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

Since this book is designed to accommodate a designer that is new to VHDL, the language is presented in a manner that builds foundational knowledge first before moving into more complex topics. As such, Chaps. 1–5 only present functionality built into the VHDL standard package. Only after a comprehensive explanation of the most commonly used packages from the IEEE library is presented in Chap. 7, are examples presented that use data types from the widely adopted STD_LOGIC_1164 package. For a reader that is using the book as a reference guide, it may be more practical to pull examples from Chaps. 7–12 as they use the types std_logic and std_logic_vector. For a VHDL novice, understanding the history and fundamentals of the VHDL base release will help form a comprehensive understanding of the language; thus it is recommended that the early chapters are covered in the sequence they are written.

Bozeman, MT, USA

Brock J. LaMeres
Acknowledgments

For Alexis. The world is a better place because you are in it.
### Contents

1: THE MODERN DIGITAL DESIGN FLOW ................................................................. 1  
   1.1 HISTORY OF HARDWARE DESCRIPTION LANGUAGES .............................. 1  
   1.2 HDL ABSTRACTION ............................................................................... 4  
   1.3 THE MODERN DIGITAL DESIGN FLOW .................................................... 8  

2: VHDL CONSTRUCTS .......................................................................................... 13  
   2.1 DATA TYPES .............................................................................................. 13  
      2.1.1 Enumerated Types ............................................................................. 13  
      2.1.2 Range Types ..................................................................................... 14  
      2.1.3 Physical Types .................................................................................. 14  
      2.1.4 Vector Types .................................................................................... 14  
      2.1.5 User-Defined Enumerated Types ....................................................... 15  
      2.1.6 Array Type ........................................................................................ 15  
      2.1.7 Subtypes ........................................................................................... 15  
   2.2 VHDL MODEL CONSTRUCTION .................................................................. 16  
      2.2.1 Libraries and Packages ................................................................. 16  
      2.2.2 The Entity ......................................................................................... 17  
      2.2.3 The Architecture ............................................................................. 17  

3: MODELING CONCURRENT FUNCTIONALITY .............................................. 21  
   3.1 VHDL OPERATORS .................................................................................. 21  
      3.1.1 Assignment Operator ....................................................................... 21  
      3.1.2 Logical Operators .......................................................................... 22  
      3.1.3 Numerical Operators ...................................................................... 23  
      3.1.4 Relational Operators ...................................................................... 23  
      3.1.5 Shift Operators .............................................................................. 23  
      3.1.6 Concatenation Operator .................................................................. 24  
   3.2 CONCURRENT SIGNAL ASSIGNMENTS WITH LOGICAL OPERATORS .... 24  
      3.2.1 Logical Operator Example: SOP Circuit ........................................... 25  
      3.2.2 Logical Operator Example: One-Hot Decoder .................................. 26  
      3.2.3 Logical Operator Example: 7-Segment Display Decoder ................... 27  
      3.2.4 Logical Operator Example: One-Hot Encoder .................................. 29  
      3.2.5 Logical Operator Example: Multiplexer .......................................... 31  
      3.2.6 Logical Operator Example: Demultiplexer ..................................... 32  
   3.3 CONDITIONAL SIGNAL ASSIGNMENTS ............................................. 34  
      3.3.1 Conditional Signal Assignment Example: SOP Circuit ....................... 34  
      3.3.2 Conditional Signal Assignment Example: One-Hot Decoder ............... 35  
      3.3.3 Conditional Signal Assignment Example: 7-Segment Display Decoder ... 36  
      3.3.4 Conditional Signal Assignment Example: One-Hot Encoder ............. 37  
      3.3.5 Conditional Signal Assignment Example: Multiplexer ...................... 38  
      3.3.6 Conditional Signal Assignment Example: Demultiplexer ................. 39
Chapter 1: The Modern Digital Design Flow

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

There are two dominant hardware description languages in use today. They are VHDL and Verilog. VHDL stands for very high speed integrated circuit hardware description language. Verilog is not an acronym but rather a trade name. The use of these two HDLs is split nearly equally within the digital design industry. Once one language is learned, it is simple to learn the other language, so the choice of the HDL to learn first is somewhat arbitrary. In this text, we will use 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 the background and context of the modern digital design flow using an HDL-based approach.

