DO NOT COPY

4.7 The VHDL Hardware Design Language

VHDL

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.

VHDL synthesis tools

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-87
VHDL-93

VHDL was standardized by the IEEE in 1987 (*VHDL-87*) and extended in 1993 (*VHDL-93*). In this section we'll 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

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 264.

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!).

VERILOG AND VHDL

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 in 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.

VHDL text editor

Once you’ve written some code, you will of course want to compile it. A *VHDL compiler* analyzes your code for syntax errors and also checks your code 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

THE MEANING OF VERILOG

“Verilog” isn’t an acronym, but it has some interesting palindromes, including “I, Glover” (Danny?), “G.I. lover,” “Go, liver!” and “I grovel.” Oh, I suppose it could also be a contraction of “VERIfy LOGic.”

Copyright © 1999 by John F. Wakerly Copying Prohibited

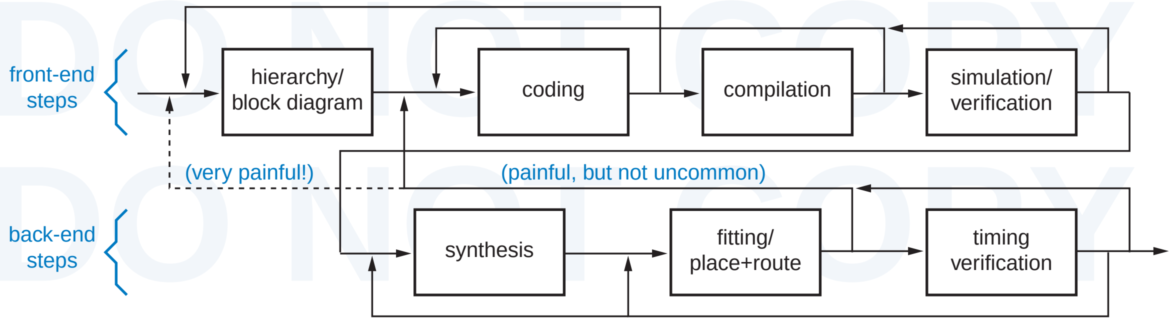

Figure 4-50 Steps in a VHDL or other HDL-based design flow.

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!

VHDL simulator

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.

verification

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.

functional verification

timing verification

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.

synthesis

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

Copyright © 1999 by John F. Wakerly Copying Prohibited

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.

netlist

constraints

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.

fitting

fitter

place and route

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!"

Copyright © 1999 by John F. Wakerly Copying Prohibited

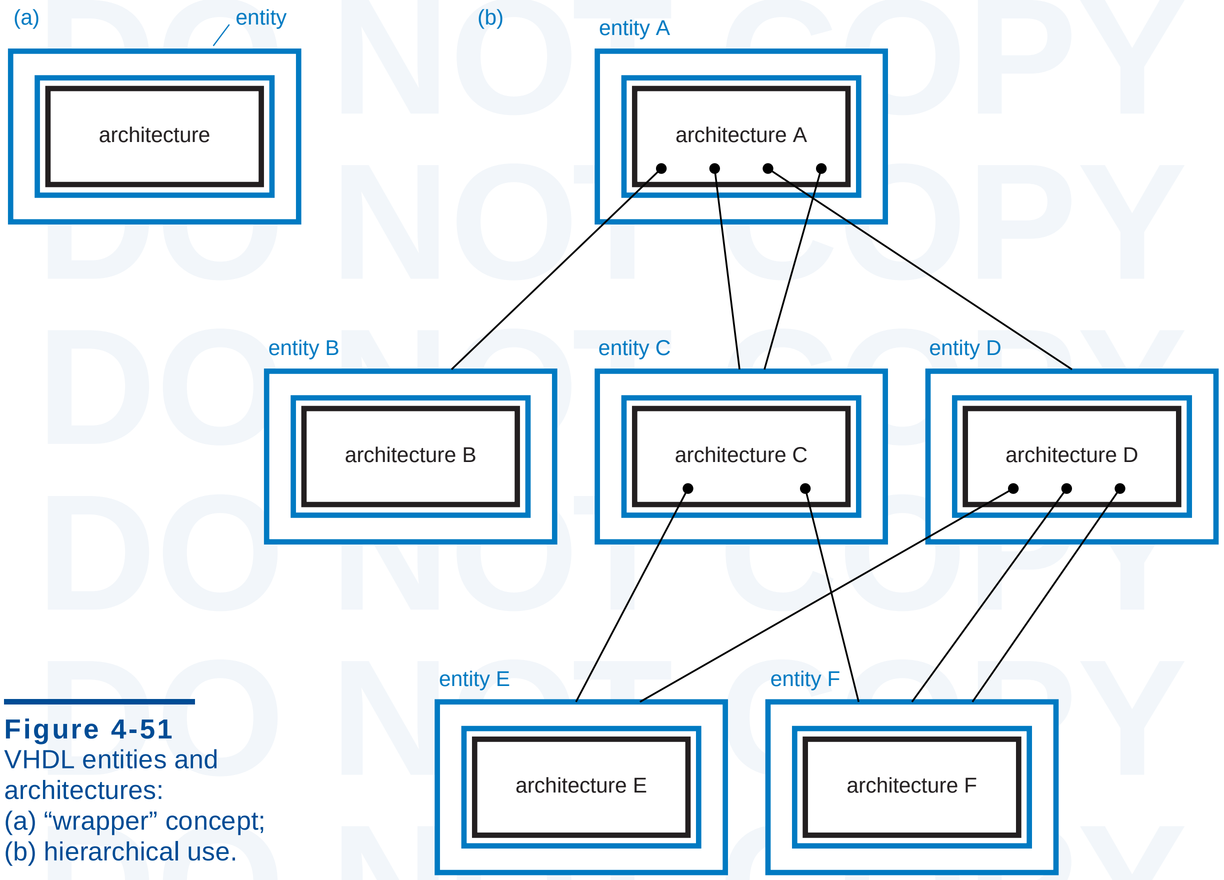

Figure 4-51
VHDL entities and architectures:
(a) “wrapper” concept;
(b) hierarchical use.

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.

entity
architecture

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.

Copyright © 1999 by John F. Wakerly Copying Prohibited

GO CONFIGURE!

VHDL actually allows you to define multiple architectures for a single entity, and provides a configuration management facility that allow 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.

