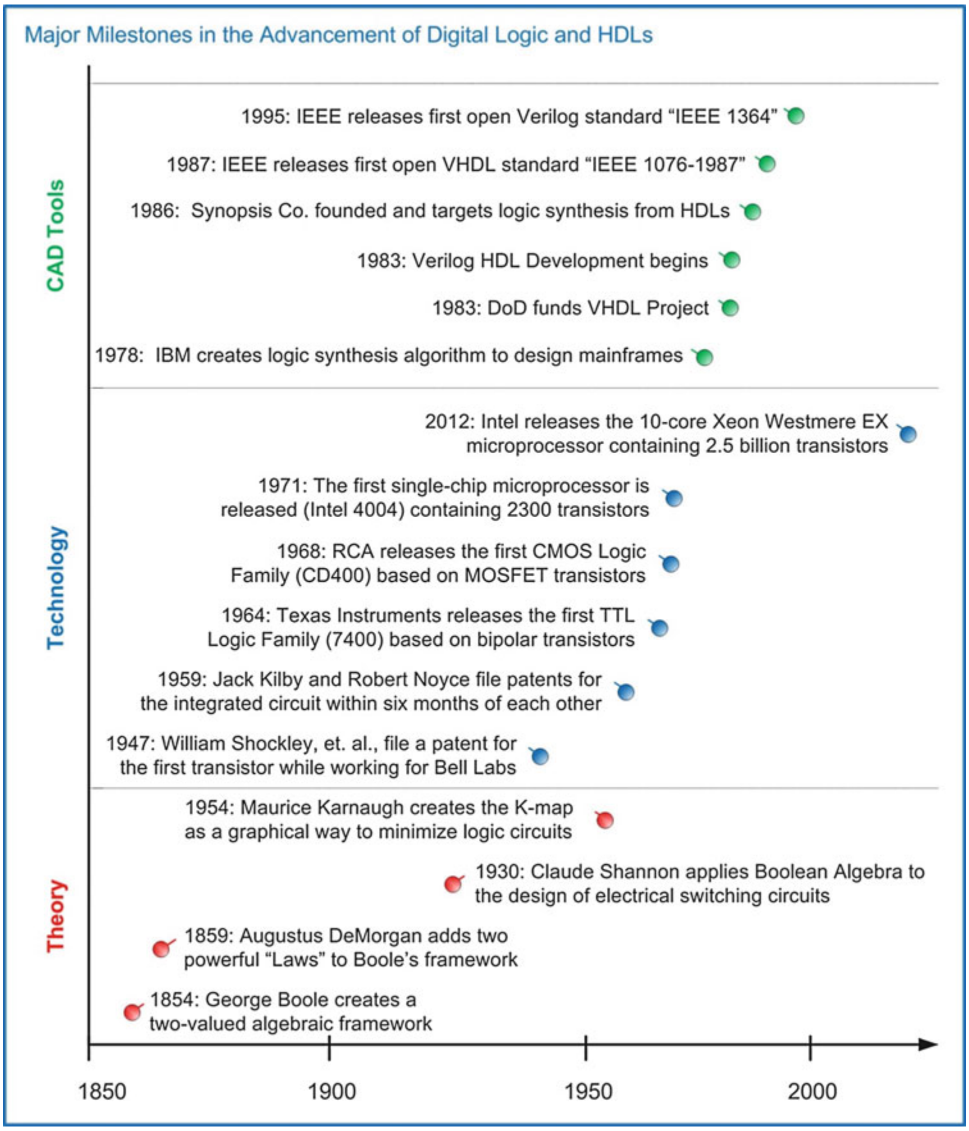


Fig. 5.1
Major milestones in the advancement of digital logic and HDLs

Concept Check

CC5.1 Why does VHDL support modeling techniques that *aren't* synthesizable?

A) Since synthesis wasn't within the original scope of the VHDL project, there wasn't sufficient time to make everything synthesizable.

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

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

D) VHDL 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.

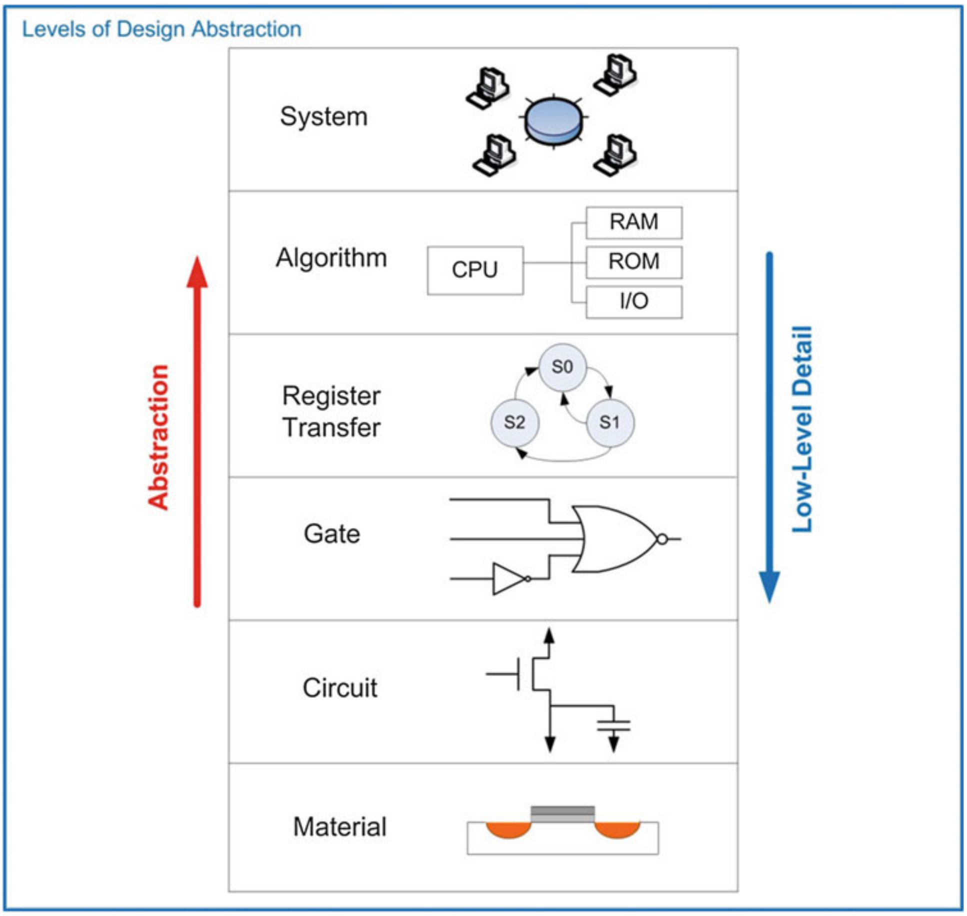


Fig. 5.2
Levels of design abstraction

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 subsystems, each with an associated behavior that will accomplish a part of the primary task. At this level, the example computer specifications might be broken down into subsystems such as a central processing unit (CPU) to perform the computation and random access memory (RAM) to hold the inputs and outputs of the computation. One level down from the algorithmic level is the *register transfer level* (*RTL*). At this level, the details of how data is moved between and within subsystems 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.

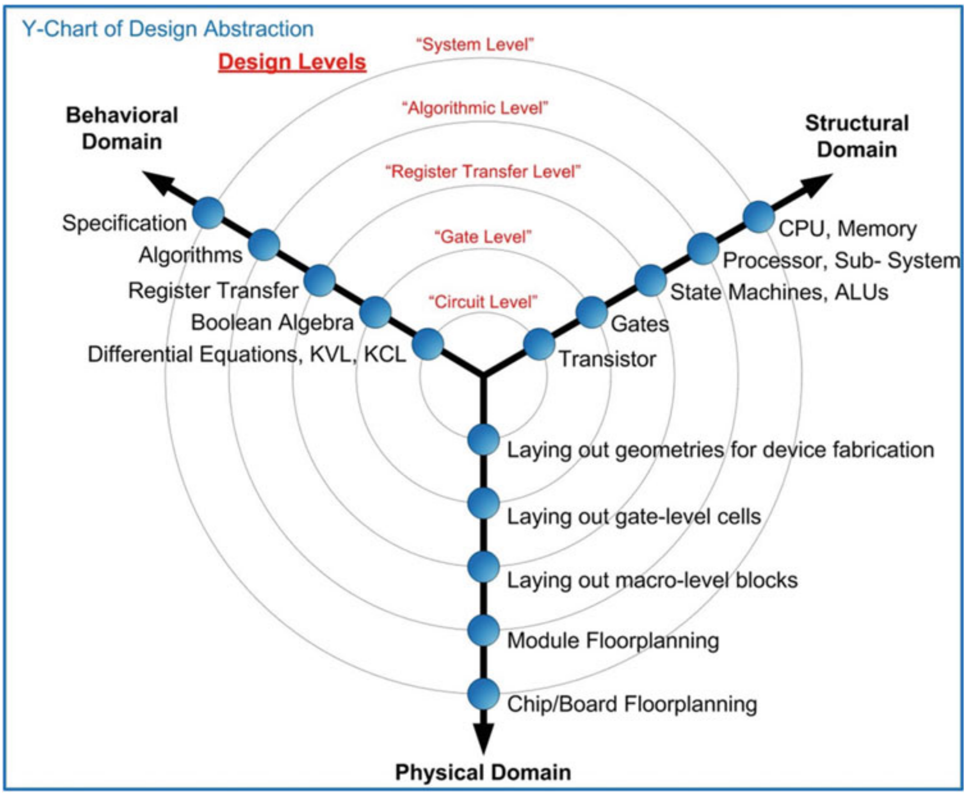


Fig. 5.3
Y-Chart of design abstraction

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.

