VHDL Language Reference

Summary

This comprehensive reference provides a detailed overview of the VHDL language and describes each of the standard VHDL keywords (reserved words).

VHDL is a programming language that has been designed and optimized for describing the behavior of digital systems.

VHDL has many features appropriate for describing the behavior of electronic components ranging from simple logic gates to complete microprocessors and custom chips. Features of VHDL allow electrical aspects of circuit behavior (such as rise and fall times of signals, delays through gates, and functional operation) to be precisely described. The resulting VHDL simulation models can then be used as building blocks in larger circuits (using schematics, block diagrams or system-level VHDL descriptions) for the purpose of simulation.

VHDL is also a general-purpose programming language: just as high-level programming languages allow complex design concepts to be expressed as computer programs, VHDL allows the behavior of complex electronic circuits to be captured into a design system for automatic circuit synthesis or for system simulation. Like Pascal, C and C++, VHDL includes features useful for structured design techniques, and offers a rich set of control and data representation features. Unlike these other programming languages, VHDL provides features allowing concurrent events to be described. This is important because the hardware described using VHDL is inherently concurrent in its operation.

One of the most important applications of VHDL is to capture the performance specification for a circuit, in the form of what is commonly referred to as a test bench. Test benches are VHDL descriptions of circuit stimuli and corresponding expected outputs that verify the behavior of a circuit over time. Test benches should be an integral part of any VHDL project and should be created in tandem with other descriptions of the circuit.

A Standard Language

One of the most compelling reasons for you to become experienced with and knowledgeable in VHDL is its adoption as a standard in the electronic design community. Using a standard language such as VHDL virtually guarantees that you will not have to throw away and recapture design concepts simply because the design entry method you have chosen is not supported in a newer generation of design tools. Using a standard language also means that you are more likely to be able to take advantage of the most up-to-date design tools and that you will have access to a knowledge base of thousands of other engineers, many of whom are solving problems similar to your own.

Entities and Architectures

Every VHDL design description consists of at least one entity/architecture pair. (In VHDL jargon, this combination of an entity and its corresponding architecture is sometimes referred to as a **design entity**.) In a large design, you will typically write many entity/architecture pairs and connect them together to form a complete circuit.

An **entity declaration** describes the circuit as it appears from the “outside” - from the perspective of its input and output interfaces. If you are familiar with schematics, you might think of the entity declaration as being analogous to a block symbol on a schematic.

The second part of a minimal VHDL design description is the architecture declaration. Before simulation or synthesis can proceed, every referenced entity in a VHDL design description must be bound with a corresponding architecture. The architecture describes the actual function – or contents – of the entity to which it is bound. Using the schematic as a metaphor, you can think of the architecture as being roughly analogous to a lower-level schematic referenced by the higher-level functional block symbol.

Entity Declaration

An entity declaration provides the complete interface for a circuit. Using the information provided in an entity declaration (the names, data types and direction of each port), you have all the information you need to connect that portion of a circuit into other, higher-level circuits, or to develop input stimuli (in the form of a test bench) for verification purposes. The actual operation of the circuit, however, is not included in the entity declaration.

Let's take a closer look at the entity declaration for this simple design description:

```
entity compare is
```

```
    port( A, B: in bit_vector(0 to 7);
              EQ: out bit);
end compare;
```

The entity declaration includes a name, `compare`, and a **port** statement defining all inputs and outputs of the entity. The port list includes definitions of three ports: `A`, `B`, and `EQ`. Each of these three ports is given a direction (either **in**, **out** or **inout**), and a type (in this case either `bit_vector(0 to 7)`, which specifies an 8-bit array, or `bit`, which represents a single-bit value).

There are many different data types available in VHDL. To simplify things in this introductory circuit, we're going to stick with the simplest data types in VHDL, which are **bit** and **bit_vector**.

Architecture Declaration and Body

The second part of a minimal VHDL source file is the architecture declaration. Every entity declaration you reference in your design must be accompanied by at least one corresponding architecture (we'll discuss why you might have more than one architecture in a moment).