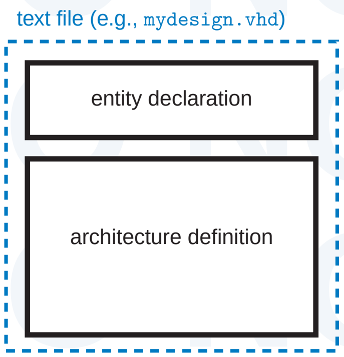

Figure 4-52 VHDL program file structure.

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, and the compiler matches them up according to their declared names.

entity declaration
architecture definition

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.

comments

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.

reserved words
keywords
identifiers

```
entity Inhibit is     -- also known as 'BUT-NOT'
  port (X,Y: in BIT;    --  as in 'X but not Y'
        Z:   out BIT);  --  (see [Klir, 1972])
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.

DO NOT COPY

Table 4-27 Syntax of a VHDL entity declaration.

```
entity entity-name is
  port (signal-names : mode signal-type;
        signal-names : mode signal-type;
        ...
        signal-names : mode signal-type);
end entity-name;
```

An entity declaration has the general 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:

port
port declaration

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.

architecture definition

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.

```
architecture architecture-name of entity-name is
   type declarations
   signal declarations
   constant declarations
   function definitions
   procedure definitions
   component declarations
begin
   concurrent-statement
   . . .
   concurrent-statement
end architecture-name;
```

Table 4-28 Syntax of a VHDL architecture definition.

DO NOT COPY

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:

signal declaration

```
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:

variable

variable declaration

```
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.

type

VHDL has just a few *predefined types*, listed in Table 4.7.3. 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`"

predefined types

`bit`	`character`	`severity_level`
`bit_vector`	`integer`	`string`
`boolean`	`real`	`time`

Table 4-29 VHDL predefined types.

Table 4-30 Predefined operators for VHDL's `integer` and `boolean` types.

`integer` Operators		`boolean` Operators	
`+`	addition	`and`	AND
`-`	subtraction	`or`	OR
`*`	multiplication	`nand`	NAND
`/`	division	`nor`	NOR
`mod`	modulo division	`xor`	Exclusive OR
`rem`	modulo remainder	`xnor`	Exclusive NOR
`abs`	absolute value	`not`	complementation
`**`	exponentiation		

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

integer

boolean

true, false

character

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 characters are the ASCII characters. Built-in operators for the `integer` and `boolean` types are listed in Table 4-30.

user-defined types

enumerated types

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 4-31 Syntax of VHDL type and constant declarations.

```
type type-name is (value-list);

subtype subtype-name is type-name start to end;
subtype subtype-name is type-name start downto end;

constant constant-name: type-name := value;
```

STRONG TYPING

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 because 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
                     '0',  -- 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`").

DO NOT COPY

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.

std_logic

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:

subtypes

```
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.

to
downto

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 "`U`", "`X`", and so on would be easier to type than "`'U'`", "`'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 type-name is array (start to end) of element-type;

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

type type-name is array (range-type) of element-type;

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

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
constant declaration

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 can be put to especially good use in type definitions, as we'll soon show.

array types
array
array index

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, and also shows 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."

Copyright © 1999 by John F. Wakerly Copying Prohibited

```
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;
```

Table 4-34 Examples of VHDL array declarations.

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 element 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(0,0)`, `R(0)`, and `S(reset)` are all valid elements.

Literal values 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');
```

others

The methods just described work for arrays with any *element-type*, but the easiest way write a literal of type `STD_LOGIC` 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:

string

```
B := "11111111";
W := "11111110111111101111111011111110";
```

array slice

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 to 9)`, `B(3 downto 0)`, `W(15 downto 8)`, `R(0,7 downto 0)`, `R(1 to 2)`, `S(stop to go)`. Notice that the slice's direction must be the same as the original array's.

`std_logic_vector`

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:

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

unconstrained array type

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

function
arguments
result
function definition
formal parameters
actual parameters

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 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.

Table 4-35 Syntax of a VHDL function definition.

```
function function-name (
     signal-names : signal-type;
     signal-names : signal-type;
     ...
     signal-names : signal-type
) return return-type is
  type declarations
  constant declarations
  function definitions
  procedure definitions
begin
  sequential-statement
  ...
  sequential-statement
end function-name;
```

DO NOT COPY

As shown in the table, a function may define its own types, constants, and nested functions and procedures which are local to the enclosing function declaration. 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 267 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.

`return`

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` and `or`. 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.

operator overloading

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 non-logic 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';
                                                -- ('U','X','0','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    0    1    Z    W    L    H    -         |   |
--      ----------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);

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

function will return one of only four possible values. Type `stdlogic_table` defines a two-dimensional, 9×9 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 26. 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

`null` statement

```
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;
```

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

range attribute

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 26.

procedure

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 to 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.

```
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
    return STD_LOGIC_VECTOR is
  variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
  variable temp: integer;
begin
  temp := ARG;
  for i in 0 to SIZE-1 loop
    if (temp mod 2) = 1 then result(i) := '1';
    else result(i) := '0';
    end if;
    temp := temp / 2;
  end loop;
  return result;
end;
```

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

DO NOT COPY

DO NOT COPY

4.7.5 Libraries and Packages

library

A VHDL *library* is a place where the VHDL compiler stores information about a particular design project, including intermediate files that are used in the analysis, simulation, and synthesis of the design. The location of the library within a host computer's file system is implementation dependent. For a given VHDL design, the compiler automatically creates and uses a library named "`work`".

A complete VHDL design usually has multiple files, each containing different design units including entities and architectures. When the VHDL compiler analyzes the each file in the design, it places the results in the "`work`" library, and it also searches this library for needed definitions, such as other entities. Because of this feature, a large design can be broken up into multiple files, yet the compiler will find external references as needed.

Not all of the information needed in a design may be in the "`work`" library. For example, a designer may rely on common definitions or functional modules across a family of different projects. Each project has its own "`work`" library (typically a subdirectory within that project's overall directory), but must also refer to a common library containing the shared definitions. Even small projects may use a standard library such as the one containing IEEE standard definitions.

`library` clause

The designer can specify the name of such a library using a *`library` clause* at the beginning of the design file. For example, we can specify the IEEE library:

