4.7 The VHDL Hardware Description Language

In the mid-1980s, the U.S. Department of Defense (DoD) and the IEEE sponsored the development of a highly capable hardware-description language called VHDL. The language started out with and still has the following features:

- Designs may be decomposed hierarchically.
- Each design element has both a well-defined interface (for connecting it to other elements) and a precise behavioral specification (for simulating it).
- Behavioral specifications can use either an algorithm or an actual hardware structure to define an element's operation. For example, an element can be defined initially by an algorithm, to allow design verification of higher-level elements that use it; later, the algorithmic definition can be replaced by a hardware structure.
- Concurrency, timing, and clocking can all be modeled. VHDL handles asynchronous as well as synchronous sequential-circuit structures.
- The logical operation and timing behavior of a design can be simulated.

Thus, VHDL started out as a documentation and modeling language, allowing the behavior of digital-system designs to be precisely specified and simulated.

While the VHDL language and simulation environment were important innovations by themselves, VHDL's utility and popularity took a quantum leap with the commercial development of VHDL synthesis tools. These programs can create logic-circuit structures directly from VHDL behavioral descriptions. Using VHDL, you can design, simulate, and synthesize anything from a simple combinational circuit to a complete microprocessor system on a chip.

VHDL was standardized by the IEEE in 1987 (VHDL-87) and extended in 1993 (VHDL-93). In this section we'll describe a subset of language features that are legal under either standard. We'll describe additional features for sequential logic design in Section 7.12.

4.7.1 Design Flow

It's useful to understand the overall VHDL design environment before jumping into the language itself. There are several steps in a VHDL-based design process, often called the design flow. These steps are applicable to any HDL-based design process and are outlined in Figure 4-50 on page 266.

THE MEANING OF VHDL

"VHDL" stands for "VHSIC Hardware Description Language." VHSIC, in turn, stands for "Very High Speed Integrated Circuit," which was a U.S. Department of Defense program to encourage research on high-performance IC technology (using Very Healthy Sums of Instant Cash!).
At about the same time that VHDL was developing, a different hardware design language appeared on the scene. Verilog HDL, or simply Verilog, was introduced by Gateway Design Automation in 1984 as a proprietary hardware description and simulation language. The subsequent introduction of Verilog-based synthesis tools in 1988 by then-fledgling Synopsys and the 1989 acquisition of Gateway by Cadence Design Systems was a winning combination that led to widespread use of the language.

Today, VHDL and Verilog both enjoy widespread use and share the logic synthesis market roughly 50/50. Verilog has its syntactic roots in C and is in some respects an easier language to learn and use, while VHDL is more like Ada (a DoD-sponsored software programming language) and has more features that support large project development.

Comparing the pros and cons of starting out with one language versus the other, David Pellerin and Douglas Taylor probably put it best in their book, *VHDL Made Easy!* (Prentice Hall, 1997):

> Both languages are easy to learn and hard to master. And once you have learned one of these languages, you will have no trouble transitioning to the other.

The so-called “front end” begins with figuring out the basic approach and building blocks at the block-diagram level. Large logic designs, like software programs, are usually hierarchical, and VHDL gives you a good framework for defining modules and their interfaces and filling in the details later.

The next step is the actual writing of VHDL code for modules, their interfaces, and their internal details. Since VHDL is a text-based language, in principle you can use any text editor for this part of the job. However, most design environments include a specialized VHDL text editor that makes the job a little easier. Such editors include features like automatic highlighting of VHDL keywords, automatic indenting, built-in templates for frequently used program structures, and built-in syntax checking and one-click access to the compiler.

Once you’ve written some code, you will want to compile it, of course. A VHDL compiler analyzes your code for syntax errors and also checks it for compatibility with other modules on which it relies. It also creates the internal information that is needed for a simulator to process your design later. As in other programming endeavors, you probably shouldn’t wait until the very end of coding to compile all of your code. Doing a piece at a time can prevent you from

“Verilog” isn’t an acronym, but it has some interesting anagrams, including “I, Glover” (Danny?), “G.I. lover,” “Go, live!” and “I grovel.” Oh, I suppose it could also be a contraction of “VERIfy LOGic.”
proliferating syntax errors, inconsistent names, and so on, and can certainly give you a much-needed sense of progress when the project end is far from sight!