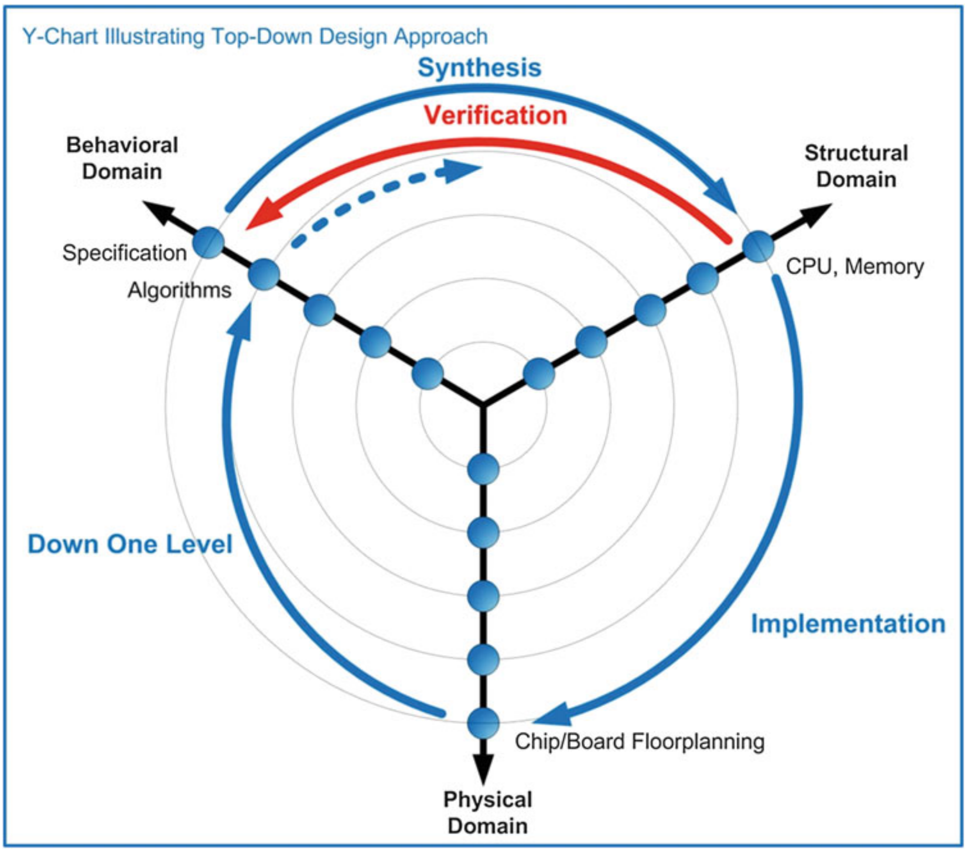


Fig. 5.4
Y-Chart illustrating top-down design approach

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 subsystems, 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.

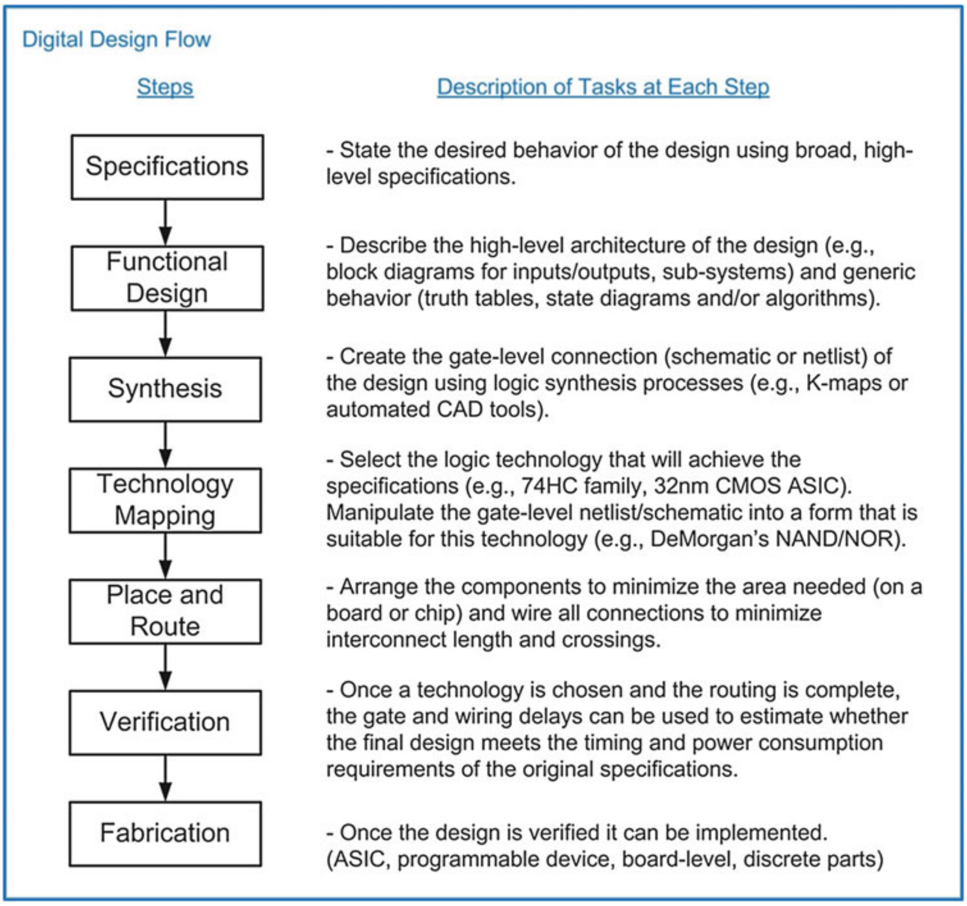
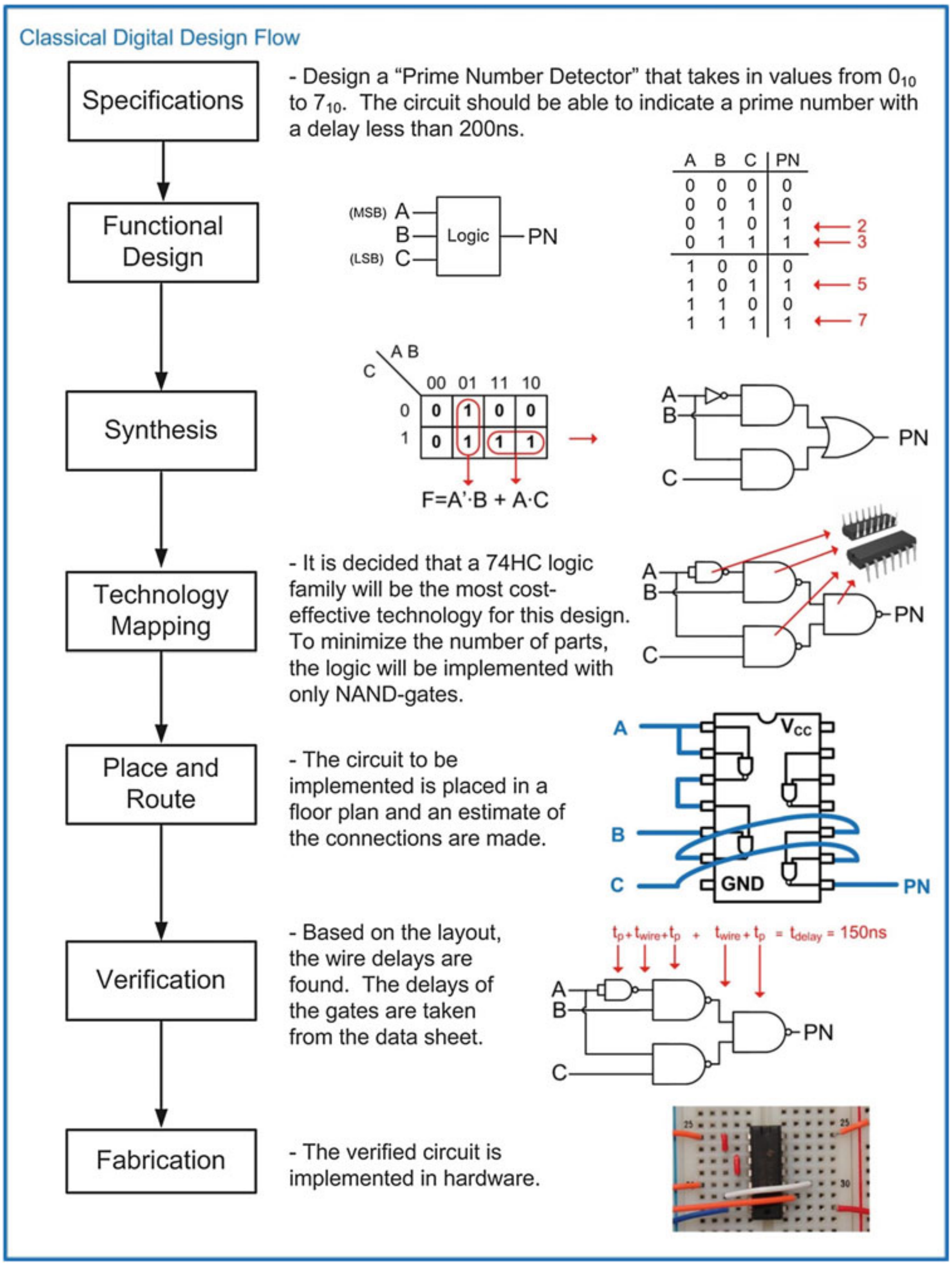

Digital Design Flow
Steps
Description of Tasks at Each Step
Specifications
- State the desired behavior of the design using broad, high-level specifications.
Functional Design
- Describe the high-level architecture of the design (e.g., block diagrams for inputs/outputs, sub-systems) and generic behavior (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.