```
library ieee;
```

The clause "`library work;`" is included implicitly at the beginning of every VHDL design file.

Specifying a library name in a design gives it access to any previously analyzed entities and architectures stored in the library, but it does not give access to type definitions and the like. This is the function of "packages" and "`use` clauses," described next.

package

A VHDL *package* is a file containing definitions of objects that can be used in other programs. The kind of objects that can be put into a package include signal, type, constant, function, procedure, and component declarations.

Signals that are defined in a package are "global" signals, available to any VHDL entity that uses the package. Types and constants defined in a package are known in any file that uses the package. Likewise, functions and procedures defined in a package can be called in files that use the package, and components (described in the next subsection) can be "instantiated" in architectures that use the package.

`use` clause

A design can "use" a package by including a *`use` clause* at the beginning of the design file. For example, to use all of the definitions in the IEEE standard 1164 package, we would write

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

DO NOT COPY

Here, “`ieee`” is the name of a library which has been previously given in a `library` clause. Within this library, the file named “`std_logic_1164`” contains the desired definitions. The suffix “`all`” tells the compiler to use all of the definitions in this file. Instead of “`all`”, you can write the name of a particular object to use just its definition, for example,

```
use ieee.std_logic_1164.std_ulogic
```

This clause would make available just the definition of the `std_ulogic` type in Table 4-32 on page 271, without all of the related types and functions. However, multiple “`use`” clauses can be written to use additional definitions.

Defining packages is not limited to standards bodies. Anyone can write a package, using the syntax shown in Table 4-40. All of the objects declared between “`package`” and the first “`end`” statement are visible in any design file that uses the package; objects following the “`package body`” keyword are local. In particular, notice that the first part includes “function declarations,” not definitions. A *function declaration* lists only the function name, arguments, and type, up to but not including the “`is`” keyword in Table 4-35 on page 274. The complete function definition is given in the package body and is not visible to function users.

function declaration

IEEE VHDL STANDARDS

VHDL has excellent capabilities for extending its data types and functions. This is important, because the language’s built-in `BIT` and `BIT_VECTOR` actually are quite inadequate for modeling real circuits that also handle three-state, unknown, don’t-care, and varying-strength signals.

As a result, soon after the language was formalized as IEEE standard 1076, commercial vendors began to introduce their own built-in data types to deal with logic values other than 0 and 1. Of course, each vendor had different definitions for these extended types, creating a potential “Tower of Babel.”

To avoid this situation, the IEEE developed the 1164 standard logic package (`std_logic_1164`) with a nine-valued logic system that satisfies most designers’ needs. This was later followed by standard 1076-3, discussed in Section 5.9.6, which includes several packages with standard types and operations for vectors of `STD_LOGIC` components that are interpreted as signed or unsigned integers. The packages include `std_logic_arith`, `std_logic_signed`, and `std_logic_unsigned`.

By using IEEE standards, designers can ensure a high degree of portability and interoperability among their designs. This is increasingly important as the deployment of very large ASICs necessitates the cooperation not only from multiple designers, but also from multiple vendors who may each contribute different pieces of a “system-on-a-chip” design.

DO NOT COPY

Table 4-40 Syntax of a VHDL package definition.

```
package package-name is
   type declarations
   signal declarations
   constant declarations
   component declarations
   function declarations
   procedure declarations
end package-name;
package body package-name is
   type declarations
   constant declarations
   function definitions
   procedure definitions
end package-name;
```

4.7.6 Structural Design Elements

We're finally ready to look at the guts of a VHDL design, the "executable" portion of an architecture. Recall from Table 4-28 on page 269 that the body of an architecture is a series of concurrent statements. In VHDL, each *concurrent statement* executes simultaneously with the other concurrent statements in the same architecture body.

concurrent statement

This behavior is markedly different from that of statements in conventional software programming languages, where statements execute sequentially. Concurrent statements are necessary to simulate the behavior of hardware, where connected elements affect each other continuously, not just at particular, ordered time steps. Thus, in a VHDL architecture body, if the last statement updates a signal that is used by the first statement, then the simulator will go back to that first statement and update its results according to the signal that just changed. In fact, the simulator will keep propagating changes and updating results until the simulated circuit stabilizes; we'll discuss this in more detail in Section 4.7.9.

VHDL has several different concurrent statements, as well as a mechanism for bundling a set of sequential statements to operate as a single concurrent statement. Used in different ways, these statements give rise to three somewhat distinct styles of circuit design and description, which we cover in this and the next two subsections.

The most basic of VHDL's concurrent statements is the `component` *statement*, whose basic syntax is shown in Table 4-41. Here, *component-name* is the name of a previously defined entity that is to be used, or *instantiated*, within the current architecture body. One instance of the named entity is created for each

instantiate

Table 4-41 Syntax of a VHDL `component` statement.

```
label: component-name port map(signal1, signal2, ..., signaln);

label: component-name port map(port1=>signal1, port2=>signal2, ..., portn=>signaln);
```

