CHAPTER 3

BASIC LANGUAGE CONSTRUCTS OF VHDL

To use a programming language, we first have to learn its syntax and language constructs. In this chapter, we illustrate the basic skeleton of a VHDL program and provide an overview of the basic language constructs, including lexical elements, objects, data types and operators. VHDL is a *strongly typed language* and imposes rigorous restriction on data types and operators. We discuss this aspect in more detail.

3.1 INTRODUCTION

VHDL is a complex language. It is designed to describe both the structural and behavioral views of a digital system at various levels of abstraction. Many of the language constructs are intended for modeling and for abstract, behavioral description. Only a small portion of VHDL can be synthesized and realized physically in hardware. The IEEE 1076.6 RTL synthesis standard tries to define a subset that can be accepted by most synthesis tools. The focus of this book is synthesis, and thus the discussion is limited primarily to this subset.

VHDL was revised twice by IEEE and there are three versions: VHDL-87, VHDL-93 and VHDL-2001. Since only simple, primitive language constructs can be synthesized, the revisions do not have a significant impact on synthesis except for some differences in the syntactical appearances. Since IEEE 1076.6 mainly follows the syntax of VHDL-87, we use the syntax of VHDL-87 in the book in general and highlight the difference if any VHDL-93 feature is used.

This chapter discusses only the basic, most commonly used language constructs in VHDL and some extensions defined in IEEE standards 1076.3 and 1164. In subsequent chapters, more specialized features are covered within the context.

3.2 SKELETON OF A BASIC VHDL PROGRAM

3.2.1 Example of a VHDL program

A VHDL program is composed of a collection of *design units*. A synthesizable VHDL program needs at least two design units: an entity declaration and an architecture body associated with the entity. The skeleton of a typical VHDL program can best be explained by an example. Let us consider the `even_detector` circuit of Chapter 2. The VHDL code is shown in Listing 3.1. It uses implicit δ-delays in signal assignment statements. Note that we use the boldface font for the VHDL's reserved words.

Listing 3.1 Even-parity detector

```
library ieee;
use ieee.std_logic_1164.all;
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;

architecture sop_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4);
   p1 <= (not a(0)) and (not a(1)) and (not a(2));
   p2 <= (not a(0)) and a(1) and a(2);
   p3 <= a(0) and (not a(1)) and a(2);
   p4 <= a(0) and a(1) and (not a(2));
end sop_arch ;
```

3.2.2 Entity declaration

The entity declaration describes the external interface, or "outline" of a circuit, including the name of the circuit and the names and basic characteristics of its input and output ports. In the example, the entity declaration indicates that the name of the circuit is `even_detector` and the circuit has a 3-bit input port, `a`, and a 1-bit output port, `even`.

The simplified syntax of an entity declaration is

```
entity entity_name is
   port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type
   );
end entity_name;
```

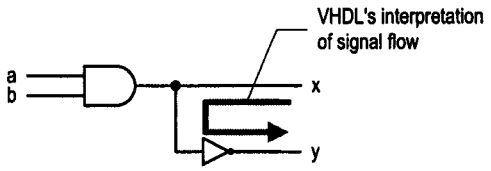

Figure 3.1 Demonstration circuit for mode.

Note that there is no semicolon (;) in the last port declaration.

A *port declaration* is composed of the `port_names`, `mode` and `data_type` terms. The `port_names` and `data_type` terms are self-explanatory. The `mode` term indicates the direction of the signal, which can be **in**, **out** or **inout**. The **in** and **out** keywords indicate that the signal flows "into" and "out of" the circuit respectively. They represent the fact that the corresponding port is an input or an output of the circuit. The **inout** keyword indicates that the signal flows in both directions and that the corresponding port is a bidirectional port. The `mode` term can also be **buffer**. It can cause a subtle compatibility problem and is not used in the book.

In the example, the port declaration shows that there are two ports. The `a` port is an input signal and its data type is `std_logic_vector(2 downto 0)`, which represents a 3-bit bus, and the `even` port is an output port and its data type is `std_logic`.

Note that a port with the **out** mode cannot be used as an input signal. For example, consider the simple circuit shown in Figure 3.1. We may be tempted to use the following code to describe the circuit:

```
library ieee;
use ieee.std_logic_1164.all;
entity mode_demo is
   port(
      a, b: in std_logic;
      x, y: out std_logic
   );
end mode_demo;
architecture wrong_arch of mode_demo is
begin
   x <= a and b;
   y <= not x;
end wrong_arch;
```

Since the `x` signal is used to obtain the `y` signal, VHDL considers it as an external signal that "flows into" the circuit, as shown in Figure 3.1. This violates the **out** mode and leads to a syntax error. One way to fix the problem is to change the mode of the `x` port to the **inout** mode. It is a poor solution since the `x` port is not actually a bidirectional port. A better alternative is to use an internal signal to represent the intermediate result, as shown in the revised code:

```
architecture ok_arch of mode_demo is
   signal ab: std_logic;
begin
   ab <= a and b;
   x <= ab;
   y <= not ab;
end ok_arch;
```

3.2.3 Architecture body

The architecture body specifies the internal operation or organization of a circuit. In VHDL, we can develop multiple architecture bodies for the same entity declaration and later choose one body to bind with the entity for simulation or synthesis. The simplified syntax of an architecture body is

```
architecture arch_name of entity_name is
   declarations;
begin
   concurrent statement;
   concurrent statement;
   concurrent statement;
   . . .
end arch_name;
```

The first line of the architecture body shows the name of the body and the corresponding entity. An architecture body may include an optional declarative section, which consists of the declarations of some objects, such as signals and constants, which are used in the architecture description. The example includes a declaration of internal signals:

```
signal p1, p2, p3, p4: std_logic;
```

The main part of the architecture body consists of the concurrent statements that describe the operation or organization of the circuit. As we discussed in Chapter 2, each concurrent statement describes an individual part and the architecture can be thought of as a collection of interconnected circuit parts. There are a variety of concurrent statements, which are discussed in subsequent chapters.