Perhaps the most satisfying step comes next—simulation. A VHDL simulator allows you to define and apply inputs to your design, and to observe its outputs, without ever having to build the physical circuit. In small projects, the kind you might do as homework in a digital-design class, you would probably generate inputs and observe outputs manually. But for larger projects, VHDL gives you the ability to create "test benches" that automatically apply inputs and compare them with expected outputs.

Actually, simulation is just one piece of a larger step called verification. Sure, it is satisfying to watch your simulated circuit produce simulated outputs, but the purpose of simulation is larger—it is to verify that the circuit works as desired. In a typical large project, a substantial amount of effort is expended both during and after the coding stage to define test cases that exercise the circuit over a wide range of logical operating conditions. Finding design bugs at this stage has a high value; if bugs are found later, all of the so-called "back-end" steps must typically be repeated.

Note that there are at least two dimensions to verification. In functional verification, we study the circuit's logical operation independent of timing considerations; gate delays and other timing parameters are considered to be zero. In timing verification, we study the circuit's operation including estimated delays, and we verify that the setup, hold, and other timing requirements for sequential devices like flip-flops are met. It is customary to perform thorough functional verification before starting the back-end steps. However, our ability to do timing verification at this stage is often limited, since timing may be heavily dependent on the results of synthesis and fitting. We may do preliminary timing verification to gain some comfort with the overall design approach, but detailed timing verification must wait until the end.

After verification, we are ready to move into the "back-end" stage. The nature of and tools for this stage vary somewhat, depending on the target technology for the design, but there are three basic steps. The first is synthesis, converting the VHDL description into a set of primitives or components that can
be assembled in the target technology. For example, with PLDs or CPLDs, the synthesis tool may generate two-level sum-of-products equations. With ASICs, it may generate a list of gates and a netlist that specifies how they should be interconnected. The designer may “help” the synthesis tool by specifying certain technology-specific constraints, such as the maximum number of logic levels or the strength of logic buffers to use.

In the fitting step, a fitting tool or fitter maps the synthesized primitives or components onto available device resources. For a PLD or CPLD, this may mean assigning equations to available AND-OR elements. For an ASIC, it may mean laying down individual gates in a pattern and finding ways to connect them within the physical constraints of the ASIC die; this is called the place-and-route process. The designer can usually specify additional constraints at this stage, such as the placement of modules with a chip or the pin assignments of external input and output pins.

The “final” step is timing verification of the fitted circuit. It is only at this stage that the actual circuit delays due to wire lengths, electrical loading, and other factors can be calculated with reasonable precision. It is usual during this step to apply the same test cases that were used in functional verification, but in this step they are run against the circuit as it will actually be built.

As in any other creative process, you may occasionally take two steps forward and one step back (or worse!). As suggested in the figure, during coding you may encounter problems that force you to go back and rethink your hierarchy, and you will almost certainly have compilation and simulation errors that force you to rewrite parts of the code.

The most painful problems are the ones that you encounter in the back end of the design flow. For example, if the synthesized design doesn’t fit into an available FPGA or doesn’t meet timing requirements, you may have to go back as far as rethinking your whole design approach. That’s worth remembering—excellent tools are still no substitute for careful thought at the outset of a design.

---

**IT WORKS!?**

As a long-time logic designer and system builder, I always thought I knew what it means when someone says about their circuit, “It works!” It means you can go into the lab, power-up a prototype without seeing smoke, and push a reset button and use an oscilloscope or logic analyzer to watch the prototype go through its paces.