```
component component-name
  port (signal-names : mode signal-type;
        signal-names : mode signal-type;
        ...
        signal-names : mode signal-type);
end component;
```

Table 4-42 Syntax of a VHDL component declaration.

component statement that invokes its name, and each instance must be named by a unique *label*.

port map

The *port map* keywords introduce a list that associates ports of the named entity with signals in the current architecture. The list may be written in either of two different styles. The first is a positional style; as in conventional programming languages, the signals in the list are associated with the entity's ports in the same order that they appear in the entity's definition. The second is an explicit style; each of the entity's ports is connected to a signal using the "=>" operator, and these associations may be listed in any order.

component declaration

Before being instantiated in an architecture, a component must be declared in a *component declaration* in the architecture's definition (see Table 4-28 on page 269). As shown in Table 4-42, a component declaration is essentially the same as the port declaration part of the corresponding entity declaration—it lists the name, mode, and type of each of its ports.

The components used in an architecture may be ones that were previously defined as part of a design, or they may be part of a library. Table 4-43 is an example of a VHDL entity and its architecture that uses components, a "prime-number detector" that is structurally identical to the gate-level circuit in Figure 4-30(c) on page 224. The entity declaration names the inputs and the output of the circuit. The declarations section of the architecture defines all of the signal names and the components that are used internally. The components, INV, AND2, AND3, and OR4, are predefined in the design environment in which this example was created and compiled (Xilinx Foundation 1.5, see References).

Note that component statements in Table 4-43 execute *concurrently*. Even if the statements were listed in a different order, the same circuit would be synthesized, and the simulated circuit operation would be the same.

structural description
structural design

A VHDL architecture that uses components is often called a *structural description* or *structural design*, because it defines the precise interconnection structure of signals and entities that realize the entity. In this regard, a pure structural description is equivalent to a schematic or a net list for the circuit.

In some applications, it is necessary to create multiple copies of a particular structure within an architecture. For example, we'll see in Section 5.10.2 that an *n*-bit "ripple adder" can be created by cascading *n* "full adders." VHDL includes a *generate statement* that allows you to create such repetitive structures using a kind of "for loop," without having to write out all of the component instantiations individually.

DO NOT COPY

Table 4-43 Structural VHDL program for a prime-number detector.

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

entity prime is
    port ( N: in STD_LOGIC_VECTOR (3 downto 0);
           F: out STD_LOGIC );
end prime;

architecture prime1_arch of prime is
signal N3_L, N2_L, N1_L: STD_LOGIC;
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND2 port (I0,I1: in STD_LOGIC; O: out STD_LOGIC); end component;
component AND3 port (I0,I1,I2: in STD_LOGIC; O: out STD_LOGIC); end component;
component OR4 port (I0,I1,I2,I3: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  U1: INV port map (N(3), N3_L);
  U2: INV port map (N(2), N2_L);
  U3: INV port map (N(1), N1_L);
  U4: AND2 port map (N3_L, N(0), N3L_N0);
  U5: AND3 port map (N3_L, N2_L, N(1), N3L_N2L_N1);
  U6: AND3 port map (N2_L, N(1), N(0), N2L_N1_N0);
  U7: AND3 port map (N(2), N1_L, N(0), N2_N1L_N0);
  U8: OR4 port map (N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prime1_arch;
```

The syntax of a simple iterative `generate` loop is shown in Table 4-44. The *identifier* is implicitly declared as a variable with type compatible with the *range*. The *concurrent statement* is executed once for each possible value of the *identifier* within the range, and *identifier* may be used within the concurrent statement. For example, Table 4-45 shows how an 8-bit inverter can be created.

The value of a constant must be known at the time that a VHDL program is compiled. In many applications, it is useful to design and compile an entity and its architecture while leaving some of its parameters, such as bus width, unspecified. VHDL's "generic" facility lets you do this.

generic constant
generic declaration

One or more *generic constants* can be defined in an entity declaration with a *generic declaration* before the port declaration, using the syntax shown in Table 4-46. Each of the named constants can be used within the architecture definition for the entity, and the value of the constant is deferred until the entity is instantiated using a component statement within another architecture. Within that component statement, values are assigned to the generic constants using a

`generic map`

`generic map` clause in the same style as the `port map` clause. Table 4-47 is an example that combines `generic` and `generate` statements to define a "bus inverter" with a user-specifiable width. Multiple copies of this inverter, each with a different width, are instantiated in the program in Table 4-48.

Copyright © 1999 by John F. Wakerly Copying Prohibited

```
label: for identifier in range generate
         concurrent-statement
       end generate;
```

Table 4-44
Syntax of a VHDL `for-generate` loop.

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

entity inv8 is
    port ( X: in STD_LOGIC_VECTOR (1 to 8);
           Y: out STD_LOGIC_VECTOR (1 to 8) );
end inv8;

architecture inv8_arch of inv8 is
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in 1 to 8 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end inv8_arch;
```

Table 4-45
VHDL entity and architecture for an 8-bit inverter.

```
entity entity-name is
  generic (constant-names : constant-type;
           constant-names : constant-type;
           ...
           constant-names : constant-type);
  port (signal-names : mode signal-type;
        signal-names : mode signal-type;
        ...
        signal-names : mode signal-type);
end entity-name;
```

Table 4-46
Syntax of a VHDL generic declaration within an entity declaration.

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

entity businv is
    generic (WIDTH: positive);
    port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
           Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
end businv;

architecture businv_arch of businv is
component INV port (I: in STD_LOGIC; O: out STD_LOGIC); end component;
begin
  g1: for b in WID-1 downto 0 generate
        U1: INV port map (X(b), Y(b));
      end generate;
end businv_arch;
```

Table 4-47
VHDL entity and architecture for an arbitrary-width bus inverter.

Table 4-48
VHDL entity and architecture that use the arbitrary-width bus inverter.

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

entity businv_example is
    port ( IN8: in STD_LOGIC_VECTOR (7 downto 0);
           OUT8: out STD_LOGIC_VECTOR (7 downto 0);
           IN16: in STD_LOGIC_VECTOR (15 downto 0);
           OUT16: out STD_LOGIC_VECTOR (15 downto 0);
           IN32: in STD_LOGIC_VECTOR (31 downto 0);
           OUT32: out STD_LOGIC_VECTOR (31 downto 0) );
end businv_example;

architecture businv_ex_arch of businv_example is
component businv
    generic (WIDTH: positive);
    port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
           Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0) );
end component;
begin
U1: businv generic map (WIDTH=>8) port map (IN8, OUT8);
U2: businv generic map (WIDTH=>16) port map (IN16, OUT16);
U3: businv generic map (WIDTH=>32) port map (IN32, OUT32);
end businv_ex_arch;
```

4.7.7 Dataflow Design Elements

If component statements were VHDL's only concurrent statements, then VHDL would be little more than a strongly typed, hierarchical net-list description language. Several additional concurrent statements allow VHDL to describe a circuit in terms of the flow of data and operations on it within the circuit. This style is called a *dataflow description* or *dataflow design*.

dataflow description
dataflow design
concurrent signal-assignment statement

Two additional concurrent statements used in dataflow designs are shown in Table 4-49. The first of these is the most often used and is called a *concurrent signal-assignment statement*. You can read this as "*signal-name* gets *expression*." Because of VHDL's strong typing, the type of *expression* must be compatible with that of *signal-name*. In general, this means that either the types must be identical or *expression*'s type is a subtype of *signal-name*'s. In the case of arrays, both the element type and the length must match; however, the index range and direction need not match.

Table 4-49
Syntax of VHDL concurrent signal-assignment statements.