- Design a "Prime Number Detector" that takes in values from 0_{10} to 7_{10}. The circuit should be able to indicate a prime number with a delay less than 200ns.

A	B	C	PN
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

C \ AB	00	01	11	10
0	0	1	0	0
1	0	1	1	1

- 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.

- The circuit to be implemented is placed in a floor plan and an estimate of the connections are made.

- Based on the layout, the wire delays are found. The delays of the gates are taken from the data sheet.

- 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.

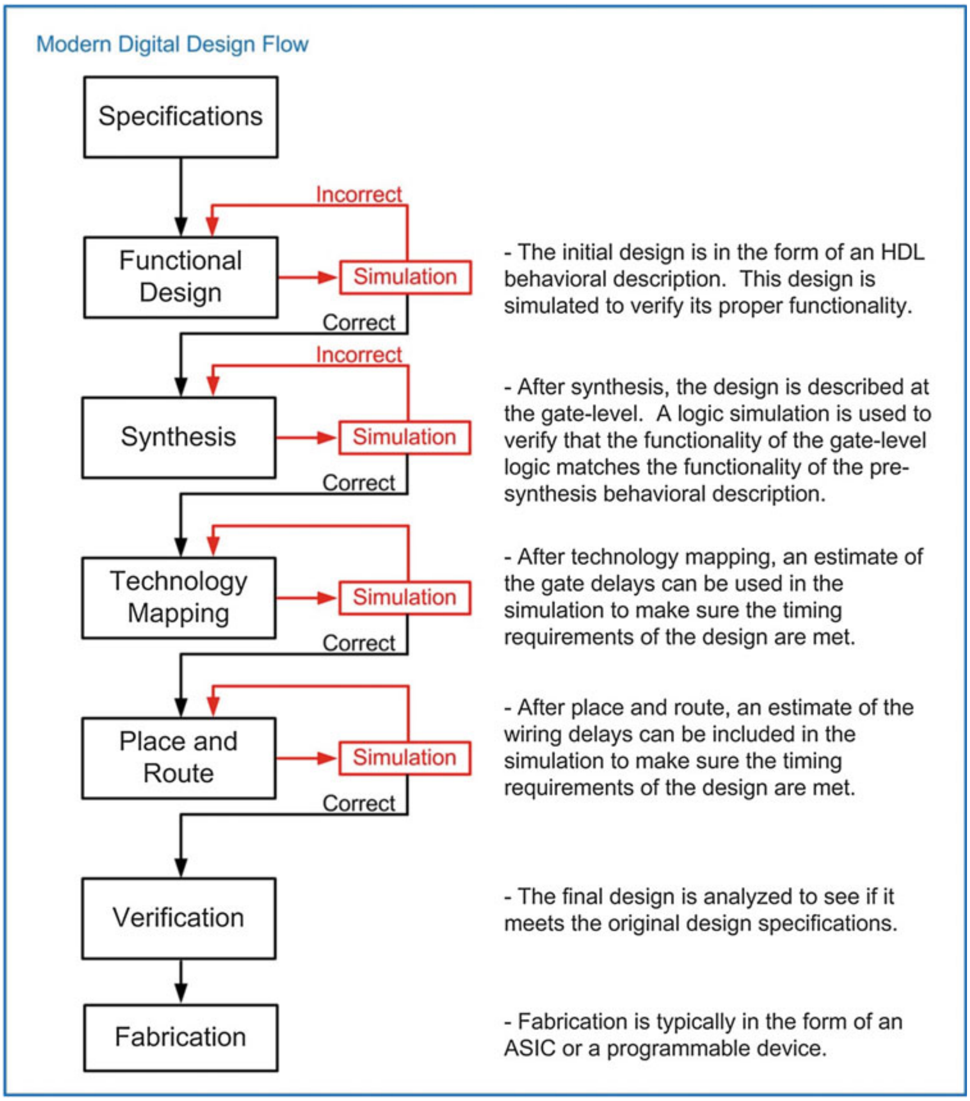

Modern Digital Design Flow
Specifications
Incorrect
Functional Design
Simulation
Correct
- The initial design is in the form of an HDL behavioral description. This design is simulated to verify its proper functionality.
Incorrect
Synthesis
Simulation
Correct
- After synthesis, the design is described 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.
Technology Mapping
Simulation
Correct
- 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.
Place and Route
Simulation
Correct
- 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.
Verification
- The final design is analyzed to see if it meets the original design specifications.
Fabrication
- 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 VHDL Constructs

Now we begin looking at the details of VHDL. A VHDL design describes a single system in a single file. The file has the suffix *.vhd. Within the file, there are two parts that describe the system: the **entity** and the **architecture**. The entity describes the interface to the system (i.e., the inputs and outputs) and the architecture describes the behavior. The functionality of VHDL (e.g., operators, signal types, functions) is defined in the **package**. Packages are grouped within a **library**. IEEE defines the base set of functionality for VHDL in the *standard* package. This package is contained within a library called *IEEE*. The library and package inclusion is stated at the beginning of a VHDL file before the entity and architecture. Additional functionality can be added to VHDL by including other packages, but all packages are based on the core functionality defined in the standard package. As a result, it is not necessary to explicitly state that a design is using the IEEE standard package because it is inherent in the use of VHDL. All functionality described in this chapter is for the IEEE standard package while other common packages are covered in Chap. 8. Figure 5.8 shows a graphical depiction of a VHDL file.

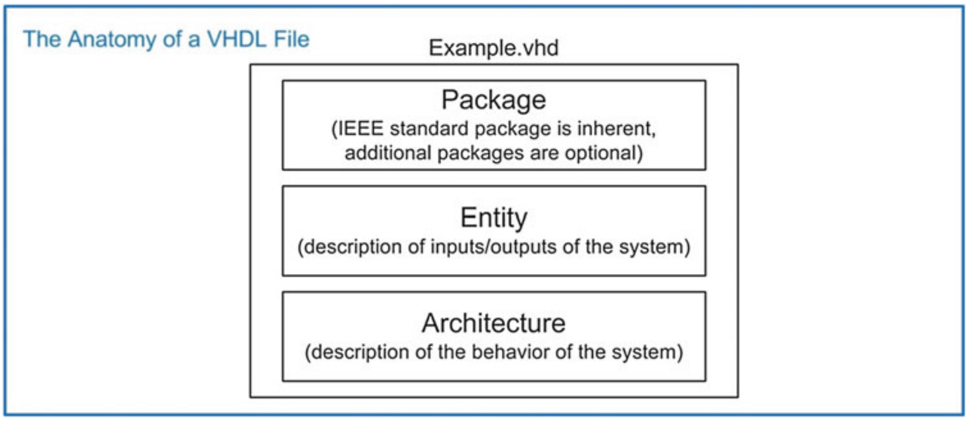

The Anatomy of a VHDL File
Example.vhd
Package
(IEEE standard package is inherent, additional packages are optional)
Entity
(description of inputs/outputs of the system)
Architecture
(description of the behavior of the system)

Fig. 5.8
The anatomy of a VHDL file

VHDL is not case sensitive. Also, each VHDL 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 the VHDL more readable. Comments in VHDL are preceded with two dashes (i.e., --) and continue until the end of the line. All user-defined names in VHDL must start with an alphabetic letter, not a number. User-defined names are not allowed to be the same as any VHDL keyword. This chapter contains many definitions of syntax in VHDL. The following notations will be used throughout the chapter when introducing new constructs:

bold	= VHDL keyword, use as is
italics	= User-defined name
<>	= A required characteristic such as a data type and input/output

5.4.1 Data Types

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

5.4.1.1 Enumerated Types

An *enumerated type* is one in which the exact values that the type can take on are defined.

Type	Values that the type can take on
bit	{0, 1}
boolean	{false, true}
character	{"any of the 256 ASCII characters defined in ISO 8859-1"}

The type bit is synthesizable while Boolean and character are not. The individual scalar values are indicated by putting them inside single quotes (e.g., '0,' 'a,' 'true').

5.4.1.2 Range Types

A *range type* is one that can take on any value within a range.

Type	Values that the type can take on
integer	Whole numbers between −2,147,483,648 to +2,147,483,647
real	Fractional numbers between $-1.7e^{38}$ to $+1.7e^{38}$

The integer type is a 32-bit, signed, two's complement number and is synthesizable. If the full range of integer values is not desired, this type can be bounded by including *range* <*min*> *to* <*max*>. The real type is a 32-bit, floating point value and is not directly synthesizable unless an additional package is included that defines the floating point format. The values of these types are indicated by simply using the number without quotes (e.g., 33, 3.14).