Here's the architecture declaration for the comparator circuit:

```
architecture compare1 of compare is
begin
    EQ <= '1' when (A = B) else '0';
end compare1;
```

The architecture declaration begins with a unique name, `compare1`, followed by the name of the entity to which the architecture is bound, in this case `compare`. Within the architecture declaration (between the **begin** and **end** keywords) is the actual functional description of the comparator. There are many ways to describe combinational logic functions in VHDL; the method used in this description is a type of concurrent statement known as a **conditional assignment**. This assignment specifies that the value of the output (`EQ`) will be assigned a value of `'1'` when `A` and `B` are equal, and a value of `'0'` when they differ.

This single concurrent assignment demonstrates the simplest form of a VHDL architecture. As you will see, there are many different types of concurrent statements available in VHDL, allowing you to describe very complex architectures. Hierarchy and subprogram features of the language allow you to include lower-level components, subroutines and functions in your architectures, and a powerful statement known as a **process** allows you to describe complex registered sequential logic as well.

Data Types

Like a high-level software programming language, VHDL allows data to be represented in terms of high-level data types. A data type is an abstract representation of stored data, such as you might encounter in software languages. These data types might represent individual wires in a circuit, or they might represent collections of wires.

The preceding description of the comparator circuit used the data types bit and bit_vector for its inputs and outputs. The bit data type has only two possible values: '1' or '0'. (A bit_vector is simply an array of bits.) Every data type in VHDL has a defined set of values, and a defined set of valid operations. Type checking is strict, so it is not possible, for example, to directly assign the value of an integer data type to a bit_vector data type. (There are ways to get around this restriction, using what are called type conversion functions.

The chart below summarizes the fundamental data types available in VHDL.

Data Type	Values	Example
Bit	'1', '0'	Q <= '1';
Bit_vector	(array of bits)	DataOut <= “00010101”;
Boolean	True, False	EQ <= True;
Integer	-2, -1, 0, 1, 2, 3, 4 . . .	Count <= Count + 2;
Real	1.0, -1.0E5	V1 = V2 / 5.3
Time	1 us, 7 ns, 100 ps	Q <= '1' after 6 ns;
Character	'a', 'b', '2, '$', etc.	CharData <= 'X';
String	(Array of characters)	Msg <= “MEM: “ & Addr

Notes

The VHDL symbol <= is an assignment operator that assigns the value(s) on its right to the variable on its left.

Design Units

One concept unique to VHDL (when compared to software programming languages and to its main rival, Verilog) is the concept of a design unit. Design units in VHDL (which may also be referred to as library units) are segments of VHDL code that can be compiled separately and stored in a library.

There are actually five types of design units in VHDL; **entities**, **architectures**, **packages**, **package bodies**, and **configurations**. Entities and architectures are the only two design units that you must have in any VHDL design description. Packages and configurations are optional.

Entities

A VHDL entity is a statement (indicated by the **entity** keyword) that defines the external specification of a circuit or sub-circuit. The minimum VHDL design description must include at least one entity and one corresponding architecture.

When you write an entity declaration, you must provide a unique name for that entity and a port list defining the input and output ports of the circuit. Each port in the port list must be given a name, direction (or mode, in VHDL jargon) and a type. Optionally, you may also include a special type of parameter list (called a generic list) that allows you to pass additional information into an entity.

An example of an entity declaration is given below:

```
entity fulladder is
    port (X: in bit;
          Y: in bit;
          Cin: in bit;
          Cout: out bit;
          Sum: out bit);
end fulladder;
```

Architectures

A VHDL architecture declaration is a statement (beginning with the **architecture** keyword) that describes the underlying function and/or structure of a circuit. Each architecture in your design must be associated (or bound) by name with one entity in the design.