```
signal-name <= expression;

signal-name <= expression when boolean-expression else
               expression when boolean-expression else
                ...
               expression when boolean-expression else
               expression;
```

Copyright © 1999 by John F. Wakerly Copying Prohibited

```
architecture prime2_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0     <= not N(3)                           and N(0);
  N3L_N2L_N1 <= not N(3) and not N(2) and     N(1)         ;
  N2L_N1_N0  <=              not N(2) and     N(1) and N(0);
  N2_N1L_N0  <=                  N(2) and not N(1) and N(0);
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime2_arch;
```

Table 4-50 Dataflow VHDL architecture for the prime-number detector.

DO NOT COPY

Table 4-50 shows an architecture for the prime-number detector entity (Table 4-43 on page 282) written in dataflow style. In this style, we don't show the explicit gates and their connections; rather, we use VHDL's built-in `and`, `or`, and `not` operators. (Actually, these operators are not built-in for signals of type `STD_LOGIC`, but they are defined and overloaded by the IEEE 1164 package.) Note that the `not` operator has the highest precedence, so no parentheses are required around subexpressions like "`not N(3)`" to get the intended result.

We can also use the second, *conditional* form of the concurrent signal-assignment statement, using the keywords ***when*** and ***else*** as shown in Table 4-49. Here, a *boolean-expression* combines individual boolean terms using VHDL's built-in boolean operators such as `and`, `or`, and `not`. Boolean terms are typically boolean variables or results of comparisons using *relational operators* `=`, `/=` (not equal), `>`, `>=`, `<`, and `<=`.

conditional signal-assignment statement

when

else

relational operators

=, /=, >, >=, <, <=

Table 4-51 is an example using conditional concurrent assignment statements. Each of the comparisons of a individual `STD_LOGIC` bit such as `N(3)` is made against a character literal `'1'` or `'0'`, and returns a value of type `boolean`. These comparison results are combined in the boolean expression between the `when` and `else` keywords in each statement. The `else` clauses are generally required; the combined set of conditions in a single statement should cover all possible input combinations.

Another kind of concurrent assignment statement is the *selected signal assignment*, whose syntax is shown in Table 4-52. This statement evaluates the given *expression*, and when the value matches one of the *choices*, it assigns the corresponding *signal-value* to *signal-name*. The *choices* in each `when` clause

selected signal-assignment statement

```
architecture prime3_arch of prime is
signal N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0     <= '1' when N(3)='0' and N(0)='1' else '0';
  N3L_N2L_N1 <= '1' when N(3)='0' and N(2)='0' and N(1)='1' else '0';
  N2L_N1_N0  <= '1' when N(2)='0' and N(1)='1' and N(0)='1' else '0';
  N2_N1L_N0  <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
  F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
end prime3_arch;
```

Table 4-51 Prime-number detector architecture using conditional assignments.

DO NOT COPY

Table 4-52
Syntax of VHDL selected signal-assignment statement.

```
with expression select
  signal-name <= signal-value when choices,
                 signal-value when choices,
                 ...
                 signal-value when choices;
```

may be a single value of *expression* or a list of values separated by vertical bars (|). The *choices* for the entire statement must be mutually exclusive and all inclusive. The keyword `others` can be used in the last `when` clause to denote all values of *expression* that have not yet been covered.

others

Table 4-53 is an architecture for the prime-number detector that uses a selected signal-assignment statement. All of the *choices* for which `F` is `'1'` could have been written in a single `when` clause, but multiple clauses are shown just for instructional purposes. In this example, the selected signal-assignment statement reads somewhat like a listing of the on-set of the function `F`.

We can modify the previous architecture slightly to take advantage of the numeric interpretation of `N` in the function definition. Using the `CONV_INTEGER` function that we defined previously, Table 4-54 writes the *choices* in terms of integers, which we can readily see are prime as required. We can think of this version of the architecture is as a “behavioral” description, because it describes the desired function in a way that its behavior is quite evident.

Table 4-53
Prime-number detector architecture using selected signal assignment.

```
architecture prime4_arch of prime is
begin
  with N select
    F <= '1' when "0001",
         '1' when "0010",
         '1' when "0011" | "0101" | "0111",
         '1' when "1011" | "1101",
         '0' when others;
end prime4_arch;
```

COVERING ALL THE CASES

Conditional and selected signal assignments require all possible conditions to be covered. In a conditional signal assignment, the final “`else` *expression*” covers missing conditions. In a selected signal assignment, “`others`” can be used in the final `when` clause to pick up the remaining conditions.

In Table 4-53, you might think that instead of writing “`others`” in the final `when` clause, we could have written the nine remaining 4-bit combinations, `"0000"`, `"0100"`, and so on. But that’s not true! Remember that `STD_LOGIC` is a nine-valued system, so a 4-bit `STD_LOGIC_VECTOR` actually has 9^4 possible values. So “`others`” in this example is really covering 6,554 cases!

```
architecture prime5_arch of prime is
begin
  with CONV_INTEGER(N) select
    F <= '1' when 1 | 2 | 3 | 5 | 7 | 11 | 13,
         '0' when others;
end prime5_arch;
```

Table 4-54 A more behavioral description of the prime-number detector.

DO NOT COPY

4.7.8 Behavioral Design Elements

As we saw in the last example, it is sometimes possible to directly describe a desired logic circuit behavior using a concurrent statement. This is a good thing, as the ability to create a *behavioral design* or *behavioral description* is one of the key benefits of hardware description languages in general and VHDL in particular. However, for most behavioral descriptions, we need to employ some additional language elements described in this subsection.

behavioral design
behavioral description

VHDL's key behavioral element is the "process." A *process* is a collection of "sequential" statements (described shortly) that executes in parallel with other concurrent statements and other processes. Using a process, you can specify a complex interaction of signals and events in a way that executes in essentially zero simulated time during simulation, and that gives rise to a synthesized combinational or sequential circuit that performs the modeled operation directly.

process

A VHDL *process statement* can be used anywhere that a concurrent statement can be used. A process statement is introduced by the keyword `process` and has the syntax shown in Table 4-55 Since a process statement is written within the scope of an enclosing architecture, it has visibility of the types, signals, constants, functions, and procedures that are declared or are otherwise visible in the enclosing architecture. However, you can also define types, signals, constants, functions, and procedures that are local to the process.

process statement
`process`

Note that a process may not declare signals, only "variables." A VHDL *variable* keeps track of the state within a process, and is not visible outside of the process. Depending on its use, it may or may not give rise to a corresponding signal in a physical realization of the modeled circuit. The syntax for defining a

variable