3.2.4 Design unit and library

Design units are the fundamental building blocks in a VHDL program. When a program is processed, it is broken into individual design units and each unit is analyzed and stored independently. There are five kinds of design units:

- Entity declaration
- Architecture body
- Package declaration
- Package body
- Configuration

We have just studied the entity declaration and architecture body. A *package* of VHDL normally contains a collection of commonly used items, such as data types, subprograms and components, which are needed by many VHDL programs. As the name suggests, a *package declaration* consists of the declaration of these items. A *package body* normally contains the implementation and code of the subprograms.

In VHDL, multiple architecture bodies can be associated with an entity declaration. A *configuration* specifies which architecture body is to be bound with the entity declaration. The package and configuration are discussed in Chapter 13.

A VHDL *library* is a place to store the design units. It is normally mapped into a directory in the computer's hard disk storage. The software defines mapping between the symbolic VHDL library name and the physical directory. By VHDL default, the design units will be stored in a library named `work`.

To facilitate the synthesis, IEEE has developed several VHDL packages, including the `std_logic_1164` package and the `numeric_std` package, which are defined in IEEE standards 1164 and 1076.3. These packages are discussed in Sections 3.5.2 and 3.5.4. To use a predefined package, we must include the *library* and *use statements* before the entity declaration. The first two lines of the example are for this purpose:

```
library ieee;
use ieee.std_logic_1164.all;
```

The first line invokes a library named `ieee`, and the second line makes the `std_logic_1164` package visible to the subsequent design unit. We must invoke this library because we want to use some predefined data types, `std_logic` and `std_logic_vector`, of the `std_logic_1164` package.

3.2.5 Processing of VHDL code

A VHDL program is normally processed in three stages:

1. Analysis
2. Elaboration
3. Execution

During the *analysis stage*, the software checks the syntax and some static semantic errors of the VHDL code. The analysis is performed on a design unit basis. If there is no error, the software translates the code of the design unit into an intermediate form and stores it in the designated library. A VHDL file can contain multiple design units, but a design unit cannot be split into two or more files.

In a complex design, the system is normally described in a hierarchical manner. The top level may include subsystems as instantiated components, as in the example in Section 2.2.2. During the *elaboration stage*, the software starts from the designated top-level entity declaration and binds its architecture body according to the configuration specification. If there are instantiated components, the software replaces each instantiated component with the corresponding architecture body description. The process may repeat recursively until all instantiated components are replaced. The elaboration process essentially selects and combines the needed architectural descriptions, and creates a single "flattened" description.

During the *execution stage*, the analyzed and elaborated description is usually fed to simulation or synthesis software. The former simulates and "runs" the description in a computer, and the latter realizes the description by physical circuits.

3.3 LEXICAL ELEMENTS AND PROGRAM FORMAT

3.3.1 Lexical elements

The lexical elements are the basic syntactical units in a VHDL program. They include comments, identifiers, reserved words, numbers, characters and strings.

Comments A comment starts with two dashes, `--`, followed by the comment text. Anything after the `--` symbol in the line will be ignored. The comment is for documentation purposes only and has no effect on the code. For example, we have added comments to the previous VHDL code:

```
--*****************************************************
-- example to show the caveat of the out mode
--*****************************************************
architecture ok_arch of mode_demo is
   signal ab: std_logic; -- ab is the internal signal
begin
   ab <= a and b;
   x <= ab;                -- ab connected to the x output
   y <= not ab;
end ok_arch;
```

For clarity, we use *italic type* for comments.

Identifiers An identifier is the name of an object in VHDL. The basic rules to form an identifier are:

- The identifier can contain only alphabetic letters, decimal digits and underscores.
- The first character must be a letter.
- The last character cannot be an underscore.
- Two successive underscores are not allowed.

For example, the following identifiers are valid:

```
A10, next_state, NextState, mem_addr_enable
```

On the other hand, the following identifiers violate one of the rules and will cause a syntax error during analysis of the program:

```
sig#3, _X10, 7segment, X10_, hi__there
```

Since VHDL is not case sensitive, the following identifiers are the same:

```
nextstate, NextState, NEXTSTATE, nEXTsTATE
```

It is good practice to be consistent with the use of case. In this book, we use capital letters for symbolic constants and use a special suffix, such as _n, to represent a special characteristics of an identifier. For example, the _n suffix is used to indicate an active-low signal. If we see a signal with a name like oe_n, we know it is an active-low signal.

It is also a good practice to use descriptive identifier for better readability. For example, consider the name for a signal that enables the memory address buffer. The mem_addr_en is good, mae is too short, and memory_address_enable is probably too cumbersome.

Reserved words Some words are reserved in VHDL to form the basic language constructs. These reserved words are:

```
abs access after alias all and architecture array assert
attribute begin block body buffer bus case component
configuration constant disconnect downto else elsif end
entity exit file for function generate generic guarded
if impure in inertial inout is label library linkage
literal loop map mod nand new next nor not null of on
open or others out package port postponed procedure
process pure range record register reject rem report
return rol ror select severity shared signal sla sll
sra srl subtype then to transport type unaffected units
until use variable wait when while with xnor xor
```

Numbers, characters and strings A *number* in VHDL can be integer, such as `0`, `1234` and `98E7`, or real, such as `0.0`, `1.23456` or `9.87E6`. It can be represented in other number bases. For example, 45 can be represented as `2#101101#` and `16#2D#` in base 2 and base 16 respectively. We can also add an underscore to enhance readability. For example, `12_3456` is the same as `123456`, and `2#0011_1010_1101#` is the same as `2#001110101101#`.

A *character* in VHDL is enclosed in single quotation marks, such as `'A'`, `'Z'` and `'3'`. Note that `1` and `'1'` are different since the former is a number and the latter is a character.

A *string* is a sequence of characters enclosed in double quotation marks, such as `"Hello"` and `"10000111"`. Again, note that that `2#10110010#` and `"10110010"` are different since the former is a number and the latter is a string. Unlike the number, we cannot arbitrarily use an underscore inside a string. The `"10110010"` and `"1011_0010"` strings are different.