VHDL allows you to create more than one alternate architecture for each entity. This feature is particularly useful for simulation and for project team environments in which the design of the system interfaces (expressed as entities) is performed by a different engineer than the lower-level architectural description of each component circuit, or when you simply want to experiment with different methods of description.

An architecture declaration consists of zero or more declarations (of items such as intermediate signals, components that will be referenced in the architecture, local functions and procedures, and constants) followed by a **begin** statement, a series of concurrent statements, and an **end** statement, as illustrated by the following example:

```
architecture concurrent of fulladder is
begin
    Sum <= X xor Y xor Cin;
    Cout <= (X and Y) or (X and Cin) or (Y and Cin);
end concurrent;
```

Packages and Package Bodies

A VHDL package declaration is identified by the **package** keyword, and is used to collect commonly-used declarations for use globally among different design units. You can think of a package as a common storage area, one used to store such things as type declarations, constants, and global subprograms. Items defined within a package can be made visible to any other design unit in the complete VHDL design, and they can be compiled into libraries for later re-use.

A package can consist of two basic parts: a package declaration and an optional package body. Package declarations can contain the following types of statements:

- Type and subtype declarations
- Constant declarations

- Global signal declarations
- Function and procedure declarations
- Attribute specifications
- File declarations
- Component declarations
- Alias declarations
- Disconnect specifications
- Use clauses

Items appearing within a package declaration can be made visible to other design units through the use of a **use** statement.

If the package contains declarations of subprograms (functions or procedures) or defines one or more deferred constants (constants whose value is not immediately given), then a package body is required in addition to the package declaration. A package body (which is specified using the **package body** keyword combination) must have the same name as its corresponding package declaration, but it can be located anywhere in the design, in the same or a different source file.

The relationship between a package and package body is somewhat akin to the relationship between an entity and its corresponding architecture. (There may be only one package body written for each package declaration, however.) While the package declaration provides the information needed to use the items defined within it (the parameter list for a global procedure, or the name of a defined type or subtype), the actual behavior of such things as procedures and functions must be specified within package bodies.

An example of a package is given below:

```
package conversion is
    function to_vector (size: integer; num: integer) return std_logic_vector;
end conversion;
package body conversion is
    function to_vector(size: integer; num: integer) return std_logic_vector is
        variable ret: std_logic_vector (1 to size);
        variable a: integer;
    begin
        a := num;
        for i in size downto 1 loop
            if ((a mod 2) = 1) then
                ret(i) := '1';
            else
                ret(i) := '0';
            end if;
            a := a / 2;
        end loop;
        return ret;
    end to_vector;
end conversion;
```

Configurations

The final type of design unit available in VHDL is called a configuration declaration. You can think of a configuration declaration as being roughly analogous to a parts list for your design. A configuration declaration (identified with the **configuration** keyword) specifies which architectures are to be bound to which entities, and it allows you to change how components are connected in your design description at the time of simulation. (Configurations are not generally used for synthesis, and may not be supported in the synthesis tool that you will use.)

Configuration declarations are always optional, no matter how complex a design description you create. In the absence of a configuration declaration, the VHDL standard specifies a set of rules that provide you with a default configuration. For example, in the case where you have provided more than one architecture for an entity, the last architecture compiled will take precedence and will be bound to the entity.

A simple example of a configuration is given below:

```
configuration this_build of rcomp is
    for structure
        for COMP1: compare use entity work.compare(compare1);
        for ROT1: rotate use entity work.rotate(rotate1);
    end for;
end this_build;
```

Levels of Abstraction

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. When talking about the different styles of VHDL, the differing levels of abstraction possible using the language are being considered - **behavior**, **dataflow**, and **structure**.

Suppose the performance specifications for a given project are: “the compressed data coming out of the DSP chip needs to be analyzed and stored within 70 nanoseconds of the strobe signal being asserted...” This human language specification must be refined into a description that can actually be simulated. A test bench written in combination with a sequential description is one such expression of the design. These are all points in the behavior level of abstraction.