```
process (signal-name, signal-name, ..., signal-name)
  type declarations
  variable declarations
  constant declarations
  function definitions
  procedure definitions
begin
  sequential-statement
  ...
  sequential-statement
end process;
```

Table 4-55 Syntax of a VHDL process statement.

DO NOT COPY

variable within a process is similar to the syntax for a signal declaration within an architecture, except that the keyword *`variable`* is used:

`variable`

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

running process
suspended process
sensitivity list

A VHDL process is always either *running* or *suspended*. The list of signals in the process definition, called the *sensitivity list*, determines when the process runs. A process initially is suspended; when any variable in its sensitivity list changes value, the process resumes execution, starting with its first sequential statement and continuing until the end. If any signal in the sensitivity list change value as a result of running the process, it runs again. This continues until the process runs without any of these signals changing value. In simulation, all of this happens in zero simulated time.

Upon resumption, a properly written process will suspend after one or a few runs. However, it is possible to write an incorrect process that never suspends. For example, consider a process with just one sequential statement, "`X <= not X`" and a sensitivity list of "`(X)`". Since `X` changes on every pass, the process will run forever in zero simulated time, not very useful! In practice, simulators have safeguards that can normally detect such unwanted behavior, and that terminate the misbehaving process after a thousand or so passes.

The sensitivity list is optional; a process without a sensitivity list starts running at time zero in simulation. We'll look at the applications of such processes in \secref{VHDLseqproc}.

sequential signal-assignment statement

VHDL has several kinds of sequential statements. The first is a *sequential signal-assignment statement*; this has the same syntax as the concurrent version (*signal-name* `<=` *expression*`;`), but it occurs within the body of a process rather than an architecture. An analogous statement for variables is the *variable-assignment statement*, which has the syntax "*variable-name* `:=` *expression*`;`". Notice that a different assignment operator, `:=`, is used for variables.

variable- assignment statement

`:=`

For instruction purposes, the dataflow architecture of the prime-number detector in Table 4-50 is rewritten as a process in Table 4-56. Notice that we're still working off of the same original entity declaration of `prime` that appeared in

Table 4-56 Process-based dataflow VHDL architecture for the prime-number detector.

```
architecture prime6_arch of prime6 is
begin
  process(N)
    variable N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  begin
    N3L_N0     := not N(3)                           and N(0);
    N3L_N2L_N1 := not N(3) and not N(2) and     N(1)        ;
    N2L_N1_N0  :=              not N(2) and     N(1) and N(0);
    N2_N1L_N0  :=                  N(2) and not N(1) and N(0);
    F <= N3L_N0 or N3L_N2L_N1 or N2L_N1_N0 or N2_N1L_N0;
  end process;
end prime6_arch;
```

WEIRD BEHAVIOR

Remember that the statements within a process are executed *sequentially*. Suppose that for some reason we wrote the last statement in Table 4-43 (the signal assignment to `F`) as the first. Then we would see rather weird behavior from this process.

The first time the process was run, the simulator would complain that the values of the variables were being read before any value was assigned to them. On subsequent resumptions, a value would be assigned to `F` based on the *previous* values of the variables, which are remembered while the process is suspended. New values would then be assigned to the variables and remembered until the next resumption. So the circuit's output value would always be one input-change behind.

Table 4-43. Within the new architecture (`prime6_arch`), we have just one concurrent statement, which is a process. The process sensitivity list contains just `N`, the primary inputs of the desired combinational logic function. The AND-gate outputs must be defined as variables rather than signals, since signal definitions are not allowed within a process. Otherwise, the body of the process is very similar to that of the original architecture. In fact, a typical synthesis tool would probably create the same circuit from either description.

Other sequential statements, beyond simple assignment, can give us more creative control in expressing circuit behavior. The *`if` statement*, with the syntax shown in Table 4-57, is probably the most familiar of these. In the first and simplest form of the statement, a *boolean-expression* is tested, and a *sequential-statement* is executed if the expression's value is `true`. In the second form,

`if` statement

Table 4-57
Syntax of a VHDL `if` statement.

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
else sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
elsif boolean-expression then sequential-statement
...
elsif boolean-expression then sequential-statement
else sequential-statement
end if;
```

Copyright © 1999 by John F. Wakerly Copying Prohibited

DO NOT COPY

Table 4-58
Prime-number-detector architecture using an `if` statement.

```
architecture prime7_arch of prime is
begin
  process(N)
    variable NI: INTEGER;
  begin
    NI := CONV_INTEGER(N);
    if NI=1 or NI=2 then F <= '1';
    elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F <= '1';
    else F <= '0';
    end if;
  end process;
end prime7_arch;
```

else

we've added an "*else*" clause with another *sequential-statement* that's executed if the expression's value is `false`.

elsif

To create nested `if-then-else` statements, VHDL uses a special keyword *elsif*, which introduces the "middle" clauses. An `elsif` clause's *sequential-statement* is executed if its *boolean-expression* is `true` and all of the preceding *boolean-expressions* were `false`. The optional final `else`-clause's *sequential-statement* is executed if all of the preceding *boolean-expressions* were `false`.

Table 4-58 is a version of the prime-number-detector architecture that uses an `if` statement. An local variable `NI` is used to hold a converted, integer version of the input `N`, so that the comparisons in the `if` statement can be written using integer values.

case statement

The boolean expressions in Table 4-58 are non-overlapping, that is, only one of them is `true` at a time. For this application, we really didn't need the full power of nested `if` statements. In fact, a synthesis engine might create a circuit that evaluates the boolean expressions in series, with slower operation than might otherwise be possible. When we need to select among multiple alternatives based on the value of just one signal or expression, a *case statement* is usually more readable and may yield a better synthesized circuit.

Table 4-59 shows the syntax of a `case` statement. This statement evaluates the given *expression*, finds a matching value in one of the *choices*, and executes the corresponding *sequential-statements*. Note that one or more sequential statements can be written for each set of *choices*. The *choices* may take the form of a single value or multiple values separated by vertical bars (`|`). The *choices* must be mutually exclusive and include all possible values of *expression*'s type; the keyword `others` can be used as the last *choices* to denote all values that have not yet been covered.

Table 4-59
Syntax of a VHDL `case` statement.

```
case expression is
  when choices => sequential-statements
  ...
  when choices => sequential-statements
end case;
```

```
architecture prime8_arch of prime is
begin
  process(N)
  begin
    case CONV_INTEGER(N) is
      when 1 => F <= '1';
      when 2 => F <= '1';
      when 3 | 5 | 7 | 11 | 13 => F <= '1';
      when others => F <= '0';
    end case;
  end process;
end prime8_arch;
```

Table 4-60 Prime-number-detector architecture using a `case` statement.