3.3.2 VHDL program format

VHDL is a case-insensitive free-format language, which means that the letter case does not matter, and "white space" (space, tab and new-line characters) can be inserted freely between lexical elements. For example, the VHDL program

```
library ieee;
use ieee.std_logic_1164.all;
entity even_detector is
   port(
      a: in std_logic_vector(2 downto 0);
      even: out std_logic
   );
end even_detector;

architecture eg_arch of even_detector is
   signal p1, p2, p3, p4 : std_logic;
begin
   even <= (p1 or p2) or (p3 or p4);
   p1 <= (not a(0)) and (not a(1)) and (not a(2));
   p2 <= (not a(0)) and a(1) and a(2);
   p3 <= a(0) and (not a(1)) and a(2);
   p4 <= a(0) and a(1) and (not a(2));
end eg_arch;
```

is the same as

```
library ieee; use ieee.std_logic_1164.all;entity
even_detector is port(a: in std_logic_vector(2
downto 0);even: out std_logic);end even_detector;
architecture eg_arch of even_detector is signal p1,
p2, p3, p4: std_logic; begin even <= (p1 or p2) or
(p3 or p4); p1 <= (not a(0)) and (not a(1)) and
(not a(2)); p2 <= (not a(0)) and a(1) and a(2);
p3 <= a(0) and (not a(1)) and a(2); p4 <= a(0) and
a(1) and (not a(2)); end eg_arch;
```

This extreme example demonstrates the importance of proper formatting. Although the program format does not affect the content or efficiency of a design, it has a significant impact on human users. An adequately documented and formatted program makes the code easier to comprehend and helps us to locate potential design errors. It will save a

tremendous amount of time for future code revision and maintenance. This is perhaps the easiest way to enhance the reusability of the code. Section 3.6.2 lists the basic guidelines for code formatting and documentation.

It is a good idea to include a short "header" comment in the beginning of the file. The header should provide general information about the design and the "design environment." A representative header of the previous VHDL program is shown below.

```
--*************************************************
--
-- Author: p chu
--
-- File: even_det.vhd
--
-- Design units:
--    entity even_detector
--       function: check even # of 1's from input
--       input: a
--       output: even
--    architecture sop_arch:
--       truth-table-based sum-of-products
--       implementation
--
-- Library/package:
--    ieee.std_logic_1164: to use std_logic
--
-- Synthesis and verification:
--    Synthesis software: . . .
--    Options/script: . . .
--    Target technology: . . .
--    Testbench: even_detector_tb
--
-- Revision history
--   Version 1.0:
--   Date: 9/2005
--   Comments: Original
--
--*************************************************
```

The first two parts list the author and file name. The "Design units" part provides a brief description about the design units in the file. The description includes the input and output ports and the function of the entity, and the implementation method of the architecture body. The final "Revision history" part provides general information about the development. The "Library/package" and "Synthesis and verification" parts describe the design environment. The idea here is to provide the necessary information for users to reconstruct or duplicate the implementation. The "Library/package" part lists the packages and libraries that are referred to in the design file, and explains briefly the use of these packages. It is especially essential when a nonstandard or custom package is involved. The "Synthesis and verification" part lists the EDA software and the script or relevant options used in the synthesis, the original targeting device technology, as well as, if available, the testbench used to verify the design. Since synthesis software from different manufacturers supports different subsets of the VHDL and may interpret certain VHDL constructs differently, this information allows future users to duplicate the original implementation.

3.4 OBJECTS

An *object* in VHDL is a named item that holds the value of a specific data type. There are four kinds of objects: **signal**, **variable**, **constant** and **file**. A construct known as **alias** is somewhat like an object. We do not discuss the **file** object in this book since it cannot be synthesized.

Signals The signal is the most common object and we already used it in previous examples. A signal has to be declared in the architecture body's declaration section. The simplified syntax of signal declaration is

```
signal signal_name, signal_name, ... : data_type;
```

For example, the following line declares the a, b and c signals with the `std_logic` data type:

```
signal a, b, c: std_logic;
```

According to the VHDL definition, we can specify an optional initial value in the signal declaration. For example, we can assign an initial value of `'0'` to the previous signals:

```
signal a, b, c: std_logic := '0';
```

While this is sometimes handy for simulation purposes, it should not be used in synthesis since not many physical devices can implement the desired effect.

The simplified syntax of signal assignment is

```
signal_name <= projected_waveform;
```

We examined the concept of `projected_waveform` in Section 2.2.1 and discuss it in more detail in Chapter 4. From the synthesis point of view, a signal represents a wire or "a wire with memory" (i.e., a register or latch).

The input and output ports of the entity declaration are also considered as signals.

Variables A variable is a concept found in a traditional programming language. It can be thought of as a "symbolic memory location" where a value can be stored and modified. There is no direct mapping between a variable and a hardware part. A variable can only be declared and used in a process and is local to that process (the exception is a shared variable, which is difficult to use and is not discussed). The main application of a variable is to describe the abstract behavior of a system.

The syntax of variable declaration is similar to that of signal declaration:

```
variable variable_name, variable_name, ... : data_type
```

An optional initial value can be assigned to variables as well.

The simplified syntax of variable assignment is

```
variable_name := value_expression;
```

Note that no timing information is associated with a variable, and thus only a value, not a waveform, can be assigned to a variable. Since there is no delay, the assignment is known as an *immediate assignment* and the notion := is used. We examine variables in detail when the VHDL process is discussed in Chapter 5.

Constants A constant holds a value that cannot be changed. The syntax of constant declaration is

```
constant constant_name: data_type := value_expression;
```

The value_expression term specifies the value of the constant. A simple example is

```
constant BUS_WIDTH: integer := 32;
constant BUS_BYTES: integer := BUS_WIDTH / 8;
```

Note that we use capital letters for constants in this book.

Since an identifier name and data type convey more information than does a literal alone, the proper use of constants can greatly enhance readability of the VHDL code and make the code more descriptive. Consider the behavioral description of even_detector in Section 2.2.3:

```
architecture beh1_arch of even_detector is
   signal odd: std_logic;
begin
   . . .
   tmp := '0';
   for i in 2 downto 0 loop
      tmp := tmp xor a(i);
   end loop;
   . . .
```

The code uses a "hard literal," 2, to specify the upper boundary of the loop's range. It becomes much clearer if we replace it with a symbolic constant:

```
architecture beh1_arch of even_detector is
   signal odd: std_logic;
   constant BUS_WIDTH: integer := 3;
begin
   . . .
   tmp := '0';
   for i in (BUS_WIDTH-1) downto 0 loop
      tmp := tmp xor a(i);
   end loop;
 . . .
```

Alias Alias is not a data object. It is the alternative name for an existing object. As a constant, the purpose of an alias is to enhance code clarity and readability. One form of the signal alias is especially helpful for synthesis. Consider a machine instruction of a processor that is 16 bits wide and consists of fields with an operation code and three registers. The instruction is stored in memory as a 16-bit word. After it is read from memory, we can use an alias to identify the individual field:

```
signal word: std_logic_vector(15 downto 0);
alias op: std_logic_vector(6 downto 0) is word(15 downto 9);
alias reg1: std_logic_vector(2 downto 0) is word(8 downto 6);
alias reg2: std_logic_vector(2 downto 0) is word(5 downto 3);
alias reg3: std_logic_vector(2 downto 0) is word(2 downto 0);
```

Clearly, a name like reg1 is more descriptive than word(8 downto 6). Unfortunately, some synthesis software does not support this language construct. We can achieve this in a somewhat cumbersome way by declaring four new signals in the architecture body and assigning them with the proper portions of the word signal.