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

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

1.1 History of Hardware Description Languages

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

Gordon Moore, cofounder of Intel, predicted in 1965 that the number of transistors on an integrated circuit would double every 2 years. This prediction, now known as Moore’s Law, has held true since the...
invention of the integrated circuit. As the number of transistors on an integrated circuit grew, so did the size of the design and the functionality that could be implemented. Once the first microprocessor was invented in 1971, the capability of CAD tools increased rapidly enabling larger designs to be accomplished. These larger designs, including newer microprocessors, enabled the CAD tools to become even more sophisticated and, in turn, yield even larger designs. The rapid expansion of electronic systems based on digital integrated circuits required that different manufacturers needed to produce designs that were compatible with each other. The adoption of logic family standards helped manufacturers ensure their parts would be compatible with other manufacturers at the physical layer (e.g., voltage and current); however, one challenge that was encountered by the industry was a way to document the complex behavior of larger systems. The use of schematics to document large digital designs became too cumbersome and difficult to understand by anyone besides the designer. Word descriptions of the behavior were easier to understand, but even this form of documentation became too voluminous to be effective for the size of designs that were emerging.

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

Also in 1983, the Verilog HDL was developed by Automated Integrated Design Systems as a logic simulation language. The development of Verilog took place completely independent from the VHDL project. Automated Integrated Design Systems (renamed Gateway Design Automation in 1985) was acquired by CAD tool vendor Cadence Design Systems in 1990. In response to the 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 1.1 shows a timeline of some of the major technology milestones that have occurred in the past 150 years in the field of digital logic and HDLs.

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

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

![Levels of Design Abstraction](attachment:image.png)

**Fig. 1.2**
Levels of design abstraction
1.2 HDL 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 W 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 **Gajski and Kuhn’s Y-chart**. A Y-chart depicts abstraction across three different design domains: behavioral, structural, and physical. Each of these design domains contains levels of abstraction (i.e., system, algorithm, RTL, gate, and circuit). An example Y-chart is shown in Fig. 1.3.
A Y-chart also depicts how the abstraction levels of different design domains are related to each other. A top-down design flow can be visualized in a Y-chart by spiraling inward in a clockwise direction. Moving from the behavioral domain to the structural domain is the process of synthesis. Whenever synthesis is performed, the resulting system should be compared with the prior behavioral description. This checking is called verification. The process of creating the physical circuitry corresponding to the structural description is called implementation. The spiral continues down through the levels of abstraction until the design is implemented at a level that the geometries representing circuit elements (transistors, wires, etc.) are ready to be fabricated in silicon. Figure 1.4 shows the top-down design process depicted as an inward spiral on the Y-chart.

---

**Fig. 1.3**
Y-chart of design abstraction

---

Chapter 1: The Modern Digital Design Flow
The Y-chart represents a formal approach for large digital systems. For large systems that are designed by teams of engineers, it is critical that a formal, top-down design process is followed to eliminate potentially costly design errors as the implementation is carried out at lower levels of abstraction.

**Concept Check**

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

- **(A)** Without abstraction all schematics would be drawn at the transistor-level.
- **(B)** Abstraction allows computer programs to aid in the design process.
- **(C)** Abstraction allows the details of the implementation to be hidden, while the higher-level systems are designed. Without abstraction, the details of the implementation would overwhelm the designer.
- **(D)** Abstraction allows analog circuit designers to include digital blocks in their systems.
1.3 The Modern Digital Design Flow