Table 4-60 is yet another architecture for the prime-number detector, this time coded with a `case` statement. Like the concurrent version, the `select` statement in Table 4-54 on page 287, the `case` statement makes it very easy to see the desired functional behavior.

`loop` statement

Another important class of sequential statements are the *`loop` statements*, The simplest of these has the syntax shown in Table 4-61 and creates an infinite loop. Although infinite loops are undesirable in conventional software programming languages, we'll show in \secref{seqVHDL} how such a loop can be very useful in hardware modeling.

```
loop
  sequential-statement
  ...
  sequential-statement
end loop;
```

Table 4-61 Syntax of a basic VHDL `loop` statement.

`for` loop

A more familiar loop, one that we've seen before, is the *`for` loop*, with the syntax shown in Table 4-62. Note that the loop variable, *identifier*, is declared implicitly by its appearance in the for loop, and has the same type as *range*. This variable may be used within the loop's sequential statements, and it steps through all of the values in *range*, from left to right, one per iteration.

`exit` statement
`next` statement

Two more useful sequential statements that can be executed within a loop are "`exit`" and "`next`". When executed, *`exit`* transfers control to the statement immediately following the loop end. On the other hand, *`next`* causes any remaining statements in the loop to be bypassed, and begins the next iteration of the loop.

```
for identifier in range loop
  sequential-statement
  ...
  sequential-statement
end loop;
```

Table 4-62 Syntax of a VHDL `for` loop.

Copyright © 1999 by John F. Wakerly Copying Prohibited

Table 4-63
Prime-number-detector architecture using a `for` statement.

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

entity prime9 is
    port ( N: in STD_LOGIC_VECTOR (15 downto 0);
           F: out STD_LOGIC );
end prime9;

architecture prime9_arch of prime9 is
begin
  process(N)
  variable NI: INTEGER;
  variable prime: boolean;
  begin
    NI := CONV_INTEGER(N);
    prime := true;
    if NI=1 or NI=2 then null; -- take care of boundary cases
    else for i in 2 to 253 loop
           if NI mod i = 0 then
             prime := false; exit;
           end if;
         end loop;
    end if;
    if prime then F <= '1'; else F <= '0'; end if;
  end process;
end prime9_arch;
```

Our good old prime-number detector is coded one more time in Table 4-63, this time using a `for` loop. The striking thing about this example is that it is truly a behavioral description—we have actually used VHDL compute whether the input `N` is a prime number. We've also increased the size of `N` to 16 bits, just to emphasize the fact that we were able to create a compact model for the circuit without having to explicitly list hundreds of primes.

BAD DESIGN

Table 4-63 has a good example of a `for` loop, but is a bad example of how to design a circuit. Although VHDL is a powerful programming language, design descriptions that use its full power may be inefficient or unsynthesizable.

The culprit in Table 4-63 is the `mod` operator. This operation requires an integer division, and most VHDL tools are unable to synthesize division circuits except for special cases, such as division by a power of two (realized as a shift).

Even if the tools could synthesize a divider, we wouldn't want specify a prime number detector in this way. The description in Table 4-63 implies a combinational circuit, and the tools would have to create 252 combinational dividers, one for each value of `i`, to "unroll" the `for` loop and realize the circuit!

DO NOT COPY

```
while boolean-expression loop
   sequential-statement
   ...
   sequential-statement
end loop;
```

Table 4-64 Syntax of a VHDL `while` loop.

The last kind of `loop` statement is the ***`while`** loop*, with the syntax shown in Table 4-64. In this form, *boolean-expression* is tested before each iteration of the loop, and the loop is executed only of the value of the expression is `true`.

We can use processes to write behavioral descriptions of both combinational and sequential circuits. Many more examples of combinational-circuit descriptions appear in the VHDL subsections of Chapter 5. A few additional language features are needed to describe sequential circuits; these are described in \secref{SeqVHDL}, and sequential examples appear in the VHDL subsections of Chapter 8.

4.7.9 The Time Dimension and Simulation

None of the examples that we've dealt with so far model the time dimension of circuit operation—everything happens in zero simulated time. However, VHDL has excellent facilities for modeling the time, and it is indeed another significant dimension of the language. In this book, we won't go into detail on this subject, but we'll introduce just a few ideas here.

`after`

VHDL allows you to specify a time delay using the keyword *`after`* in any signal assignment statement, including sequential, concurrent, conditional and selected assignments. For example, in the inhibit-gate architecture of Table 4-26 on page 267 you could write

```
Z <= '1' after 4 ns when X='1' and Y='0' else '0' after 3 ns;
```

This allows you to model an inhibit gate that has 4 ns of delay on a 0-to-1 output transition, and only 3 ns on a 1-to-0 transition. In typical ASIC design environments, the VHDL models for all of the low-level components in the component library include such delay parameters. Using these estimates, a VHDL simulator can predict the approximate timing behavior of a larger circuit that uses these components.

`wait`

Another way to invoke the time dimension is with *`wait`*, a sequential statement. This statement can be used to suspend a process for a specified time period. Table 4-65 is an example program that uses `wait` to create simulated input waveforms to test the operation of the inhibit gate for four different input combinations at 10-ns time steps.

Once you have a VHDL program that whose syntax and semantics are correct, you can use a VHDL simulator to observe its operation. Although we won't go into great detail, it's useful to have a basic understanding of how such a simulator works.

DO NOT COPY

Table 4-65 Using the VHDL `wait` statement to generate input waveforms in a test bench program.