3.5 DATA TYPES AND OPERATORS

In VHDL, each object has a *data type*. A data type is defined by:

- A set of values that an object can assume.
- A set of operations that can be performed on objects of this data type.

VHDL is a known as *strongly typed language*, which means that an object can only be assigned a value of its type, and only the operations defined with the data type can be performed on the object. If a value of a different type has to be assigned to an object, the value must be converted to the proper data type by a *type conversion function* or *type casting*.

The motivation behind a strongly typed language is to catch errors in the early stage. For example, if a Boolean value is assigned to a signal of integer type or an arithmetic operation is applied to a signal of character type, the software can detect the error during the analysis stage. On the downside, the rigid type requirement may introduce many type-conversion functions and make the code cumbersome and difficult to understand.

To facilitate modeling and simulation, VHDL is rich in data types. In theory, any data type with a finite number of values can be mapped into a set of binary representations and thus can be realized in hardware. However, we refrain from doing this since the mapping introduces another dimension of uncertainty in synthesis and may lead to compatibility problems in larger designs. Our focus is on a small set of predefined data types that are relevant to synthesis. For a signal, we are mainly confined to the `std_logic`, `std_logic_vector`, `signed` and `unsigned` data types. A few user-defined data types will be used for specific applications and they will be discussed as needed.

The following subsections examine the relevant data types, operators and type conversions in VHDL and two synthesis-related IEEE packages.

3.5.1 Predefined data types in VHDL

Commonly used data types There are about a dozen predefined data types in VHDL. Only the following data types are relevant to synthesis:

- `integer`: VHDL does not define the exact range of the `integer` type but specifies that the minimal range is from $-(2^{31}-1)$ to $2^{31}-1$, which corresponds to 32 bits. Two related data types (formally known as *subtypes*) are the `natural` and `positive` data types. The former includes 0 and the positive numbers and the latter includes only the positive numbers.
- `boolean`: defined as `(false, true)`.
- `bit`: defined as `('0', '1')`.
- `bit_vector`: defined as a one-dimensional array with elements of the `bit` data type.

The original intention of the `bit` data type is to represent the two binary values used in Boolean algebra and digital logic. However, in a real design, a signal may assume other values, such as the high impedance of a tri-state buffer's output or a "fighting" value because of a conflict (e.g., two outputs are wired together, forming a short circuit). To solve the problem, a set of more versatile data types, `std_logic` and `std_logic_vector`, are introduced in the IEEE `std_logic_1164` package. To achieve better compatibility, we should avoid using the `bit` and `bit_vector` data types. The `std_logic` and `std_logic_vector` data types are discussed in Section 3.5.2.

In VHDL, data types similar to the `boolean` and `bit` types are known as the *enumeration* data types since their values are enumerated in a list.

Table 3.1 Operators and applicable data types of VHDL-93

Operator	Description	Data type of operand a	Data type of operand b	Data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a mod b	modulo			
a rem b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a sll b	shift-left logical	bit_vector	integer	bit_vector
a srl b	shift-right logical			
a sla b	shift-left arithmetic			
a srl b	shift-right arithmetic			
a rol b	rotate left			
a ror b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a and b	and	boolean, bit, bit_vector	same as a	same as a
a or b	or			
a xor b	xor			
a nand b	nand			
a nor b	nor			
a xnor b	xnor			

Operators About 30 operators are defined in VHDL. In a strongly typed language, the definition of data type includes the operations that can be performed on the object of this data type. It is important to know which data types can be used with a particular operator.

Descriptions of these operators and the applicable data types are summarized in Table 3.1. Only synthesis-related data types are listed. Most operators and data types are self-explanatory. Relational operators and the concatenation operator (&) can be applied to arrays and are discussed in Section 3.5.2.

Note that the operators in the table are defined in VHDL-93. The shift operators and the **xnor** operator are not defined in VHDL-87 and are not supported by IEEE 1076.6 RTL synthesis standard either.

Table 3.2 Precedence of the VHDL operators

Precedence	Operators
Highest	** **abs** **not**
	* / **mod** **rem**
	+ - (identity and negation)
	& + - (addition and subtraction)
	sll **srl** **sla** **sra** **rol** **ror**
	= /= < <= > >=
Lowest	**and** **or** **nand** **nor** **xor** **xnor**

During synthesis, operators in the VHDL code will be realized by physical components. Their hardware complexities vary significantly, and many operators, such as multiplication and division, cannot be synthesized automatically. This issue is discussed in Chapter 6.

The *precedence* of the operators is shown in Table 3.2, which is divided into seven groups. The operators in the same group have the same precedence. The operators in the upper group have higher precedence over the operators in the lower group. For example, consider the expression

```
a + b > c or a < d
```

The + operator will be evaluated first, and then the > and < operators, and then the **or** operator.

If an expression consists of several identical operators, evaluation begins at the leftmost operator and progresses toward the right (known as *left-associative*). For example, consider the expression

```
a + b + c + d
```

The `a + b` expression will be evaluated first, and then `c` is added, and then `d` is added.

Parentheses can be used in an expression. They have the highest preference and thus can alter the order of evaluation. For example, we can use parentheses to make the previous expression be evaluated from right to left:

```
a + (b + (c + d))
```

Unlike the logic expression used in Boolean algebra, the **and** and **or** operators have the same precedence in VHDL, and thus we must use parentheses to specify the desired order, as in

```
(a and b) or (c and d)
```

It is a good practice to use parentheses to make the code clear and readable, even when they are not needed. For example, the expression

```
a + b > c or a < d
```

can be written as

```
((a + b) > c) or (a < d)
```

This is more descriptive and reduces the chance for error or misinterpretation.