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

<table>
<thead>
<tr>
<th>Steps</th>
<th>Description of Tasks at Each Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications</td>
<td>- State the desired behavior of the design using broad, high-level specifications.</td>
</tr>
<tr>
<td>Functional Design</td>
<td>- 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).</td>
</tr>
<tr>
<td>Synthesis</td>
<td>- Create the gate-level connection (schematic or netlist) of the design using logic synthesis processes (e.g., K-maps or automated CAD tools).</td>
</tr>
<tr>
<td>Technology Mapping</td>
<td>- 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).</td>
</tr>
<tr>
<td>Place and Route</td>
<td>- Arrange the components to minimize the area needed (on a board or chip) and wire all connections to minimize interconnect length and crossings.</td>
</tr>
<tr>
<td>Verification</td>
<td>- 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.</td>
</tr>
<tr>
<td>Fabrication</td>
<td>- Once the design is verified it can be implemented. (ASIC, programmable device, board-level, discrete parts)</td>
</tr>
</tbody>
</table>

This generic design process flow can be used across classical and modern digital design, although modern digital design allows additional verification at each step using automated CAD tools. Figure 1.6 shows how this flow is used in the classical design approach of a combinational logic circuit.
The modern design flow based on HDLs includes the ability to simulate functionality at each step of the process. Functional simulations can be performed on the initial behavioral description of the system. At each step of the design process, the functionality is described in more detail, ultimately moving toward the fabrication step. At each level, the detailed information can be included in the simulation to verify that the functionality is still correct and that the design is still meeting the original specifications. Figure 1.7 shows the modern digital design flow with the inclusion of simulation capability at each step.
CONCEPT CHECK

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

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

(B) Schematics required sophisticated graphics hardware to display correctly.

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

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

- The modern digital design flow relies on computer-aided engineering (CAE) and computer-aided design (CAD) tools to manage the size and complexity of today’s digital designs.
- Hardware description languages (HDLs) allow the functionality of digital systems to be entered using text. VHDL and Verilog are the two most common HDLs in use today.
- 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.

Exercise Problems

Section 1.1: History of HDLs

1.1.1 What was the original purpose of VHDL?
1.1.2 Can all of the functionality that can be described in VHDL be simulated?
1.1.3 Can all of the functionality that can be described in VHDL be synthesized?

Section 1.2: HDL Abstraction

1.2.1 Give the level of design abstraction that the following statement relates to: if there is ever an error in the system, it should return to the reset state.
1.2.2 Give the level of design abstraction that the following statement relates to: once the design is implemented in a sum of products form, DeMorgan’s Theorem will be used to convert it to a NAND-gate only implementation.
1.2.3 Give the level of design abstraction that the following statement relates to: the design will be broken down into two sub-systems, one that will handle data collection and the other that will control data flow.
1.2.4 Give the level of design abstraction that the following statement relates to: the interconnect on the IC should be changed from aluminum to copper to achieve the performance needed in this design.
1.2.5 Give the level of design abstraction that the following statement relates to: the MOSFETs need to be able to drive at least eight other loads in this design.
1.2.6 Give the level of design abstraction that the following statement relates to: this system will contain 1 host computer and support up to 1000 client computers.

Section 1.3: The Modern Digital Design Flow

1.3.1 Which step in the modern digital design flow does the following statement relate to: a CAD tool will convert the behavioral model into a gate-level description of functionality.
1.3.2 Which step in the modern digital design flow does the following statement relate to: after realistic gate and wiring delays are determined, one last simulation should be performed to make sure the design meets the original timing requirements.
1.3.3 Which step in the modern digital design flow does the following statement relate to: if the memory is distributed around the perimeter of the CPU, the wiring density will be minimized.
1.3.4 Which step in the modern digital design flow does the following statement relate to: the design meets all requirements so now I’m building the hardware that will be shipped.
1.3.5 Which step in the modern digital design flow does the following statement relate to: the system will be broken down into three sub-systems with the following behaviors.