After this initial simulation, the design must be further refined until the description is something a VHDL synthesis tool can digest. Synthesis is a process of translating an abstract concept into a less-abstract form. The highest level of abstraction accepted by today's synthesis tools is the dataflow level.

The structure level of abstraction comes into play when little chunks of circuitry are to be connected together to form bigger circuits. (If the little chunks being connected are actually quite large chunks, then the result is commonly called a block diagram.) Physical information is the most basic level of all and is outside the scope of VHDL. This level involves actually specifying the interconnects of transistors on a chip, placing and routing macrocells within a gate array or FPGA, etc.

As an example of these three levels of abstraction, it is possible to describe a complex controller circuit in a number of ways. At the lowest level of abstraction (the structural level), you could use VHDL's hierarchy features to connect a sequence of predefined logic gates and flip-flips to form the complete circuit. To describe this same circuit at a dataflow level of abstraction, you might describe the combinational logic portion of the controller (its input decoding and transition logic) using higher-level Boolean logic functions and then feed the output of that logic into a set of registers that match the registers available in some target technology. At the behavioral level of abstraction, you might ignore the target technology (and the requirements of synthesis tools) entirely and instead describe how the controller operates over time in response to various types of stimulus.

Behavior

The highest level of abstraction supported in VHDL is called the behavioral level of abstraction. When creating a behavioral description of a circuit, you will describe your circuit in terms of its operation over time. The concept of time is the critical distinction between behavioral descriptions of circuits and lower-level descriptions (specifically descriptions created at the dataflow level of abstraction).

Examples of behavioral forms of representation might include state diagrams, timing diagrams and algorithmic descriptions.

In a behavioral description, the concept of time may be expressed precisely, with actual delays between related events (such as the propagation delays within gates and on wires), or it may simply be an ordering of operations that are expressed sequentially (such as in a functional description of a flip-flop). When you are writing VHDL for input to synthesis tools, you may use behavioral statements in VHDL to imply that there are registers in your circuit. It is unlikely, however, that your synthesis tool will be capable of creating precisely the same behavior in actual circuitry as you have defined in the language. (Synthesis tools today ignore detailed timing specifications, leaving the actual timing results at the mercy of the target device technology.) It is also unlikely that your synthesis tool will be capable of accepting and processing a very wide range of behavioral description styles.

If you are familiar with software programming, writing behavior-level VHDL will not seem like anything new. Just like a programming language, you will be writing one or more small programs that operate sequentially and communicate with one another through their interfaces. The only difference between behavior-level VHDL and a software programming language is the underlying execution platform: in the case of software, it is some operating system running on a CPU; in the case of VHDL, it is the simulator and/or the synthesized hardware.

Dataflow

In the dataflow level of abstraction, you describe your circuit in terms of how data moves through the system. At the heart of most digital systems today are registers, so in the dataflow level of abstraction you describe how information is passed between registers in the circuit. You will probably describe the combinational logic portion of your circuit at a relatively high level (and let

a synthesis tool figure out the detailed implementation in logic gates), but you will likely be quite specific about the placement and operation of registers in the complete circuit.

The dataflow level of abstraction is often called **register transfer logic**, or **RTL**. This level of abstraction is an intermediate level that allows the drudgery of combinational logic to be simplified (and, presumably, taken care of by logic synthesis tools) while the more important parts of the circuit, the registers, are more completely specified.

There are some drawbacks to using a dataflow method of design in VHDL. First, there are no built-in registers in VHDL; the language was designed to be general-purpose, and the emphasis was placed by VHDL's designers on its behavioral aspects. If you are going to write VHDL at the dataflow level of abstraction, you must first create (or obtain) behavioral descriptions of the register elements you will be using in your design. These elements must be provided in the form of components (using VHDL's hierarchy features) or in the form of subprograms (functions or procedures).