Table 3.3 VHDL operators versus conventional Boolean algebra notations

VHDL operator	Boolean algebra notation
not	$'$
and	$\cdot$
or	$+$
xor	$\oplus$
+	$+$

Notation To make an expression compact, we use the conventional symbols $'$, $\cdot$ and $+$ of Boolean algebra for *not*, *and* and *or* operations in our discussion. They are expressed as **not**, **and** and **or** in VHDL code. We also assume that $\cdot$ has precedence over $+$. For example, in our discussion, we may write

$$y = a \cdot b + a' \cdot b'$$

When coded in VHDL, This expression becomes

```
y <= (a and b) or ((not a) and (not b));
```

The notations used in our discussion and VHDL code are summarized in Table 3.3. Note that the + notation is used as both or and addition operations in our discussion. Since they are used in different contexts, it should not introduce confusion.

3.5.2 Data types in the IEEE std_logic_1164 package

To better reflect the electrical property of digital hardware, several new data types were developed by IEEE to serve as an extension to the `bit` and `bit_vector` data types. Theses data types are defined in the `std_logic_1164` package of IEEE standard 1164. In this subsection, we discuss the new data types, the operations defined over these data types and the conversion between these data types and the predefined VHDL data types.

std_logic and std_logic_vector data types The two most useful data types defined in the `std_logic_1164` package are `std_logic` and `std_logic_vector`. Formally speaking, the `std_logic` data type is actually a subtype of the `std_ulogic` data type. Since the `std_ulogic` data type is "unresolved," it has some limitations and will not be used in this book.

To use the new data types, we must include the necessary library and use statements before the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

The `std_logic` data type consists of nine possible values, which are shown in the following list:

```
('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
```

These values are interpreted as follows:

- `'0'` and `'1'`: stand for "forcing logic 0" and "forcing logic 1," which mean that the signal is driven by a circuit with a regular driving current.

- 'Z': stands for high impedance, which is usually encountered in a tri-state buffer.
- 'L' and 'H': stand for "weak logic 0" and "weak logic 1," which means that the signal is obtained from wired-logic types of circuits, in which the driving current is weak.
- 'X' and 'W': stand for "unknown" and "weak unknown." The unknown represents that a signal reaches an intermediate voltage value that can be interpreted as neither logic 0 or logic 1. This may happen because of a conflict in output (such as a logic-0 signal and a logic-1 signal being tied together). They are used in simulation for an erroneous condition.
- 'U': stands for uninitialized. It is used in simulation to indicate that a signal or variable has not yet been assigned a value.
- '-': stands for don't-care.

Among these values, only '0', '1' and 'Z' are used in synthesis. The 'L' and 'H' values are seldom used now since current design practice rarely utilizes a wired-logic circuit. The use of 'Z' and the potential problem of '-' are discussed in Chapter 6.

A VHDL array is defined as a collection of elements with the same data type. Each element in the array is identified by an index. The std_logic_vector data type is an array of elements with the std_logic data type. It can be thought of as a group of signals or a bus in a logic circuit.

The use of std_logic_vector can best be explained by a simple example. Let us consider an 8-bit signal, a. Its declaration is

```
signal a: std_logic_vector(7 downto 0);
```

It indicates that the a signal has 8 bits, which are indexed from 7 down to 0. The most significant bit (MSB, the leftmost bit) has the index 7, and the least significant bit (LSB, the rightmost bit) has the index 0. We can access a single bit by using an index, such as a(7) or a(2), and access a portion of the index by using a range, such as a(7 **downto** 3) or a(2 **downto** 0).

Another form of std_logic_vector is using an ascending range, as in

```
signal a: std_logic_vector(0 to 7);
```

Since its MSB is associated with index 0 and may cause some confusion if the array is interpreted as a binary number, we don't use this form in the book.

Overloaded operators Recall that the definition of a data type includes a set of values and a set of operations to be performed on this data type. In VHDL, we can use the same function or operator name for operands of different data types. There may exist multiple functions with the same name, each for a different data type. This is known as *overloading* of a function or operator.

In the std_logic_1164 package, all logical operators, which include **not**, **and**, **nand**, **or**, **nor**, **xor** and **xnor**, are overloaded with the std_logic and std_logic_vector data types. In other words, we can perform the logical operations over the objects with the std_logic or std_logic_vector data types. The overloaded operators are summarized in Table 3.4. Note that the arithmetic operators are not overloaded, and thus these operations cannot be applied.

Type conversion The std_logic_1164 package also defines several type conversion functions for conversion between the bit and std_logic data types as well as between the bit_vector and std_logic_vector data types. The relevant functions are summarized in Table 3.5.

Table 3.4 Overloaded operators in the IEEE std_logic_1164 package

Overloaded operator	Data type of operand a	Data type of operand b	Data type of result
not a	std_logic_vector std_logic		same as a
a **and** b a **or** b a **xor** b a **nand** b a **nor** b a **xnor** b	std_logic_vector std_logic	same as a	same as a

Table 3.5 Functions in the IEEE std_logic_1164 package

Function	Data type of operand a	Data type of result
to_bit(a)	std_logic	bit
to_stdulogic(a)	bit	std_logic
to_bitvector(a)	std_logic_vector	bit_vector
to_stdlogicvector(a)	bit_vector	std_logic_vector

Use of the conversion function is shown below. Assume that the s1, s2, s3, b1 and b2 signals are defined as

```
signal s1, s2, s3: std_logic_vector(7 downto 0);
signal b1, b2: bit_vector(7 downto 0);
```

The following statements are wrong because of data type mismatch:

```
s1 <= b1;          -- bit_vector assigned to std_logic_vector
b2 <= s1 and s2;   -- std_logic_vector assigned to bit_vector
s3 <= b1 or s2;    -- or is undefined between bit_vector
                   -- and std_logic_vector
```

We can use the conversion functions to correct these problems:

```
s1 <= to_stdlogicvector(b1);
b2 <= to_bitvector(s1 and s2);
s3 <= to_stdlogicvector(b1) or s2;
```

The last statement can also be written as

```
s3 <= to_stdlogicvector(b1 or to_bitvector(s2));
```

3.5.3 Operators over an array data type

Several operations are defined over the one-dimensional array data types in VHDL, including the concatenation and relational operators and the array aggregate. In this subsection, we demonstrate the use of these operators with the std_logic_vector data type. Note that these operators can be applied in any array data types, and thus no overloading is needed.