But over the years, the meaning of “It works” has changed, at least for some people. When I took a new job a few years ago, I was very pleased to hear that several key ASICs for an important new product were all “working.” But later (just a short time later) I figured out that the ASICs were working only in simulation, and that the design team still had to do several arduous months of synthesis, fitting, timing verification, and repeating, before they could order any prototypes. “It works!”—sure. Just like my kids’ homework—“It’s done!”
4.7.2 Program Structure

VHDL was designed with principles of structured programming in mind, borrowing ideas from the Pascal and Ada software programming languages. A key idea is to define the interface of a hardware module while hiding its internal details. Thus, a VHDL entity is simply a declaration of a module’s inputs and outputs, while a VHDL architecture is a detailed description of the module’s internal structure or behavior.

Figure 4.51(a) illustrates the concept. Many designers like to think of a VHDL entity declaration as a “wrapper” for the architecture, hiding the details of what’s inside while providing the “hooks” for other modules to use it. This forms the basis for hierarchical system design—the architecture of a top-level entity may use (or “instantiate”) other entities, while hiding the architectural details of lower-level entities from the higher-level ones. As shown in (b), a higher-level architecture may use a lower-level entity multiple times, and multiple top-level architectures may use the same lower-level one. In the figure, architectures B, E and F stand alone; they do not use any other entities.

**Figure 4.51** VHDL entities and architectures:
(a) “wrapper” concept; (b) hierarchical use.
VHDL actually allows you to define multiple architectures for a single entity, and it provides a configuration management facility that allows you to specify which one to use during a particular compilation or synthesis run. This lets you try out a different architectural approach without throwing away or hiding your other efforts. However, we won’t use or further discuss this facility in this text.

In the text file of a VHDL program, the *entity declaration* and *architecture definition* are separated, as shown in Figure 4-52. For example, Table 4-26 is a very simple VHDL program for a 2-input “inhibit” gate. In large projects, entities and architectures are sometimes defined in separate files, which the compiler matches up according to their declared names.

Like other high-level programming languages, VHDL generally ignores spaces and line breaks, and these may be provided as desired for readability. *Comments* begin with two hyphens (--) and end at the end of a line.

VHDL defines many special character strings, called *reserved words* or *keywords*. Our example includes several—entity, port, is, in, out, end, architecture, begin, when, else, and not. User-defined *identifiers* begin with a letter and contain letters, digits, and underscores. (An underscore may not follow another underscore or be the last character in an identifier.) Identifiers in the example are Inhibit, X, Y, BIT, Z, and Inhibit_arch. “BIT” is a built-in identifier for a predefined type; it’s not considered a reserved word because it can be redefined. Reserved words and identifiers are not case sensitive.

```
entity Inhibit is -- also known as 'BUT-NOT'
  port (X, Y: in BIT;
       Z: out BIT); -- as in 'X but not Y'
end Inhibit;

architecture Inhibit_arch of Inhibit is
begin
  Z <= '1' when X='1' and Y='0' else '0';
end Inhibit_arch;
```

Table 4-26
VHDL program for an “inhibit” gate.
### Table 4-27
Syntax of a VHDL entity declaration.

<table>
<thead>
<tr>
<th>entity-name is</th>
</tr>
</thead>
<tbody>
<tr>
<td>port (signal-names : mode signal-type;</td>
</tr>
<tr>
<td>signal-names : mode signal-type;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>signal-names : mode signal-type);</td>
</tr>
<tr>
<td>end entity-name;</td>
</tr>
</tbody>
</table>

A basic entity declaration has the syntax shown in Table 4-27. Besides naming the entity, the purpose of the entity declaration is to define its external interface signals or ports in its port declaration part. In addition to the keywords entity, is, port, and end, an entity declaration has the following elements:

- **entity-name**: A user-selected identifier to name the entity.
- **signal-names**: A comma-separated list of one or more user-selected identifiers to name external-interface signals.
- **mode**: One of four reserved words, specifying the signal direction:
  - *in*: The signal is an input to the entity.
  - *out*: The signal is an output of the entity. Note that the value of such a signal cannot be "read" inside the entity’s architecture, only by other entities that use it.
  - *buffer*: The signal is an output of the entity, and its value can also be read inside the entity’s architecture.
  - *inout*: The signal can be used as an input or an output of the entity. This mode is typically used for three-state input/output pins on PLDs.
- **signal-type**: A built-in or user-defined signal type. We’ll have a lot to say about types in the next subsection.