5.4.1.3 Physical Types

A *physical type* is one that contains both a value and units. In VHDL, *time* is the primary supported physical type.

Type	Values that the type can take on
time	Whole numbers between –2,147,483,648 to +2,147,483,647

(unit relationships)	**fs**	(femtosecond, 10^{-15}), base unit
	ps = 1000 fs	(picosecond, 10^{-12})
	ns = 1000 ps	(nanosecond, 10^{-9})
	μs = 1000 ns	(microsecond, 10^{-6})
	ms = 1000 μs	(millisecond, 10^{-3})
	s = 1000 ms	(second)
	min = 60 s	(minute)
	h = 60 min	(hour)

The base unit for time is fs, meaning that if no units are provided, the value is assumed to be in femtoseconds. The value of time is held as a 32-bit, signed number and is not synthesizable.

5.4.1.4 Vector Types

A *vector type* is one that consists of a linear array of scalar types.

Type	Construction
bit_vector	A linear array of type bit
string	A linear array of type character

The size of a vector type is defined by including the maximum index, the keyword **downto**, and the minimum index. For example, if a signal called *BUS_A* was given the type bit_vector(7 downto 0), it would create a vector of 8 scalars, each of type bit. The leftmost scalar would have an index of 7 and the rightmost scalar would have an index of 0. Each of the individual scalars within the vector can be accessed by providing the index number in parentheses. For example, BUS_A(0) would access the scalar in the rightmost position. The indices do not always need to have a minimum value of 0, but this is the most common indexing approach in logic design. The type bit_vector is synthesizable while string is not. The values of these types are indicated by enclosing them inside double quotes (e.g., "0011," "abcd").

5.4.1.5 User-Defined Enumerated Types

A *user-defined enumerated type* is one in which the name of the type is specified by the user in addition to all of the possible values that the type can assume. The creation of a user-defined enumerated type is shown below:

```
type name is (value1, value2, ...);
```

Example:

```
type traffic_light is (red, yellow, green);
```

In this example, a new type is created called *traffic_light.* If we declared a new signal called Sig1 and assigned it the type traffic_light, the signal could only take on values of red, yellow, and green. User-defined enumerated types are synthesizable in specific applications.

5.4.1.6 Array Type

An *array* contains multiple elements of the same type. Elements within an array can be scalar or vectors. In order to use an array, a new type must be declared that defines the configuration of the array. Once the new type is created, signals may be declared of that type. The *range* of the array must be defined in the array type declaration. The range is specified with integers (min and max) and either the keywords *downto* or *to*. The creation of an array type is shown below:

```
type name is array (<range>) of <element_type>;
```

Example:

```
type block_8x16 is array (0 to 7) bit_vector(15 downto 0);
signal my_array : block_8x16;
```

In this example, the new array type is declared with eight elements. The beginning index of the array is 0 and the ending index is 7. Each element in the array is a 16-bit vector of type bit_vector.

5.4.1.7 Subtypes

A *subtype* is a constrained version, or subset of another type. Subtypes are user defined, although a few commonly used subtypes are predefined in the standard package. The following is the syntax for declaring a subtype and two examples of commonly used subtypes (NATURAL and POSTIVE) that are defined in the standard package:

```
subtype name is <type> range <min> to <max>;
```

Example:

```
subtype NATURAL is integer range 0 to 255;
subtype POSTIVE is integer range 1 to 256;
```

5.4.2 Libraries and Packages

As mentioned earlier, the IEEE standard package is implied when using VHDL; however, we can use it as an example of how to include packages in VHDL. The keyword **library** is used to signify that packages are going to be added to the VHDL design from the specified library. The name of the library follows this keyword. To include a specific package from the library, a new line is used with the keyword **use** followed by the package details. The package syntax has three fields separated with a period. The first field is the library name. The second field is the package name. The third field is the specific functionality of the package to be included. If all functionality of a package is to be used, then the keyword **all** is used in the third field. Examples of how to include some of the commonly used packages from the IEEE library are shown below:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_textio.all;
```

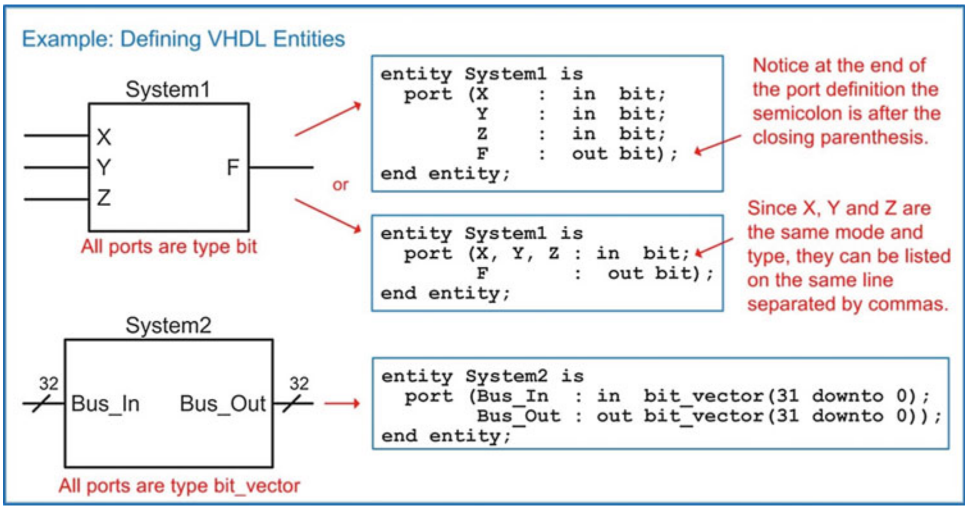
5.4.3 The Entity

The entity in VHDL describes the inputs and outputs of the system. These are called **ports**. Each port needs to have its name, mode, and type specified. The name is user defined. The mode describes the direction data is transferred through the port and can take on values of **in**, **out**, **inout**, and **buffer**.

The type is one of the legal data types described above. Port names with the same mode and type can be listed on the same line separated by commas. The definition of an entity is given below:

```
entity entity_name is
  port (port_name  : <mode> <type>;
      port_name  : <mode> <type>);
end entity;
```

Example 5.1 shows multiple approaches for defining an entity.


Example: Defining VHDL Entities
System1
X
Y
Z
F
All ports are type bit
or
Notice at the end of the port definition the semicolon is after the closing parenthesis.
Since X, Y and Z are the same mode and type, they can be listed on the same line separated by commas.
System2
32
Bus_In
Bus_Out
32
All ports are type bit_vector

```
entity System1 is
  port (X    :  in  bit;
        Y    :  in  bit;
        Z    :  in  bit;
        F    :  out bit);
end entity;
```

```
entity System1 is
  port (X, Y, Z : in  bit;
        F       :  out bit);
end entity;
```

```
entity System2 is
  port (Bus_In  : in  bit_vector(31 downto 0);
        Bus_Out : out bit_vector(31 downto 0));
end entity;
```

Example 5.1
Defining VHDL Entities

5.4.4 The Architecture

The architecture in VHDL describes the behavior of a system. There are numerous techniques to describe behavior in VHDL that span multiple levels of abstraction. The architecture is where the majority of the design work is conducted. The form of a generic architecture is given below:

```
architecture architecture_name of <entity associated with> is

 -- user-defined enumerated type declarations   (optional)
 -- signal declarations                         (optional)
 -- constant declarations                       (optional)
 -- component declarations                      (optional)

 begin

 -- behavioral description of the system goes here

end architecture;
```

5.4.4.1 Signal Declarations

A signal that is used for internal connections within a system is declared in the architecture. Each signal must be declared with a type. The signal can only be used to make connections of like types. A signal is declared with the keyword **signal** followed by a user-defined name, colon, and the type.

Signals 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. Signals represent wires within the system, so they do not have a direction or mode. Signals cannot have the same name as a port in the system in which they reside. The syntax for a signal declaration is as follows:

```
signal name : <type>;
```

Example:

```
signal node1  : bit;
signal a1, b1 : integer;
signal Bus3   : bit_vector (15 downto 0);
signal C_int  : integer range 0 to 255;
```

VHDL supports a hierarchical design approach. Signal names can be the same within a subsystem as those at a higher level without conflict. Figure 5.9 shows an example of legal signal naming in a hierarchical design.

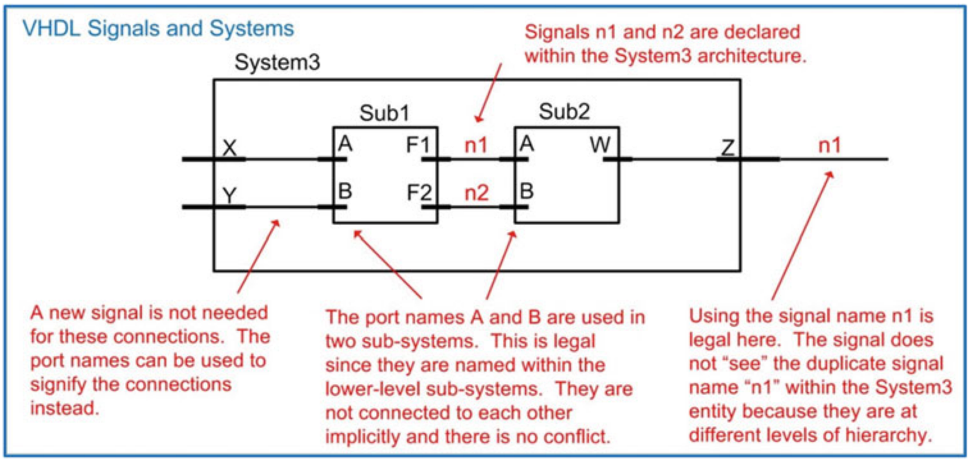

VHDL Signals and Systems
System3
Signals n1 and n2 are declared within the System3 architecture.
Sub1
Sub2
X
Y
A
B
F1
F2
n1
n2
A
B
W
Z
n1
A new signal is not needed for these connections. The port names can be used to signify the connections instead.
The port names A and B are used in two sub-systems. 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 entity because they are at different levels of hierarchy.

Fig. 5.9
VHDL signals and systems

5.4.4.2 Constant Declarations

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