Relational operators for an array In VHDL, the relational operators can be applied to the one-dimensional array data type. The two operands must have the same element type, but their lengths may differ. When an operator is applied, the two arrays are compared element by element. The comparison procedure starts from the leftmost element and continues until a result can be established. If one array reaches the end before another, that array is considered to be "smaller" and the two arrays are considered to be not equal. For example, all following operations return `true`:

```
"011"="011", "011">"010", "011">"00010", "0110">"011"
```

Arrays with unequal lengths can sometimes introduce subtle, unexpected results. For example, assume that the `sig1` and `sig2` signals are with an array data type of different lengths and we accidentally write

```
if (sig1=sig2) then
. . .
else
. . .
```

Because of the different lengths, the comparison expression is always evaluated as false, and thus the then branch will never be taken. This kind of error is difficult to debug since the code is syntactically correct. In this book, we always use operands of identical length.

Concatenation operator The concatenation operator, `&`, is very useful for array manipulation. We can combine segments of elements and smaller arrays to form a larger array. For example, we can shift the elements of the array to the right by two positions and append two 0's to the front:

```
y <= "00" & a(7 downto 2);
```

or append the MSB to the front (known as an arithmetic shift):

```
y <= a(7) & a(7) & a(7 downto 2);
```

or rotate the elements to the right by two positions:

```
y <= a(1 downto 0) & a(7 downto 2);
```

Array aggregate *Array aggregate* is not an operator. It is a VHDL language construct to assign a value to an object of array data type. For the `std_logic_vector` data type, the simplest way to express an aggregate is to use a collection of `std_logic` values inside double quotation marks. For example, if we want to assign a value of `"10100000"` to the a signal, it can be written as

```
a <= "10100000";
```

Another way is to list each value of the element in the corresponding position, which is known as *positional association*. The previous assignment becomes

```
a <= ('1','0','1','0','0','0','0','0');
```

We can also use the form of `index => value` to explicitly specify the value for each index, known as *named association*. The statement can be written as

```
a <= (7=>'1', 6=>'0', 0=>'0', 1=>'0', 5=>'1',
      4=>'0', 3=>'0', 2=>'0');
```

It means that the value associated with index 7 (i.e., a(7)) is '1', the value associated with index 6 is '0', and so on. Note that the order of the index => value pairs does not matter. We can combine the index, as in

```
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
```

or use a reserved word, **others**, to cover all the unused indexes, as in

```
a <= (7|5=>'1', others=>'0');
```

One frequently encountered array aggregate is all 0's, which is used in the initialization of a counter or a memory element. For example, if we want to assign "00000000" to the a signal, we can write

```
a <= (others=>'0');
```

It is more compact than

```
a <= "00000000";
```

The code remains the same even when the width of the a signal is later revised.

3.5.4 Data types in the IEEE numeric_std package

In addition to logical operations, digital hardware frequently involves arithmetic operation as well. If we examine VHDL and the std_logic_1164 package, the arithmetic operations are defined only over the integer data type. To perform addition of the a and b signals, we must use the integer data type, as in

```
signal a, b, sum: integer;
. . .
sum <= a + b;
```

It is difficult to realize this statement in hardware since the code doesn't indicate the range (number of bits) of the a and b signals. Although this does not matter for simulation, it is important for synthesis since there is a huge difference between the hardware complexity of an 8-bit adder and that of a 32-bit adder.

A better alternative is to use an array of 0's and 1's and interpret it as an unsigned or signed number. We can define the width of the input and the size of the adder precisely, and thus have better control over the underlying hardware. The IEEE numeric_std package was developed for this purpose.

Signed and unsigned data types The IEEE numeric_std package is a part of IEEE standard 1176.3. Two new data types, signed and unsigned, are defined in the package. Both data types are an array of elements with the std_logic data type. For the unsigned data type, the array is interpreted as an unsigned binary number, with the leftmost element as the MSB of the binary number. For the signed data type, the array is interpreted as a signed binary number in 2's-complement format. The leftmost element is the MSB of the binary number, which represents the sign of the number.

Note that the std_logic_vector, unsigned and signed data types are all defined as an array of elements with the std_logic data type. Since VHDL is a strongly typed language, they are considered as three independent data types. It is reasonable since the three data types are interpreted differently. For example, consider a 4-bit binary representation "1100". It represents the number 12 if it is interpreted as an unsigned number and represents the number

−4 if it is interpreted as a signed number. It may also just represent four independent bits (e.g., four status signals) if it is interpreted as a collection of bits.

Since the `signed` and `unsigned` data types are arrays, their declarations are similar to that of the `std_logic_vector` data type, as in

```
signal x, y: signed(15 downto 0);
```

To use the `signed` and `unsigned` data types, we must include the library statement before the entity declaration. Furthermore, we must include the `std_logic_1164` package since the `std_logic` data type is used in the `numeric_std` package. These statements are

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Overloaded operators Since the goal of the `numeric_std` package is to support the arithmetic operations, the relevant arithmetic operators, which include **abs**, *, /, **mod**, **rem**, + and -, are overloaded. These operators can now take two operands, with data types `unsigned` and `unsigned`, `unsigned` and `natural`, `signed` and `signed` as well as `signed` and `integer`. For example, the following are valid assignment statements:

```
signal a, b, c, d: unsigned(7 downto 0);
. . .
a <= b + c;
d <= b + 1;
e <= (5 + a + b) - c;
```

The overloading definition of addition and subtraction follows the model of a physical adder. The sum automatically "wraps around" when overflow occurs.

The relational operators, which include =, /=, <, >, <= and >=, are also overloaded. The overloading serves two purposes. First, it makes the operator take two operands with data types `unsigned` and `natural` as well as `signed` and `integer`. Second, for two operands with the `unsigned` or `signed` data types, the overloading overrides the original left-to-right element-by-element comparison procedure and treats the two arrays as two binary numbers. For example, consider the expression `"011" > "1000"`. If the data type of the two operands is `std_logic_vector`, the expression returns `false` because the first element of `"011"` is smaller than the first element of `"1000"`. If the data type of the two operands is `unsigned`, the > operator is overloaded and the two operands are interpreted as 3 and 8 respectively. The expression returns `false` again. However, if the data type is `signed`, they are interpreted as 3 and −8, and thus the expression returns `true`.