Note that there is no semicolon after the final signal-type; swapping the closing parenthesis with the semicolon after it is a common syntax error for beginning VHDL programmers.

An entity’s ports and their modes and types are all that is seen by other modules that use it. The entity’s internal operation is specified in its architecture definition, whose general syntax is shown in Table 4-28. The entity-name in this definition must be the same as the one given previously in the entity declaration. The architecture-name is a user-selected identifier, usually related to the entity name; it can be the same as the entity name if desired.

An architecture’s external interface signals (ports) are inherited from the port-declaration part of its corresponding entity declaration. An architecture may also include signals and other declarations that are local to that architecture, similar to other high-level languages. Declarations common to multiple entities can be made in a separate “package” used by all entities, as discussed later.
The declarations in Table 4-28 can appear in any order. In due course we’ll discuss many different kinds of declarations and statements that can appear in the architecture definition, but the easiest to start with is the signal declaration. It gives the same information about a signal as in a port declaration, except that no mode is specified:

```vhdl
signal signal-names : signal-type;
```

Zero or more signals can be defined within an architecture, and they roughly correspond to named wires in a logic diagram. They can be read or written within the architecture definition and, like other local objects, can be referenced only within the encompassing architecture definition.

VHDL variables are similar to signals, except that they usually don’t have physical significance in a circuit. In fact, notice that Table 4-28 has no provision for “variable declarations” in an architecture definition. Rather, variables are used in VHDL functions, procedures, and processes, each of which we’ll discuss later. Within these program elements, the syntax of a variable declaration is just like that of a signal declaration, except that the variable keyword is used:

```vhdl
variable variable-names : variable-type;
```

### 4.7.3 Types and Constants

All signals, variables, and constants in a VHDL program must have an associated “type.” The type specifies the set or range of values that the object can take on, and there is also typically a set of operators (such as add, AND, and so on) associated with a given type.

VHDL has just a few predefined types, listed in Table 4-29. In the rest of this book, the only predefined types that we’ll use are integer, character, and boolean. You would think that types with names “bit” and “bit_vector”

<table>
<thead>
<tr>
<th>Type</th>
<th>Character</th>
<th>Severity Level</th>
<th>Predefined Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>character</td>
<td>string</td>
<td>boolean</td>
</tr>
<tr>
<td>bit_vector</td>
<td>integer</td>
<td>string</td>
<td>bits</td>
</tr>
<tr>
<td>boolean</td>
<td>real</td>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>
would be essential in digital design, but it turns out that user-defined versions of these types are more useful, as discussed shortly.

Type integer is defined as the range of integers including at least the range \(-2,147,483,647\) through \(+2,147,483,647\) (\(-2^{31} +1\) through \(+2^{31} -1\)); VHDL implementations may extend this range. Type boolean has two values, true and false. The character type contains all of the characters in the ISO 8-bit character set; the first 128 are the ASCII characters. Built-in operators for the integer and boolean types are listed in Table 4-30.

The most commonly used types in typical VHDL programs are user-defined types, and the most common of these are enumerated types, which are defined by listing their values. Predefined types boolean and character are enumerated types. A type declaration for an enumerated type has the format shown in the first line of Table 4-31. Here, value-list is a comma-separated list (enumeration) of all possible values of the type. The values may be user-defined identifiers or characters (where a “character” is an ISO character enclosed in

<table>
<thead>
<tr>
<th>Table 4-30</th>
<th>Predefined operators for VHDL’s integer and boolean types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>integer Operators</strong></td>
<td><strong>boolean Operators</strong></td>
</tr>
<tr>
<td>+</td>
<td>and</td>
</tr>
<tr>
<td>-</td>
<td>or</td>
</tr>
<tr>
<td>*</td>
<td>nand</td>
</tr>
<tr>
<td>/</td>
<td>nor</td>
</tr>
<tr>
<td>mod</td>
<td>xor</td>
</tr>
<tr>
<td>rem</td>
<td>xnor</td>
</tr>
<tr>
<td>abs</td>
<td>not</td>
</tr>
<tr>
<td>** exponentiation</td>
<td>complementation</td>
</tr>
</tbody>
</table>

would be essential in digital design, but it turns out that user-defined versions of these types are more useful, as discussed shortly.