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

1.3.7 Which step in the modern digital design flow does the following statement relate to: to meet the power requirements, the gates will be implemented in the 74HC logic family.
Chapter 2: VHDL Constructs

This chapter begins looking at the basic construction of a VHDL model. This chapter begins by covering the built-in features of a VHDL model including the file structure, data types, operators, and declarations. This chapter provides a foundation of VHDL that will lead to modeling examples provided in Chap. 3. VHDL is not case sensitive. 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, input/output, etc. |

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

2.1 Describe the data types provided in the standard VHDL package.
2.2 Describe the basic construction of a VHDL model.

2.1 Data Types

In VHDL, every signal, constant, variable, and function must be assigned a *data type*. The IEEE standard package provides a variety of 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 VHDL standard package.

2.1.1 Enumerated Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Values that the type can take on</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>boolean</td>
<td>{false, true}</td>
</tr>
<tr>
<td>character</td>
<td>{‘any of the 256 ASCII characters defined in ISO 8859-1’}</td>
</tr>
</tbody>
</table>

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’).
2.1.2 Range Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Values that the type can take on</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>Whole numbers between $-2,147,483,648$ and $+2,147,483,647$</td>
</tr>
<tr>
<td>real</td>
<td>Fractional numbers between $-1.7\times10^{38}$ and $+1.7\times10^{38}$</td>
</tr>
</tbody>
</table>

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).

2.1.3 Physical Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Values that the type can take on</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>Whole numbers between $-2,147,483,648$ and $+2,147,483,647$</td>
</tr>
</tbody>
</table>

(unit relationships)

- **fs** = $10^{-15}$, base unit
- **ps** = $10^{-12}$
- **ns** = $10^{-9}$
- **μs** = $10^{-6}$
- **ms** = $10^{-3}$
- **s** = 1 second
- **min** = 60 seconds
- **h** = 60 minutes

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.

2.1.4 Vector Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_vector</td>
<td>A linear array of type bit</td>
</tr>
<tr>
<td>string</td>
<td>A linear array of type character</td>
</tr>
</tbody>
</table>

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”).

2.1.5 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.

2.1.6 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`.

2.1.7 Subtypes

A subtype is a constrained version or subset of another type. Subtypes are user-defined, although a few commonly used subtypes are pre-defined 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;
```
CONCEPT CHECK

CC2.1 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.

2.2 VHDL Model Construction

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, etc.) 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 subsequent chapters. Figure 2.1 shows a graphical depiction of a VHDL file.

![The Anatomy of a VHDL File](image)

**Fig. 2.1**
The anatomy of a VHDL file

2.2.1 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.

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

### 2.2.2 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.

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

Example 2.1 shows multiple approaches for defining an entity.

### 2.2.3 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;

2.2.3.1 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 sub-system as those at a higher level without conflict. Figure 2.2 shows an example of legal signal naming in a hierarchical design.

---

**VHDL Signals and Systems**

Fig. 2.2

VHDL signals and systems
2.2.3.2 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 down to 0).

Example:

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

2.2.3.3 Component Declarations

A component is the term used for a VHDL sub-system 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 sub-system’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**

CC2.2 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.
Summary

- Every signal and port in VHDL needs to be associated with a data type.
- A data type defines the values that can be taken on by a signal or port.
- 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 sub-system. A component is first declared, which defines the name and entity of the sub-system to be used. The component can then be instantiated one or more times.

Exercise Problems

Section 2.1: Data Types

2.1.1 What are the possible values that the type bit can take on in VHDL?
2.1.2 What are all the possible values that the type Boolean can take on in VHDL?
2.1.3 What is the range of decimal numbers that can be represented using the type integer in VHDL?
2.1.4 What is the width of the vector defined using the type bit_vector(63 downto 0)?
2.1.5 What is the syntax for indexing the most significant bit in the type bit_vector(31 downto 0)? Assume the vector is named example.
2.1.6 What is the syntax for indexing the least significant bit in the type bit_vector(31 downto 0)? Assume the vector is named example.
2.1.7 What is the difference between an enumerated type and a range type?
2.1.8 What scalar type does a bit_vector consist.

Section 2.2: VHDL Model Construction

2.2.1 In which construct of VHDL are the inputs and outputs of the system defined?
2.2.2 In which construct of VHDL is the behavior of the system described?
2.2.3 Which construct is used to add additional functionality such as data types to VHDL?