A summary of the overloaded operators is given in Table 3.6.

Functions The `numeric_std` package defines several new functions. The new functions include:

- `shift_left, shift_right, rotate_left, rotate_right:` used for shifting and rotating operations. Note that these are new functions, not the overloaded VHDL operators.
- `resize:` used to convert an array to different sizes.
- `std_match:` used to compare objects with the `'-'` value.
- `to_unsigned, to_signed, to_integer:` used to do type conversion between the two new data types and the `integer` data type.

Table 3.6 Overloaded operators in the IEEE `numeric_std` package

Overloaded operator	Description	Data type of operand a	Data type of operand b	Data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

Table 3.7 Functions in the IEEE `numeric_std` package

Function	Description	Data type of operand a	Data type of operand b	Data type of result
shift_left(a,b) shift_right(a,b) rotate_left(a,b) rotate_right(a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize(a,b)	resize array	unsigned, signed	natural	same as a
std_match(a,b)	compare '-'	unsigned, signed std_logic_vector, std_logic	same as a	boolean
to_integer(a)	data type conversion	unsigned, signed		integer
to_unsigned(a,b)		natural	natural	unsigned
to_signed(a,b)		integer	natural	signed

The functions are summarized in Table 3.7. The shift functions are similar to the VHDL shift operators but with different data types. Note that the IEEE 1076.6 RTL synthesis standard supports the shift functions of the `numeric_std` package but not the shift operators of VHDL. The synthesis issues of the shift functions and the use of the `std_match` function are discussed in Chapter 6.

Type conversion Conversion between two different data types can be done by a *type conversion function* or *type casting*. There are three type conversion functions in the `numeric_std` package: `to_unsigned`, `to_signed`, and `to_integer`. The `to_intger` function takes an object with an `unsigned` or `signed` data type and converts it to the `integer` data type. The `to_unsigned` and `to_signed` functions convert an integer into an object with the `unsigned` or `signed` data type of a specific number of bits. It takes

Table 3.8 Type conversions of numeric data types

Data type of a	To data type	Conversion function/type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

two parameters. The first is the integer number to be converted, and the other specifies the desired number of bits (or size) in the new unsigned or signed data type.

The std_logic_vector, unsigned and signed data types are all defined as an array with elements of the std_logic data type. They are known as *closely related data types* in VHDL. Conversion between these types is done by a procedure known as *type casting*. To do type casting, we simply put the original object inside parentheses prefixed by the new data type. This can best be explained by an example:

```
signal u1, u2: unsigned(7 downto 0);
signal v1, v2: std_logic_vector(7 downto 0);
. . .
u1 <= unsigned(v1)
v2 <= std_logic_vector(u2);
```

Table 3.8 summarizes all the type conversions in the numeric_std package. Note that the std_logic_vector data type is not interpreted as a number and thus cannot be directly converted to an integer and vice versa.

Type conversion between various numeric data types is frequently confusing to new VHDL users. The following examples of signal assignment statements demonstrate and clarify the use of these data types and data conversions. Assume that some signals are declared as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
. . .
```

The following assignments to the signals u3 and u4 are valid since the + operator is overloaded with the unsigned and natural types:

```
u3 <= u2 + u1;   -- ok, both operands unsigned
u4 <= u2 + 1;    -- ok, operands unsigned and natural
```

On the other hand, the following two assignments are invalid due to type mismatch:

```
u5 <= sg;   -- not ok, type mismatch
u6 <= 5;    -- not ok, type mismatch
```

We must use type casting and the conversion function to covert the expressions to the proper type:

```
u5 <= unsigned(sg);       -- ok, type casting
u6 <= to_unsigned(5,4); -- ok, conversion function
```

The arithmetic operators are not overloaded with the mixed data types `signed` and `unsigned`, and thus the following statement is invalid:

```
u7 <= sg + u1;    -- not ok, + undefined over the types
```

We must convert the data type of the operand as follows:

```
u7 <= unsigned(sg) + u1; -- ok, but be careful
```

We need to be aware of the different interpretations of the `signed` and `unsigned` types. For example, `"1111"` is −1 for the `signed` type but is 15 for the `unsigned` type. This kind conversion should proceed with care.

Two assignments for signals with `std_logic_vector` data type are

```
s3 <= u3;   -- not ok, type mismatch
s4 <= 5;    -- not ok, type mismatch
```

Both of them are invalid because of type mismatch. We must use type casting and a conversion function to correct the problem:

```
s3 <= std_logic_vector(u3); -- ok, type casting
s4 <= std_logic_vector(to_unsigned(5,4)); -- ok
```

Note that two type conversions are needed for the second statement.

Arithmetic operations cannot be applied to the `std_logic_vector` data type since no overloading is defined for this type. Thus, the following statements are invalid:

```
s5 <= s2 + s1; -- not ok, + undefined over the types
s6 <= s2 + 1;  -- not ok, + undefined over the types
```

To fix the problem, we must convert the operands to the `unsigned` (or `signed`) data type, perform addition, and then convert the result back to the `std_logic_vector` data type. The revised code becomes

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ok
s6 <= std_logic_vector(unsigned(s2) + 1);            -- ok
```

3.5.5 The std_logic_arith and related packages

For historical reasons, several packages similar to the IEEE `numeric_std` package are used in some EDA software and existing VHDL codes. The packages are:

- `std_logic_arith`
- `std_logic_unsigned`
- `std_logic_signed`

They are not a part of the IEEE standards, but many software vendors store these packages in the `ieee` library. They can be invoked by

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
. . .
```

Because of the use of the ieee term, these packages sometimes cause confusion. They are not used them in this book. For reference, we explain briefly the use of these packages in this subsection.

The purpose of the std_logic_arith package is similar to that of the numeric_std package. It defines two new data types, unsigned and signed, and overloads the +, - and * operators with these data types. The package also includes similar shifting, sizing and type conversion functions although the names of these functions are different.

Instead of defining new data types, the std_logic_unsigned and std_logic_signed packages define overloaded arithmetic operators for the std_logic_vector data type. In other words, the std_logic_vector data type is interpreted as unsigned and signed binary numbers in the std_logic_unsigned and std_logic_signed packages respectively. The two packages clearly cannot be used at the same time.

With one of the packages, the previous code segments becomes valid and no type conversion is needed:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
. . .
s5 <= s2 + s1; -- ok, + overloaded with std_logic_vector
s6 <= s2 + 1;  -- ok, + overloaded with std_logic_vector
```

The overloading means that we can treat the std_logic_vector data type as "a collection of bits" as well as an unsigned binary number. This package actually beats the motivation behind a strongly typed language. The IEEE 1076.6 RTL synthesis standard states explicitly that the unsigned and signed data types defined in IEEE 1076.3 are the only array types that can be used to represent unsigned and signed numbers.