Type integer is defined as the range of integers including at least the range \(-2,147,483,647\) through \(+2,147,483,647\) (\(-2^{31} +1\) through \(+2^{31} -1\)); VHDL implementations may extend this range. Type boolean has two values, true and false. The character type contains all of the characters in the ISO 8-bit character set; the first 128 are the ASCII characters. Built-in operators for the integer and boolean types are listed in Table 4-30.

The most commonly used types in typical VHDL programs are user-defined types, and the most common of these are enumerated types, which are defined by listing their values. Predefined types boolean and character are enumerated types. A type declaration for an enumerated type has the format shown in the first line of Table 4-31. Here, value-list is a comma-separated list (enumeration) of all possible values of the type. The values may be user-defined identifiers or characters (where a “character” is an ISO character enclosed in

<table>
<thead>
<tr>
<th>Table 4-31</th>
<th>Syntax of VHDL type and constant declarations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>type type-name is (value-list);</td>
<td></td>
</tr>
<tr>
<td>subtype subtype-name is type-name start to end;</td>
<td></td>
</tr>
<tr>
<td>subtype subtype-name is type-name start downto end;</td>
<td></td>
</tr>
<tr>
<td>constant constant-name: type-name := value;</td>
<td></td>
</tr>
</tbody>
</table>

Unlike C, VHDL is a strongly typed language. This means that the compiler does not allow you to assign a value to a signal or variable unless the type of the value precisely matches the declared type of the signal or variable.

Strong typing is both a blessing and a curse. It makes your programs more reliable and easier to debug, because it makes it difficult for you to make “dumb errors” where you assign a value of the wrong type or size. On the other hand, it can be exasperating at times. Even simple operations, such as reinterpreting a 2-bit signal as an integer (for example, to select one of four outcomes in a “case” statement), may require you to call a type-conversion function explicitly.
type STD_ULOGIC is ( 'U', -- Uninitialized 'X', -- Forcing Unknown 'O', -- Forcing 0 '1', -- Forcing 1 'Z', -- High Impedance 'W', -- Weak Unknown 'L', -- Weak 0 'H', -- Weak 1 '-' -- Don't care);
subtype STD_LOGIC is resolved STD_ULOGIC;

Table 4-32
Definition of VHDL std_logic type (see Section 5.6.4 for discussion of "resolved").

single quotes). The first style is used most often to define cases or states for a state machine, for example,

type traffic_light_state is (reset, stop, wait, go);

The second style is used in the very important case of a standard user-defined logic type std_logic, shown in Table 4-32 and part of the IEEE 1164 standard package, discussed in Section 4.7.5. This type includes not only '0' and '1', but seven other values that have been found useful in simulating a logic signal (bit) in a real logic circuit, as explained in more detail in Section 5.6.4.

VHDL also allows users to create subtypes of a type, using the syntax shown in Table 4-31. The values in the subtype must be a contiguous range of values of the base type, from start to end. For an enumerated type, "contiguous" refers to positions in the original, defining value-list. Some examples of subtype definitions are shown below:

subtype twoval_logic is std_logic range '0' to '1';
subtype fourval_logic is std_logic range 'X' to 'Z';
subtype negint is integer range -2147483647 to -1;
subtype bitnum is integer range 31 downto 0;

Notice that the order of a range may be specified in ascending or descending order, depending on whether to or downto is used. There are certain attributes of subtypes for which this distinction is significant, but we don’t use them in this book and we won’t discuss this further.