```
constant constant_name : <type> := <value>;
```

Example:

```
constant BUS_WIDTH : integer := 32;
```

Once declared, the constant name can now be used throughout the architecture. 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 downto 0).

Example:

```
signal BUS_A : bit_vector (BUS_WIDTH-1 downto 0);
```

5.4.4.3 Component Declarations

A **component** is the term used for a VHDL subsystem that is instantiated within a higher level system. If a component is going to be used within a system, it must be declared in the architecture before the begin statement. The syntax for a component declaration is as follows:

```
component component_name
  port (port_name  : <mode> <type>;
     port_name  : <mode> <type>);
end component;
```

The port definitions of the component must match the port definitions of the subsystem's entity exactly. As such, these lines are typically copied directly from the lower level systems VHDL entity description. Once declared, a component can be instantiated after the begin statement in the architecture as many times as needed.

CONCEPT CHECK

CC5.4(a) Why don't we need to explicitly include the STANDARD package when creating a VHDL design?

A) It defines the base functionality of VHDL so its use is implied.

B) The simulator will automatically add it to the .vhd file upon compile.

C) It isn't recognized by synthesizers so it shouldn't be included.

D) It is a historical artifact that that isn't used anymore.

CC5.4(b) What is the difference between types Boolean {TRUE, FALSE} and bit {0, 1}?

A) They are the same.

B) Boolean is used for decision making constructs (when, else) while bit is used to model real digital signals.

C) Logical operators work with type Boolean but not for type bit.

D) Only type bit is synthesizable.

5.5 Modeling Concurrent Functionality in VHDL

It is important to remember that VHDL 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 VHDL, the lines of code represent the behavior of real hardware. As a result, all signal assignments are by default executed concurrently unless specifically noted otherwise. All operations in VHDL must be on like types and the result must be assigned to the same type as the operation inputs.

5.5.1 VHDL Operators

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

5.5.1.1 Assignment Operator

VHDL uses <= for all signal assignments and := for all variable and initialization assignments. These assignment operators work on all data types. The target of the assignment goes on the left of these operators and the input arguments go on the right.

Example:

```
F1 <= A;          -- F1 and A must be the same size and type
F2 <= '0';        -- F2 is type bit in this example
F3 <= "0000";     -- F3 is type bit_vector(3 downto 0) in this example
F4 <= "hello";    -- F4 is type string in this example
F5 <= 3.14;       -- F5 is type real in this example
F6 <= x"1A";      -- F6 is type bit_vector(7 downto 0), x"1A" is in HEX
```

5.5.1.2 Logical Operators

VHDL contains the following logical operators:

Operator	Operation
not	Logical negation
and	Logical AND
nand	Logical NAND
or	Logical OR
nor	Logical NOR
xor	Logical exclusive-OR
xnor	Logical exclusive-NOR

These operators work on types bit, bit_vector, and boolean. For operations on the type bit_vector, the input vectors must be the same size and will take place in a bit-wise fashion. For example, if two 8-bit buses called BusA and BusB were AND'd together, BusA(0) would be individually AND'd with BusB(0), BusA(1) would be individually AND'd with BusB(1), etc. The not operator is a unary operation (i.e., it operates on a single input), and the keyword is put before the signal being operated on. All other operators have two or more inputs and are placed in between the input names.

Example:

```
F1 <= not A;
F2 <= B and C;
```

The order of precedence in VHDL is different from that in Boolean algebra. The NOT operator is a higher priority than all other operators. All other logical operators have the same priority and have no inherent precedence. This means that in VHDL, the AND operator will *not* precede the OR operation as it does in Boolean algebra. Parentheses are used to explicitly describe precedence. If operators are used that have the same priority and parentheses are not provided, then the operations will take place on the signals listed first moving left to right in the signal assignment. The following are examples on how to use these operators.

Example:

```
F3 <= not D nand E;        -- D will be complemented first, the result
                           -- will then be NAND'd with E, then the
                           -- result will be assigned to F3
F4 <= not (F or G);        -- the parentheses take precedence so
                           -- F will be OR'd with G first, then
                       -- complemented, and then assigned to F4.

F5 <= H nor I nor J;   -- logic operations can have any number of
                               -- inputs.

F6 <= K xor L xnor M;     -- XOR and XNOR have the same priority so with
                                -- no parentheses given, the logic operations
                                -- will take place on the signals from
                                -- left to right. K will be XOR'd with L first,
                                -- then the result will be XNOR'd with M.
```

5.5.1.3 Numerical Operators

VHDL contains the following numerical operators:

Operator	Operation
+	Addition
-	Subtraction
*****	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute value
******	Exponential

These operators work on types integer and real. Note that the default VHDL standard does not support numerical operators on types bit and bit_vector.

5.5.1.4 Relational Operators

VHDL contains the following relational operators. These operators compare two inputs of the same type and returns the type Boolean (i.e., true or false).

Operator	Returns true if the comparison is:
=	Equal
/=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

5.5.1.5 Shift Operators

VHDL contains the following shift operators. These operators work on vector types bit_vector and string.

Operator	Operation
sll	Shift left logical
srl	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left
ror	Rotate right

The syntax for using a shift operation is to provide the name of the vector followed by the desired shift operator, followed by an integer indicating how many shift operations to perform. The target of the assignment must be of the same type and size as the input.

Example:

```
A <= B srl 3;       -- A is assigned the result of a logical shift
                    -- right 3 times on B.
```

5.5.1.6 Concatenation Operator

In VHDL the & is 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 <= "11" & "00";  -- Bus1 must be 4-bits and will be assigned
        -- the value "1100"

Bus2 <= BusA & BusB;  -- If BusA and BusB are 4-bits, then Bus2
                         -- must be 8-bits.

Bus3 <= '0' & BusA;   -- This attaches a leading '0' to BusA.  Bus3
                         -- must be 5-bits
```

5.5.2 Concurrent Signal Assignments

Concurrent signal assignments are accomplished by simply using the <= operator after the begin statement in the architecture. Each individual assignment will be executed concurrently and synthesized as separate logic circuits. Consider the following example.

Example:

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

When simulated, these three lines of VHDL 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 VHDL this functionality is identical to three separate wires. This description will be directly synthesized into three separate wires.

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

Example:

```
A <= B;
B <= C;
```

In a VHDL simulation, the signal assignments of C to B and B to A will take place at the same time; however, 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.

5.5.3 Concurrent Signal Assignments with Logical Operators

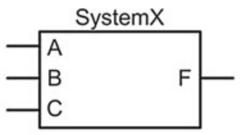
Each of the logical operators described in Sect. 5.5.1.2 can be used in conjunction with concurrent signal assignments to create individual combinational logic circuits. Example 5.2 shows how to design a VHDL model of a combinational logic circuit using this approach.

Example: Modeling Logic using Concurrent Signal Assignments and Logical Operators

Implement the following truth table using <u>concurrent signal assignments with logical operators</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

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

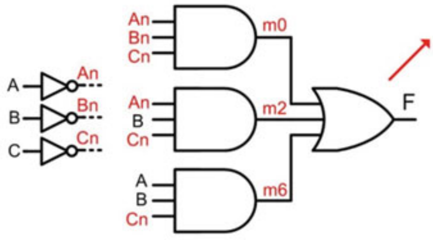


```
entity SystemX is
   port (A, B, C   :  in  bit;
            F      :  out bit);
end entity;
```

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

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

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 signals to hold the inverted versions of the inputs. Let's call them An, Bn and Cn. We'll also need three signal to hold the outputs of the AND gates. Let's call them m0, m2 and m6. Using these internal signals, the port names from the entity, and logical operators, we can describe the behavior of the logic expression above.



```
architecture SystemX_arch of SystemX is

  signal  An, Bn, Cn : bit;
  signal  m0, m2, m6 : bit;

begin

  An <= not A;              -- NOT's
  Bn <= not B;
  Cn <= not C;

  m0 <= An and Bn and Cn; -- AND's
  m2 <= An and B  and Cn;
  m6 <= A  and B  and Cn;

  F  <= m0 or m2 or m6;    -- OR

end architecture;
```

Example 5.2
Modeling Logic Using Concurrent Signal Assignments and Logical Operators

5.5.4 Conditional Signal Assignments

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 VHDL. The true power of an HDL is when the behavior of the system can be described fully without requiring any hand design. A conditional signal assignment allows us to describe a concurrent signal assignment using Boolean conditions that effect the values of the result. In a conditional signal assignment, the keyword **when** is used to describe the signal assignment for a particular Boolean condition. The keyword **else** is used to describe the signal assignments for any other conditions. Multiple Boolean conditions can be used to fully describe the output of the circuit under all input conditions. Logical operators can also be used in the Boolean conditions to create more sophisticated conditions. The Boolean conditions can be encompassed within parentheses for readability. The syntax for a conditional signal assignment is shown below:

```
signal_name <= expression_1 when condition_1 else
               expression_2 when condition_2 else
                       :
               expression_n;
```

Example:

```
F1 <= '0' when A='0' else '1';
F2 <= '1' when (A='0' and B='1') else '0';
F3 <= A when (C = D) else B;
```

An important consideration of conditional signal assignments is that they are still executed concurrently. Each assignment represents a separate, combinational logic circuit. In the above example, F1, F2, and F3 will be implemented as three separate circuits. Example 5.3 shows how to design a VHDL model of a combinational logic circuit using conditional signal assignments. Note that this example uses the same truth table as in Example 5.2 to illustrate a comparison between approaches.