```
entity InhibitTestBench is
end InhibitTB_arch;

architecture InhibitTB_arch of InhibitTestBench is
component Inhibit port (X,Y: in BIT; Z: out BIT); end component;
signal XT, YT, ZT: BIT;
begin
  U1: Inhibit port map (XT, YT, ZT);
  process
  begin
    XT <= '0'; YT <= '0';
    wait for 10 ns;
    XT <= '0'; YT <= '1';
    wait for 10 ns;
    XT <= '1'; YT <= '0';
    wait for 10 ns;
    XT <= '1'; YT <= '1';
    wait; -- this suspends the process indefinitely
  end process;
end InhibitTB_arch;
```

Simulator operation begins at a simulated time of zero. At this time, the simulator initializes all signals to a default value (which you shouldn't depend on!). It also initializes any signals or variables for which initial values have been declared explicitly (we didn't show you how to do this). Next, the simulator begins the execution of all the processes and concurrent statements in the design.

Of course, the simulator can't really simulate all of the processes and concurrent statements simultaneously, but it can pretend that it does, using a time-based "event list" and a "signal-sensitivity matrix." Note that each concurrent statement is equivalent to one process.

At time zero, all of the processes are scheduled for execution, and one of these is selected. All of its sequential statements are executed, including any looping behavior that is specified. When the execution of this process is completed, another one is selected, and so on, until all of the processes have been executed. This completes one *simulation cycle*.

simulation cycle

During its execution, a process may assign new values to signals. The new values are not assigned immediately; rather, they are placed on the *event list* and scheduled to become effective at a certain time. If the assignment has an explicit time associated with it (for example, using an `after` clause), then it is scheduled to occur at that time. Otherwise, it is supposed to occur "immediately;" however, it is actually scheduled for the current time plus one "delta delay." The *delta delay* is an infinitesimally short time, shorter than any simulated circuit delay, but long enough to ensure that the new signal value is assigned *after* all of the processes have executed in the current simulation cycle. This ensures that all processes will execute once before any signal changes are propagated.

event list

delta delay

Copyright © 1999 by John F. Wakerly Copying Prohibited

After a simulation cycle completes, the event list is scanned for the signal or signals that change at the next earliest time on the list. The "simulation time" is advanced to this time, and the scheduled signal changes are made. Various processes may be sensitive to the changing signals. The signal-sensitivity matrix indicates, for each signal, which processes have that signal in their sensitivity list. (The process equivalent of a concurrent statement has *all* of its control and data signals in its sensitivity list.) All of the processes that are sensitive to a signal that changed at the current simulation time are scheduled for execution in the next simulation cycle, which now begins.

The simulator's two-phase operation of a simulation cycle followed by advancing the simulation time and making scheduled signal assignments goes on indefinitely, until the event list is empty. At this point, the simulation is complete.

The event-list mechanism makes it possible to simulate the operation of concurrent processes even though the simulator runs on a single computer with a single thread of execution. And the delta-delay mechanism ensures correct operation even though a process or set of processes may require multiple executions, spanning several delta delays, before changing signals settle down to a stable value. This mechanism is also used to detect runaway processes (such as "`X <= not X`"); if a thousand simulation cycles occur over a thousand delta delays without advancing simulation time by any "real" amount, it's most likely that something's amiss.

4.7.10 Synthesis

As we mentioned at the beginning of this section, VHDL was originally invented as a logic circuit description and simulation language and was later adapted to synthesis. The language has many features and constructs that cannot be synthesized. However, the subset of the language and the style of programs that we've presented in this section are generally synthesizable by most tools.

Still, the code that you write can still have a big effect on the quality of the synthesized circuits that you get. A few examples are listed below:

- "Serial" control structures like `if-elsif-elsif-else` can result in a corresponding serial chain of logic gates to test conditions. It's better to use a `case` or `select` statement if the conditions are mutually exclusive.
- Loops in processes are generally "unwound" to create multiple copies of combinational logic to execute the statements in the loop. If you want to use just one copy of the combinational logic in a sequence of steps, then you have to design a sequential circuit, as discussed in later chapters.
- When using conditional statements in a process, failing to state an outcome for some input combination will cause the compiler to create a "latch" to hold the old value of a signal that might otherwise change. Such latches are generally not intended.

Copyright © 1999 by John F. Wakerly Copying Prohibited

DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY DO NOT COPY

In addition, some language features and constructs may just be unsynthesizable, depending on the tool. Naturally, you have to consult the documentation to find out what's disallowed, allowed, and recommended for a particular tool.

For the foreseeable future, digital designers who use synthesis tools will need to pay reasonably close attention to their coding style in order to obtain good results. And for the moment, the definition of "good coding style" depends somewhat on both the synthesis tool and the target technology. The examples in the rest of this book, while syntactically and semantically correct, hardly scratch the surface of coding methods for large HDL-based designs. The art and practice of large HDL-based hardware design is still very much evolving.

References

A historical description of Boole's development of "the science of Logic" appears in *The Computer from Pascal to von Neumann* by Herman H. Goldstine (Princeton University Press, 1972). Claude E. Shannon showed how Boole's work could be applied to logic circuits in "A Symbolic Analysis of Relay and Switching Circuits" (*Trans. AIEE*, Vol. 57, 1938, pp. 713–723).

Huntington postulates

Although the two-valued Boolean algebra is the basis for switching algebra, a Boolean algebra need not have only two values. Boolean algebras with 2^n values exist for every integer n; for example, see *Discrete Mathematical Structures and Their Applications* by Harold S. Stone (SRA, 1973). Such algebras may be formally defined using the so-called *Huntington postulates* devised by E. V. Huntington in 1907; for example, see *Digital Design* by M. Morris Mano (Prentice Hall, 1984). A mathematician's development of Boolean algebra based on a more modern set of postulates appears in *Modern Applied Algebra* by G. Birkhoff and T. C. Bartee (McGraw-Hill, 1970). Our engineering-style, "direct" development of switching algebra follows that of Edward J. McCluskey in his *Introduction to the Theory of Switching Circuits* (McGraw-Hill, 1965) and *Logic Design Principles* (Prentice Hall, 1986).

The prime implicant theorem was proved by W. V. Quine in "The Problem of Simplifying Truth Functions" (*Am. Math. Monthly*, Vol. 59, No. 8, 1952, pp. 521–531). In fact it is possible to prove a more general prime implicant theorem showing that there exists at least one minimal sum that is a sum of prime implicants even if we remove the constraint on the number of literals in the definition of "minimal."

A graphical method for simplifying Boolean functions was proposed by E. W. Veitch in "A Chart Method for Simplifying Boolean Functions" (*Proc. ACM*, May 1952, pp. 127–133). His *Veitch diagram*, shown in Figure 4-53, actually reinvented a chart proposed by an English archaeologist, A. Marquand ("On Logical Diagrams for n Terms," *Philosophical Magazine* XII, 1881, pp. 266–270). The Veitch diagram or Marquand chart uses "natural" binary counting order for its rows and columns, with the result that some adjacent rows and