WHAT A CHARACTER! You may be wondering why the values in the std_logic type are defined as characters rather than one-letter identifiers. Certainly "'0", "'X", and so on would be easier to type than "'O", "'X", and so on. Well, that would require a one-letter identifier other than "'" to be used for don't-care, but that's no big deal. The main reason for using characters is that "'0" and "'1" could not be used, because they’re already recognized as integer constants. This goes back to VHDL’s strong typing; it was not deemed advisable to let the compiler perform an automatic type conversion depending on the context.
Table 4-33
Syntax of VHDL
array declarations.

| type name is array (start to end) of element-type; |
| type name is array (start downto end) of element-type; |
| type name is array (range-type) of element-type; |
| type name is array (range-type range start to end) of element-type; |
| type name is array (range-type range start downto end) of element-type; |

VHDL has two predefined integer subtypes, defined below:

subtype natural is integer range 0 to highest-integer;
subtype positive is integer range 1 to highest-integer;

Constants contribute to the readability, maintainability, and portability of programs in any language. The syntax of a constant declaration in VHDL is shown in the last line of Table 4-31; examples are shown below:

constant BUS_SIZE: integer := 32;  -- width of component
constant MSB: integer := BUS_SIZE-1;  -- bit number of MSB
constant Z: character := 'Z';  -- synonym for Hi-Z value

Notice that the value of a constant can be a simple expression. Constants can be used anywhere the corresponding value can be used, and they can be put to especially good use in type definitions, as we’ll soon show.

Another very important category of user-defined types are array types. Like other languages, VHDL defines an array as an ordered set of elements of the same type, where each element is selected by an array index. Table 4-33 shows several versions of the syntax for declaring an array in VHDL. In the first two versions, start and end are integers that define the possible range of the array index and hence the total number of array elements. In the last three versions, all or a subset of the values of an existing type (range-type) are the range of the array index.

Examples of array declarations are given in Table 4-34. The first pair of examples are very ordinary and show both ascending and descending ranges. The next example shows how a constant, WORD_LEN, can be used with an array declaration, showing also that a range value can be a simple expression. The

UNNATURAL
ACTS

Although VHDL defines the subtype “natural” as being nonnegative integers starting with 0, most mathematicians consider and define the natural numbers to begin with 1. After all, in early history people began counting with 1, and the concept of “0” arrived much later. Still, there is some discussion and perhaps controversy on the subject, especially as the computer age has led more of us to think with 0 as a starting number. For the latest thinking, search the Web for “natural numbers.”
type monthly_count is array (1 to 12) of integer;
type byte is array (7 downto 0) of STD_LOGIC;

constant WORD_LEN: integer := 32;
type word is array (WORD_LEN-1 downto 0) of STD_LOGIC;

constant NUM_REGS: integer := 8;
type reg_file is array (1 to NUM_REGS) of word;

type statecount is array (traffic_light_state) of integer;

third example shows that an array element may itself be an array, thus creating a
two-dimensional array. The last example shows that an enumerated type (or a
subtype) may be specified as the array element range; the array in this example
has four elements, based on our previous definition of traffic_light_state.

Array elements are considered to be ordered from left to right, in the
same direction as index range. Thus, the leftmost elements of arrays of types
monthly_count, byte, word, reg_file, and statecount have indices 1, 7,
31, 1, and reset, respectively.

Within VHDL program statements, individual array elements are accessed
using the array name and the element’s index in parentheses. For example, if M,
B, W, R, and S are signals or variables of the five array types defined in Table 4-34,
then M(11), B(5), W(WORD_LEN-5), R(1,0), R(0), and S(reset) are all valid
elements.

Array literals can be specified by listing the element values in parentheses.
For example, the byte variable B could be set to all ones by the statement

B := ('1','1','1','1','1','1','1','1');

VHDL also has a shorthand notation that allows you to specify values by index.
For example, to set word variable W to all ones except for zeroes in the LSB of
each byte, you can write

W := (0=>'0',8=>'0',16=>'0',24=>'0',others=>'1');

The methods just described work for arrays with any element-type, but the
easiest way to write a literal of a STD_LOGIC array type is to use a “string.” A
VHDL string is a sequence of ISO characters enclosed in double quotes, such as
"Hi there". A string is just an array of characters; as a result, a STD_LOGIC
array of a given length can be assigned the value of a string of the same length,
as long as the characters in the string are taken from the set of nine characters
defined as the possible values of the STD_LOGIC elements—'0', '1', 'U', and
so on. Thus, the previous two examples can be rewritten as follows:

B := "11111111";
W := "11111110111111101111111011111110";
It is also possible to refer to a contiguous subset or slice of an array by specifying the starting and ending indices of the subset, for example, \( M(6 \text{ to } 9) \), \( B(3 \text{ downto } 0) \), \( W(15 \text{ downto } 8) \), \( R(1, 7 \text{ downto } 0) \), \( R(1 \text{ to } 2) \), \( S(\text{stop to go}) \). Notice that the slice’s direction must be the same as the original array’s.

Finally, you can combine arrays or array elements using the concatenation operator \&{}, which joins arrays and elements in the order written, from left to right. For example, `'0'&'1'&"1Z"` is equivalent to "011Z", and the expression \( B(6 \text{ downto } 0) \& B(7) \) yields a 1-bit left circular shift of the 8-bit array \( B \).

The most important array type in typical VHDL programs is the IEEE 1164 standard user-defined logic type `std_logic_vector`, which defines an ordered set of `std_logic` bits. The definition of this type is:

```vhdl
type STD_LOGIC_VECTOR is array (natural range <> ) of STD_LOGIC;
```

This is an example of an unconstrained array type—the range of the array is unspecified, except that it must be a subrange of a defined type, in this case, `natural`. This VHDL feature allows us to develop architectures, functions, and other program elements in a more general way, somewhat independent of the array size or its range of index values. An actual range is specified when a signal or variable is assigned this type. We’ll see examples in the next subsection.

### 4.7.4 Functions and Procedures

Like a function in a high-level programming language, a VHDL function accepts a number of arguments and returns a result. Each of the arguments and the result in a VHDL function definition or function call have a predetermined type.

The syntax of a function definition is shown in Table 4-35. After giving the name of the function, it lists zero or more formal parameters which are used within the function body. When the function is called, the actual parameters in the function call are substituted for the formal parameters. Following VHDL’s

<table>
<thead>
<tr>
<th>Table 4-35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax of a VHDL function definition.</td>
</tr>
</tbody>
</table>

```
function function-name ( 
    signal-names : signal-type;
    signal-names : signal-type;
    ...
    signal-names : signal-type
) return return-type is 
    type declarations 
    constant declarations 
    variable declarations 
    function definitions 
    procedure definitions 
begin 
    sequential-statement 
    ...
    sequential-statement 
end function-name;
```
strong-typing policy, the actual parameters must be the same type or a subtype of the formal parameters. When the function is called from within an architecture, a value of the type return-type is returned in place of the function call.

As shown in the table, a function may define its own local types, constants, variables, and nested functions and procedures. The keywords begin and end enclose a series of “sequential statements” that are executed when the function is called. We'll take a closer look at different kinds of sequential statements and their syntax later, but you should be able to understand the examples here based on your previous programming experience.

The VHDL “inhibit-gate” architecture of Table 4-26 on page 269 is modified in Table 4-36 to use a function. Within the function definition, the keyword return indicates when the function should return to the caller, and it is followed by an expression with the value to be returned to the caller. The type resulting from this expression must match the return-type in the function declaration.

The IEEE 1164 standard logic package defines many functions that operate on the standard types std_logic and std_logic_vector. Besides specifying a number of user-defined types, the package also defines the basic logic operations on these types such as and, or, and xor. This takes advantage of VHDL's ability to overload operators. This facility allows the user to specify a function that is invoked whenever a built-in operator symbol (and, or, +, etc.) is used with a matching set of operand types. There may be several definitions for a given operator symbol; the compiler automatically picks the definition that matches the operand types in each use of the operator.

For example, Table 4-37 contains code, taken from the IEEE package, that shows how the “and” operation is defined for std_logic operands. This code may look complicated, but we've already introduced all of the basic language elements that it uses (except for “resolved”, which we describe in connection with three-state logic in Section 5.6.4).