3.6 SYNTHESIS GUIDELINES

In this and subsequent chapters, we summarize the good design and coding practices mentioned in the chapter and present them as a set of guidelines at the end of the chapter. Since the book focuses on synthesis, these guidelines are applied only to synthesis, not to general modeling or simulation. These suggested guidelines help us to avoid some common mistakes and to increase the compatibility, portability and efficiency of VHDL codes.

3.6.1 Guidelines for general VHDL

- Use the std_logic_vector and std_logic data types instead of the bit_vector or bit data types.
- Use the numeric_std package and the unsigned and signed data types for synthesizing arithmetic operations.
- Use only the descending range (i.e., **downto**) in the array specification of the unsigned, signed and std_logic_vector data types.
- Use parentheses to clarify the intended order of evaluation.

- Don't use user-defined data types unless there is a compelling reason.
- Don't use immediate assignment (i.e., :=) to assign an initial value to a signal.
- Use operands with identical lengths for the relational operators.

3.6.2 Guidelines for VHDL formatting

- Include an information header for each file.
- Be consistent with the use of case.
- Use proper spaces, blank lines and indentations to make the code clear.
- Add necessary comments.
- Use symbolic constant names to replace hard literals in VHDL code.
- Use meaningful names for the identifiers.
- Use a suffix to indicate a signal's special property, such as _n for the active-low signal.
- Keep the line width within 72 characters so that the code can be displayed and printed properly by various editors and printers without wrapping.

3.7 BIBLIOGRAPHIC NOTES

VHDL is a complex language. It is formally specified by IEEE standard 1076. The most recent version, VHDL-2001, is specified by IEEE standard 1076-2001, and VHDL-87 is specified by IEEE standard 1076-1987. The standard is documented in *IEEE Standard VHDL Language Reference Manual*, which sometimes known simply as *LRM*. Since *LRM* gives the formal definition of VHDL, it is difficult to read. The book, *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden, provides a detailed and comprehensive discussion of the VHDL language. It has several chapters on basic VHDL concepts, data types and alias. The book, *VHDL for Logic Synthesis* by A. Rushton, has a chapter on `numeric_std` package and provides a detailed discussion on functions.

After synthesis software is installed, we can normally find the files that contain the source codes of IEEE `std_logic_1164` and `numeric_std` packages as well as `std_logic_arith`, `std_logic_unsigned` and `std_logic_signed` packages. These packages provide detailed information about operator overloading and function definitions.

Although formatting is not real design, good coding style and documentation are essential for a project, especially for a large project that involves many design teams. Many organizations set and enforce their own coding and documentation standards. An example is *VHDL Modeling Guideline* from the European Space Agency.

The text, *Reuse Methodology Manual* by M. Keating and P. Bricaud, also provides some rules and guidelines for the use and formatting of VHDL.

Problems

3.1 Write an entity declaration for a memory circuit whose input and output ports are shown below. Use only the `std_logic` or `std_logic_vector` data types.

- addr: 12-bit address input
- wr_n: 1-bit write-enable control signal
- oe_n: 1-bit output-enable control signal
- data: 8-bit bidirectional data bus

3.2 What is the difference between a variable and a signal?

3.3 What is a strongly typed language?

3.4 What is the limitation of using the `bit` data type to represent a physical signal?

3.5 Assume that a is a 10-bit signal with the `std_logic_vector(9 downto 0)` data type. List the 10 bits assigned to the a signal.

(a)
```
a <= (others=>'1');
```
(b)
```
a <= (1|3|5|7|9=>'1', others=>'0');
```
(c)
```
a <= (9|7|2=>'1', 6=>'0', 0=>'1', 1|5|8=>'0', 3|4=>'0');
```

3.6 Assume that a and y are 8-bit signals with the `std_logic_vector(7 downto 0)` data type. If the signals are interpreted as unsigned numbers, the following assignment statement performs a / 8. Explain.

```
y <= "000" & a(7 downto 3);
```

3.7 Assume the same a and y signals in Problem 3.6. We want to perform a **mod** 8 and assign the result to y. Rewrite the previous signal assignment statement using only the & operator.

3.8 Assume that the following double-quoted strings are with the `std_logic_vector` data type. Determine whether the relational operation is syntactically correct. If yes, what is the result (i.e., `true` or `false`)?

(a) `"0110" > "1001"`

(b) `"0110" > "0001001"`

(c) `2#1010# > "1010"`

(d) `1010 > "1010"`

3.9 Repeat Problem 3.8, but assume that the data type is `unsigned`.

3.10 Repeat Problem 3.8, but assume that the data type is `signed`.

3.11 Determine whether the following signal assignment is syntactically correct. If not, use the proper conversion function and type casting to correct the problem.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6, s7: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
. . .
u1 <= 2#0001#;
u2 <= u3 and u4;
u5 <= s1 + 1;
u6 <= u3 + u4 + 3;
u7 <= (others=>'1');
```

```
s2 <= s3 + s4 -1;
s5 <= (others=>'1');
s6 <= u3 and u4;
sg <= u3 - 1;
s7 <= not sg;
```

3.12 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= shift_left(src, amount);
```

3.13 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= src sll amount;
```

3.14 For the following VHDL segment, correct the type mismatch with proper conversion function(s).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
. . .
signal src, dest: std_logic_vector(15 downto 0);
signal amount: std_logic_vector(3 downto 0);
. . .
dest <= src sll amount;
```