Example: Modeling Logic using Conditional Signal Assignments

Implement the following truth table using a conditional signal assignment.

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

```
entity SystemX is
   port (A, B, C  :  in  bit;
            F     :  out bit);
end entity;
```

We can implement the entire truth table in its current form using a conditional signal assignment. While this is a verbose approach, it is sometimes more readable.

```
architecture SystemX_arch of SystemX is

begin

  F <= '1' when (A='0' and B='0' and C='0') else
       '0' when (A='0' and B='0' and C='1') else
       '1' when (A='0' and B='1' and C='0') else
       '0' when (A='0' and B='1' and C='1') else
       '0' when (A='1' and B='0' and C='0') else
       '0' when (A='1' and B='0' and C='1') else
       '1' when (A='1' and B='1' and C='0') else
       '0' when (A='1' and B='1' and C='1');

end architecture;
```

We can also reduce this into a more compressed form by only stating the input conditions that correspond to an output of '1' and using the "else" statement to produce an output of '0' for all other input codes.

```
architecture SystemX_arch of SystemX is

begin

  F <= '1' when (A='0' and B='0' and C='0') else
       '1' when (A='0' and B='1' and C='0') else
       '1' when (A='1' and B='1' and C='0') else
       '0';

end architecture;
```

Example 5.3
Modeling Logic Using Conditional Signal Assignments

5.5.5 Selected Signal Assignments

A selected signal assignment provides another technique to implement concurrent signal assignments. In this approach, the signal assignment is based on a specific value on the input signal. The keyword **with** is used to begin the selected signal assignment. It is then followed by the name of the input that will be used to dictate the value of the output. Only a single variable name can be listed as the input. This means that if the assignment is going to be based on multiple variables, they must first be concatenated into a single vector name before starting the selected signal assignment. After the input is listed, the keyword **select** signifies the beginning of the signal assignments. An assignment is made to a signal based on a list of possible input values that follow the keyword **when**. Multiple values of the input codes can be used and are separated by commas. The keyword **others** is used to cover any input values that are not explicitly stated. The syntax for a selected signal assignment is as follows:

```
with input_name select
   signal_name <= expression_1 when condition_1,
                  expression_2 when condition_2,
                             :
                  expression_n when others;
```

Example:

```
with A select
  F1 <= '1' when '0', -- F1 will be assigned '1' when A='0'
        '0' when '1'; -- F1 will be assigned '0' when A='1'

AB <= A&B; -- concatenate A and B so that they can be used as a vector
with AB select
  F2 <= '0' when "00", -- F2 will be assigned '0' when AB="00"
        '1' when "01",
        '1' when "10",
        '0' when "11";

with AB select
  F3 <= '1' when "01",
    '1' when "10",
    '0' when others;
```

One feature of selected signal assignments that makes its form even more compact is that multiple input codes that correspond to the same output assignment can be listed on the same line pipe (|)-delimited. The example for F3 can be equivalently described as:

```
with AB select
  F3 <= '1' when "01" | "10",
        '0' when others;
```

Example 5.4 shows how to design a VHDL model of a combinational logic circuit using selected signal assignments. 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 Logic using Selected Signal Assignments

Implement the following truth table using a <u>selected signal assignment</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

```
entity SystemX is
   port (A, B, C  :  in  bit;
               F  :  out bit);
end entity;
```

We can implement the entire truth table in its current form using a selected signal assignment. Since we are basing our output values on three separate scalar inputs, we need to concatenate them into a vector so that the new vector name can be used as the input in the selected signal assignment. We'll first declare a new signal called "ABC" of type bit_vector(2 downto 0). After the begin statement, we'll assign the concatenation of A, B and C to this vector. The new vector name can now be used as an input.

```
architecture SystemX_arch of SystemX is

  signal ABC : bit_vector(2 downto 0);

begin

  ABC <= A & B & C;

  with (ABC) select
     F <= '1' when "000",
          '0' when "001",
          '1' when "010",
          '0' when "011",
          '0' when "100",
          '0' when "101",
          '1' when "110",
          '0' when "111";

end architecture;
```

We can reduce the size of the selected signal assignment by only listing the input codes corresponding to an output of '1' and use the "others" keyword to handle all input codes corresponding to an output of '0'.

```
architecture SystemX_arch of SystemX is

  signal ABC : bit_vector(2 downto 0);

begin

  ABC <= A & B & C;

  with (ABC) select
     F <= '1' when "000",
          '1' when "010",
          '1' when "110",
          '0' when others;

end architecture;
```

We can further reduce the size of the selected signal assignment by pipe delimiting the input codes corresponding to an output of '1'.

```
architecture SystemX_arch of SystemX is

  signal ABC : bit_vector(2 downto 0);

begin

  ABC <= A & B & C;

  with (ABC) select
     F <= '1' when "000"|“010”|“110”,
          '0' when others;

end architecture;
```

Example 5.4
Modeling Logic Using Selected Signal Assignments

5.5.6 Delayed Signal Assignments

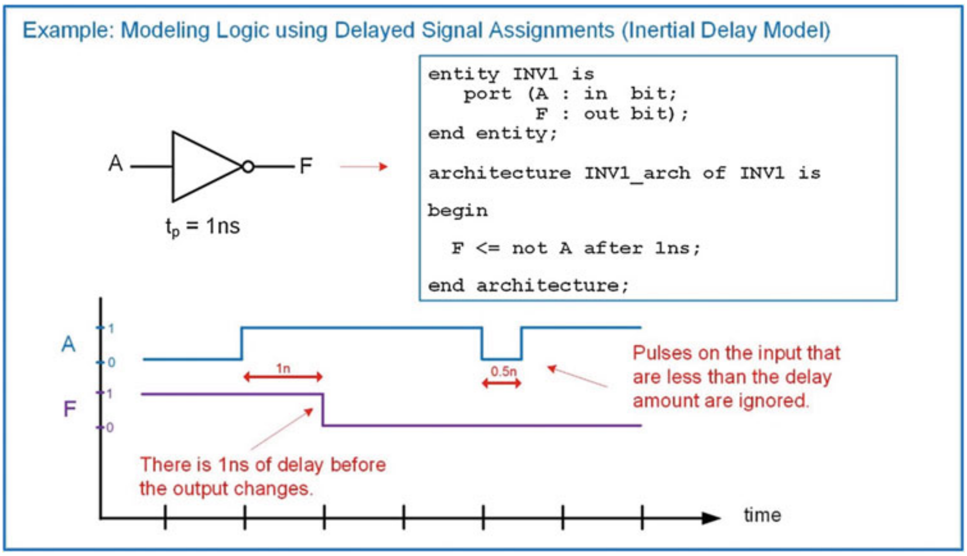
VHDL provides the ability to delay a concurrent signal assignment in order to more accurately model the behavior of real gates. The keyword **after** is used to delay an assignment by a certain amount of time. The magnitude of the delay is provided as type time. The syntax for delaying an assignment is as follows:

```
signal_name <= <expression> after <time>;
```

Example:

```
A <= B after 3us;
C <= D and E after 10ns;
```

If an input pulse is shorter in duration than the amount of the delay, the input pulse is ignored. This is called the *inertial delay model*. Example 5.5 shows how to design a VHDL model with a delayed signal assignment using the inertial delay model.