The inputs to the function may be of type std_ulogic or its subtype std_logic. Another subtype UX01 is defined to be used as the function's return type; even if one of the “and” inputs is a nonlogic value ('Z', 'W', etc.), the

```
architecture Inhibit_archf of Inhibit is

function ButNot (A, B: bit) return bit is
begin
  if B = '0' then return A;
  else return '0';
  end if;
end ButNot;

begin
  Z <= ButNot(X,Y);
end Inhibit_archf;
```

Table 4-36
VHDL program for an "inhibit" function.
Table 4-37
Definitions relating to the “and” operation on STD_LOGIC values in IEEE 1164.

```
SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1';

TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

-- truth table for "and" function
CONSTANT and_table : stdlogic_table := ("

| | U | X | O | 1 | Z | W | L | H |

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>('U', 'U', '0', '0', 'U', 'U', 'U', 'U', 'U'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('0', '0', '0', '0', '0', '0', '0', '0', '0'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('0', '0', '0', '0', '0', '0', '0', '0', '0'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X'), --</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>('U', 'X', '0', 'X', 'X', 'X', 'X', 'X', 'X') ---</td>
<td></td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

);

FUNCTION "and" ( L : std_ulogic; R : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (and_table(L, R));
END "and";
```

The function will return one of only four possible values. Type stdlogic_table defines a two-dimensional, 9\times9 array indexed by a pair of std_ulogic values. For the and_table, the table entries are arranged so that if either index is '0' or 'L' (a weak '0'), the entry is '0'. A '1' entry is found only if both inputs are '1' or 'H' (a weak '1'). Otherwise, a 'U' or 'X' entry appears.

In the function definition itself, double quotes around the function name indicate operator overloading. The “executable” part of the function is just a single statement that returns the table element indexed by the two inputs, L and R, of the “and” function.

Because of VHDL’s strong typing requirements, it’s often necessary to convert a signal from one type to another, and the IEEE 1164 package contains several conversion functions—for example, from BIT to STD_LOGIC or vice versa. A commonly needed conversion is from STD_LOGIC_VECTOR into a corresponding integer value. IEEE 1164 does not include such a conversion function, because different designs may need to use different number interpretations—for example, signed versus unsigned. However, we can write our own conversion function as shown in Table 4-38.

The CONV_INTEGER function uses a simple iterative algorithm equivalent to the nested expansion formula on page 30. We won’t be describing the FOR, CASE, and WHEN statements that it uses until Section 4.7.8, but you should get the idea. The null statement is easy—it means “do nothing.” The range of the FOR loop is specified by "X'range", where the single quote after a signal name
function CONV_INTEGER (X: STD_LOGIC_VECTOR) return INTEGER is 
  variable RESULT: INTEGER;
begin
  RESULT := 0;
  for i in X'range loop
    RESULT := RESULT * 2;
    case X(i) is
      when '0' | 'L' => null;
      when '1' | 'H' => RESULT := RESULT + 1;
      when others => null;
    end case;
  end loop;
  return RESULT;
end CONV_INTEGER;

means “attribute,” and range is a built-in attribute identifier that applies only to arrays and means “range of this array’s index, from left to right.”

In the other direction, we can convert an integer to a STD_LOGIC_VECTOR as shown in Table 4-39. Here we must specify not only the integer value to be converted (ARG), but also the number of bits in the desired result (SIZE). Notice that the function declares a local variable “result”, a STD_LOGIC_VECTOR whose index range is dependent on SIZE. For this reason, SIZE must be a constant or other value that is known when CONV_STD_LOGIC_VECTOR is compiled. To perform the conversion, the function uses the successive-division algorithm that was described on page 30.

A VHDL procedure is similar to a function, except it does not return a result. Whereas a function call can be used in the place of an expression, a procedure call can be used in the place of a statement. VHDL procedures allow their arguments to be specified with type out or inout, so it is actually possible for a procedure to “return” a result. However, we don’t use VHDL procedures in the rest of this book, so we won’t discuss them further.

Table 4-38
VHDL function for converting STD_LOGIC_VECTOR to INTEGER.

Table 4-39
VHDL function for converting INTEGER to STD_LOGIC_VECTOR.