Example: Modeling Logic using Delayed Signal Assignments (Inertial Delay Model)
A
F
t_p = 1ns
A
F
1
0
1
0
1n
0.5n
Pulses on the input that are less than the delay amount are ignored.
There is 1ns of delay before the output changes.
time

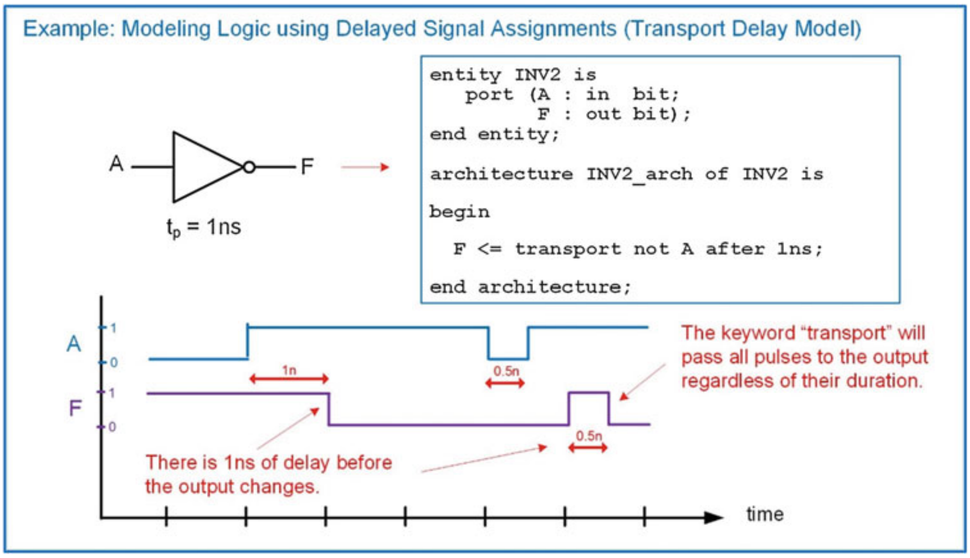
```
entity INV1 is
   port (A : in  bit;
         F : out bit);
end entity;

architecture INV1_arch of INV1 is

begin

  F <= not A after 1ns;

end architecture;
```

Example 5.5
Modeling Logic Using Delayed Signal Assignments (Inertial Delay Model)

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. If it is desired to have all pulses on the inputs show up on the outputs when modeling the behavior of other types of digital logic, the keyword **transport** is used in conjunction with the after statement. This is called the *transport delay model*:

```
signal_name <= transport <expression> after <time>;
```

Example 5.6 shows how to perform a delayed signal assignment using the transport delay model.


Example: Modeling Logic using Delayed Signal Assignments (Transport Delay Model)
A
F
t_p = 1ns
A
F
1
0
1
0
1n
0.5n
0.5n
The keyword "transport" will pass all pulses to the output regardless of their duration.
There is 1ns of delay before the output changes.
time

```
entity INV2 is
   port (A : in  bit;
         F : out bit);
end entity;

architecture INV2_arch of INV2 is

begin

  F <= transport not A after 1ns;

end architecture;
```

Example 5.6
Modeling Logic Using Delayed Signal Assignments (Transport Delay Model)

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 concurrent signal assignments 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 VHDL 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 Using Components

Structural design in VHDL refers to including lower level subsystems within a higher level system in order to produce the desired functionality. A purely structural VHDL design would not contain any behavioral modeling in the architecture such as signal assignments, but instead just contain the instantiation and interconnections of other subsystems. A subsystem is called a **component** in VHDL. For any component that is going to be used in an architecture, it must be declared before the begin statement. Refer to Sect. 5.4.4.3 for the syntax of declaring a component. A specific component only needs to be declared once. After the begin statement it can be used as many times as necessary. Each component is executed concurrently.

5.6.1 Component Instantiation

The term *instantiation* refers to the *use* or *inclusion* of the component in the VHDL system. When a component is instantiated, it needs to be given a unique identifying name. This is called the *instance name*. To instantiate a component, the instance name is given first, followed by a colon and then the component name. The last part of instantiating a component is connecting signals to its ports. The way in which signals are connected to the ports of the component is called the **port map**. The syntax for instantiating a component is as follows:

```
instance_name : <component name>
port map (<port connections>);
```

There are two techniques to connect signals to the ports of the component, *explicit port mapping* and *positional port mapping*.

5.6.1.1 Explicit Port Mapping

In explicit port mapping the name of each port of the component is given, followed by the connection indicator =>, followed by the signal it is connected to. 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 name is separated by a comma. The syntax for explicit port mapping is as follows:

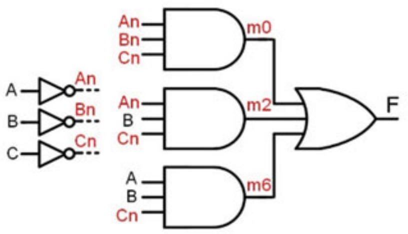
```
instance_name : <component name>
 port map (port1 => signal1, port2 => signal2, ...);
```

Example 5.7 shows how to design a VHDL model of a combinational logic circuit using structural VHDL and explicit port mapping. Note that this example again uses the same truth table as in Examples 5.2, 5.3, and 5.4 to illustrate a comparison between approaches.

Example: Modeling Logic using Structural VHDL (Explicit Port Mapping)

Implement the following truth table with structural VHDL using lower level sub-systems for the basic gates. We will assume that VHDL designs have been completed for the inverter, AND gate, and OR gate. The entities for these designs are provided.

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



```
entity INV1 is
 port (A : in  bit;
       F : out bit);
end entity;
```

```
entity AND3 is
 port (A,B,C : in  bit;
       F     : out bit);
end entity;
```

```
entity OR3 is
 port (A,B,C : in  bit;
       F     : out bit);
end entity;
```

The basic gate designs can be declared as components in our system and then instantiated in order to describe the sum of products logic diagram above.

```
entity SystemX is
   port (A, B, C  :  in  bit;
         F        :  out bit);
end entity;

architecture SystemX_arch of SystemX is

  signal  An, Bn, Cn : bit;   -- declare signals
  signal  m0, m2, m6 : bit;

  component INV1              -- declare INV1
     port (A : in  bit;
           F : out bit);
  end component;

  component AND3              -- declare AND3
     port (A,B,C : in  bit;
           F     : out bit);
  end component;

  component OR3
     port (A,B,C : in  bit;   -- declare OR3
           F     : out bit);
  end component;

begin

  U1  : INV1 port map  (A=>A, F=>An);
  U2  : INV1 port map  (A=>B, F=>Bn);
  U3  : INV1 port map  (A=>C, F=>Cn);

  U4  : AND3 port map  (A=>An, B=>Bn, C=>Cn, F=>m0);
  U5  : AND3 port map  (A=>An, B=>B,  C=>Cn, F=>m2);
  U6  : AND3 port map  (A=>A,  B=>B,  C=>Cn, F=>m6);

  U7  : OR3  port map  (A=>m0, B=>m2, C=>m6, F=>F);

end architecture;
```

The entity is named SystemX.

Internal signals are needed to connect the sub-systems.

The three lower level sub-systems are declared as components in SystemX.

The components are instantiated and connected using explicit port mapping in order to describe the behavior of the logic diagram.

NOT's

AND's

OR

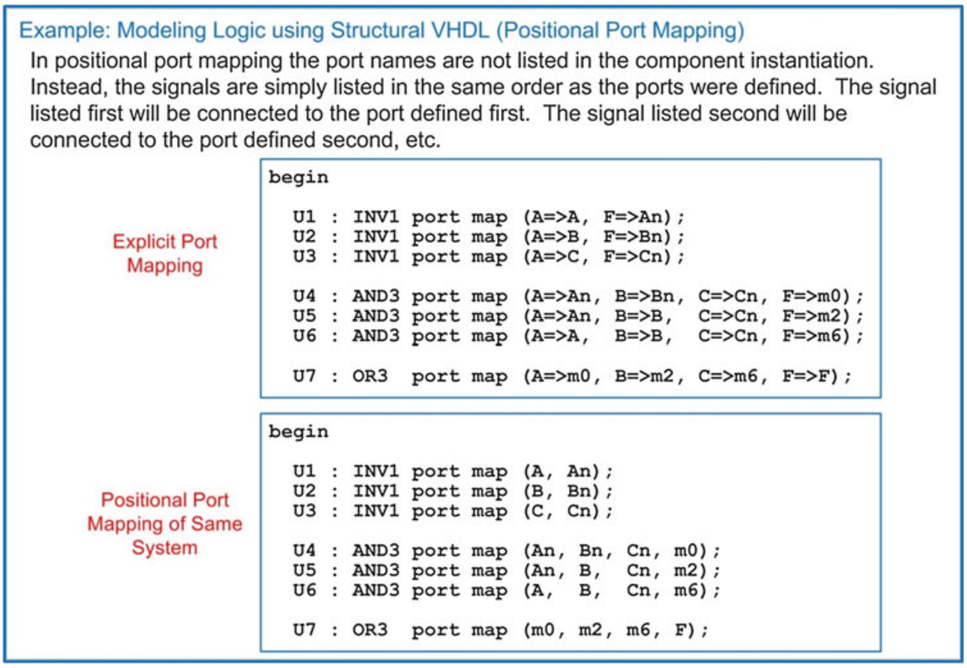
Example 5.7
Modeling Logic Using Structural VHDL (Explicit Port Mapping)

5.6.1.2 Positional Port Mapping

In positional port mapping the names of the ports of the component are not explicitly listed. Instead, the signals are listed in the same order that the ports of the component were defined. Each signal name is separated by a comma. This approach requires less text to describe but can also lead to misconnections due to mismatches in the order of the signals being connected. The syntax for positional port mapping is as follows:

```
instance_name : <component name>
port map (signal1, signal2, ...);
```

Example 5.8 shows how to create the same structural VHDL model as in Example 5.7, but using positional port mapping instead.



Example: Modeling Logic using Structural VHDL (Positional Port Mapping)

In positional port mapping the port names are not listed in the component instantiation. Instead, the signals are simply listed in the same order as the ports were defined. The signal listed first will be connected to the port defined first. The signal listed second will be connected to the port defined second, etc.

Explicit Port Mapping

```
begin

  U1 : INV1 port map (A=>A, F=>An);
  U2 : INV1 port map (A=>B, F=>Bn);
  U3 : INV1 port map (A=>C, F=>Cn);

  U4 : AND3 port map (A=>An, B=>Bn, C=>Cn, F=>m0);
  U5 : AND3 port map (A=>An, B=>B,  C=>Cn, F=>m2);
  U6 : AND3 port map (A=>A,  B=>B,  C=>Cn, F=>m6);

  U7 : OR3  port map (A=>m0, B=>m2, C=>m6, F=>F);
```

Positional Port Mapping of Same System

```
begin

  U1 : INV1 port map (A, An);
  U2 : INV1 port map (B, Bn);
  U3 : INV1 port map (C, Cn);

  U4 : AND3 port map (An, Bn, Cn, m0);
  U5 : AND3 port map (An, B,  Cn, m2);
  U6 : AND3 port map (A,  B,  Cn, m6);

  U7 : OR3  port map (m0, m2, m6, F);
```

Example 5.8
Modeling Logic Using Structural VHDL (Positional Port Mapping)

CONCEPT CHECK

CC5.6 Does the use of components model concurrent functionality? Why?

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

B) Yes. The components are treated like independent sub-systems whose behavior runs in parallel just as if separate parts were placed in a design.

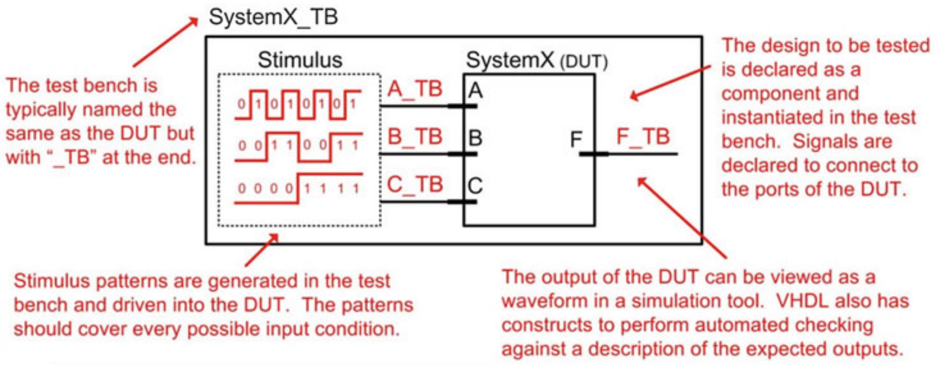
5.7 Overview of Simulation Test Benches

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 VHDL is called a **test bench**. A test bench is a file in

VHDL that has no inputs or outputs. The test bench declares the system to be tested as a component and then instantiates it. The test bench generates the input conditions and drives them into the input ports of the system being tested. VHDL 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. VHDL also has the ability to check the outputs against the expected results and notify the user if differences occur. Figure 5.10 gives an overview of how test benches are used in VHDL. The techniques to generate the stimulus patterns are covered in Chap. 8.

Overview of VHDL 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 component 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 declared as a component and 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. VHDL also has constructs to perform automated checking against a description of the expected outputs.

```
entity SystemX_TB is
end entity;

architecture SystemX_TB_arch of SystemX_TB is

  component SystemX                          -- Component Declaration
    port (A, B, C   : in  bit;
             F      : out bit);
  end component;

  signal   A_TB, B_TB, C_TB  : bit;          -- Signal Declaration
  signal   F_TB              : bit;

  begin

  DUT1  : SystemX port map (A => A_TB,       -- DUT Instantiation
                            B => B_TB,
                            C => C_TB,
                            F => F_TB);

  -- Stimulus Generation to Drive A_TB, B_TB and C_TB
  -- (covered in Chapter 8)

  -- Automated Output Checking & Reporting for F_TB
  -- (covered in Chapter 8)

end architecture;
```

Fig. 5.10
Overview of VHDL 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 VHDL 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.

❖ VHDL was originally created to document the behavior of large digital systems and support functional simulations.

❖ The ability to automatically synthesize a logic circuit from a VHDL behavioral description became possible approximately 10 years after the original definition of VHDL. As such, only a subset of the behavioral modeling techniques in VHDL 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 VHDL source file there are three main sections. These are the package, the entity, and the architecture. Including a package allows additional functionality to be included in VHDL. The entity is where the inputs and outputs of the system are declared. The architecture is where the behavior of the system is described.

❖ A *port* is an input or output to a system that is declared in the entity. A *signal* is an internal connection within the system that is declared in the architecture. A signal is not visible outside of the system.

❖ A *component* is how a VHDL system uses another subsystem. A component is first *declared*, which defines the name and entity of the subsystem to be used. The component can then be *instantiated* one or more times. 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.

❖ VHDL contains three direct techniques to model concurrent logic behavior. These are *concurrent signal assignments with logical operators*, *conditional signal assignments*, and *selected signal assignments*.

❖ VHDL components are also treated as concurrent subsystems.

❖ Delay can be modeled in VHDL using either the *initial* or *transport* model.

❖ A *simulation test bench* is a VHDL 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 VHDL?

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

5.1.3 Can all of the functionality that can be described in VHDL 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 subsystems: 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 eight 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 one 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 subsystems 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: VHDL Constructs

5.4.1 In which construct of VHDL are the inputs and outputs of the system defined?

5.4.2 In which construct of VHDL is the behavior of the system described?

5.4.3 Which construct is used to add additional functionality such as data types to VHDL?

5.4.4 What are all the possible values that the type *bit* can take on in VHDL?

5.4.5 What are all the possible values that the type *Boolean* can take on in VHDL?

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

5.4.7 What is the width of the vector defined using the type *bit_vector(63 downto 0)*?

5.4.8 What is the syntax for indexing the most significant bit in the type *bit_vector(31 downto 0)*? Assume the vector is named *example*.

5.4.9 What is the syntax for indexing the least significant bit in the type *bit_vector(31 downto 0)*? Assume the vector is named *example*.

5.4.10 What is the difference between an *enumerated* type and a *range* type?

5.4.11 What scalar type does a *bit_vector* consist of?

5.4.12 What scalar type does a *string* consist of?

Section 5.5: Modeling Concurrent Functionality in VHDL

5.5.1 Design a VHDL model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit for your ports.

$F = \sum_{A,B,C}(1,3,4,6)$

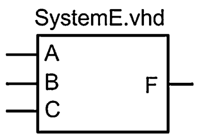


Fig. 5.11
System E Functionality

5.5.2 Design a VHDL model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.3 Design a VHDL model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.4 Design a VHDL model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit for your ports.

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

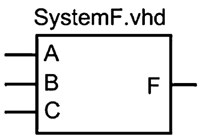


Fig. 5.12
System F Functionality

5.5.5 Design a VHDL model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.6 Design a VHDL model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.7 Design a VHDL model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit 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

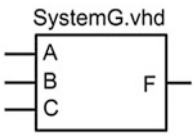


Fig. 5.13
System G Functionality

5.5.8 Design a VHDL model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.9 Design a VHDL model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.10 Design a VHDL model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit for your ports.

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

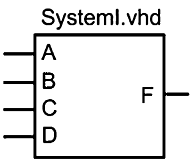


Fig. 5.14
System I Functionality

5.5.11 Design a VHDL model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.12 Design a VHDL model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.13 Design a VHDL model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.14 Design a VHDL model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

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

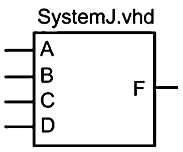


Fig. 5.15
System J Functionality

5.5.15 Design a VHDL model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.16 Design a VHDL model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. Use concurrent signal assignments and logical operators. Declare your entity to match the block diagram provided. Use the type bit 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

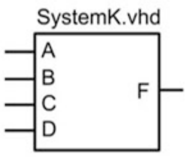


Fig. 5.16
System K Functionality

5.5.17 Design a VHDL model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. Use conditional signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.5.18 Design a VHDL model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. Use selected signal assignments. Declare your entity to match the block diagram provided. Use the type bit for your ports.

Section 5.6: Structural Design in VHDL

5.6.1 Design a VHDL model to implement the behavior described by the 3-input minterm list shown in Fig. 5.11. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.6.2 Design a VHDL model to implement the behavior described by the 3-input maxterm list shown in Fig. 5.12. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.6.3 Design a VHDL model to implement the behavior described by the 3-input truth table shown in Fig. 5.13. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.6.4 Design a VHDL model to implement the behavior described by the 4-input minterm list shown in Fig. 5.14. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.6.5 Design a VHDL model to implement the behavior described by the 4-input maxterm list shown in Fig. 5.15. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit for your ports.

5.6.6 Design a VHDL model to implement the behavior described by the 4-input truth table shown in Fig. 5.16. Use a structural design approach and basic gates. You will need to create whatever basic gates are needed for your design (e.g., INV1, AND2, OR4) and then instantiate them in your upper level architecture to create the desired functionality. The lower level gates can be implemented with concurrent signal assignments and logical operators (e.g., F <= not A). Declare your entity to match the block diagram provided. Use the type bit 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?