But for hardware designers, it can be difficult to relate the sequential descriptions and operation of behavioral VHDL with the hardware being described (or modeled). For this reason, many VHDL users, particularly those who are using VHDL as an input to synthesis, prefer to stick with levels of abstraction that are easier to relate to actual hardware devices (such as logic gates and flip-flops). These users are often more comfortable using the dataflow level of abstraction.

Structure

The third level of abstraction, structure, is used to describe a circuit in terms of its components. Structure can be used to create a very low-level description of a circuit (such as a transistor-level description) or a very high-level description (such as a block diagram).

In a gate-level description of a circuit, for example, components such as basic logic gates and flip-flops might be connected in some logical structure to create the circuit. This is what is often called a netlist. For a higher-level circuit – one in which the components being connected are larger functional blocks – structure might simply be used to segment the design description into manageable parts.

Structure-level VHDL features, such as components and configurations, are very useful for managing complexity. The use of components can dramatically improve your ability to re-use elements of your designs, and they can make it possible to work using a top-down design approach.

To give an example of how a structural description of a circuit relates to higher levels of abstraction, consider the design of a simple 5-bit counter. To describe such a counter using traditional design methods, you might connect five T flip-flops with some simple decode logic.

The following VHDL design description represents this design in the form of a netlist of connected components:

```
entity andgate is
    port(A,B,C,D: in bit := '1'; Y: out bit);
end andgate;
architecture gate of andgate is
begin
    Y <= A and B and C and D;
end gate;
entity tff is
    port(Rst,Clk,T: in bit; Q: out bit);
end tff;
architecture behavior of tff is
begin
    process(Rst,Clk)
        variable Qtmp: bit;
    begin
        if (Rst = '1') then
            Qtmp := '0';
        elsif Clk = '1' and Clk'event then
            if T = '1' then
                Qtmp := not Qtmp;
```

```
            end if;
        end if;
        Q <= Qtmp;
    end process;
end behavior;
entity TCOUNT is
    port (Rst,Clk: in bit;
          Count: out bit_vector(4 downto 0));
end TCOUNT;
architecture STRUCTURE of TCOUNT is
    component tff
        port(Rst,Clk,T: in bit; Q: out bit);
    end component;
    component andgate
        port(A,B,C,D: in bit := '1'; Y: out bit);
    end component;
    constant VCC: bit := '1';
    signal T,Q: bit_vector(4 downto 0);
begin
    T(0) <= VCC;
    T0: tff port map (Rst=>Rst, Clk=>Clk, T=>T(0), Q=>Q(0));
    T(1) <= Q(0);
    T1: tff port map (Rst=>Rst, Clk=>Clk, T=>T(1), Q=>Q(1));
    A1: andgate port map(A=>Q(0), B=>Q(1), Y=>T(2));
    T2: tff port map (Rst=>Rst, Clk=>Clk, T=>T(2), Q=>Q(2));
    A2: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), Y=>T(3));
    T3: tff port map (Rst=>Rst, Clk=>Clk, T=>T(3), Q=>Q(3));
    A3: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), D=>Q(3), Y=>T(4));
    T4: tff port map (Rst=>Rst, Clk=>Clk, T=>T(4), Q=>Q(4));
    Count <= Q;
end STRUCTURE;
```

This structural representation seems a straightforward way to describe a 5-bit counter, and it is certainly easy to relate to hardware since just about any imaginable implementation technology will have the features necessary to implement the circuit. For larger circuits, however, such descriptions quickly become impractical.

Notes

In some formal discussions of synthesis, four levels of abstraction are described; behavior, RTL, gate-level and layout. The three levels of abstraction presented here provide the most useful distinctions for today's synthesis user.

Objects, Data Types and Operators

VHDL includes a number of language elements, collectively called objects, that can be used to represent and store data in the system being described. The three basic types of objects that you will use when entering a design description for synthesis or creating functional tests (in the form of a test bench) are signals, variables and constants. Each object that you declare has a specific data type (such as bit or integer) and a unique set of possible values.

The values that an object can take will depend on the definition of the type used for that object. For example, an object of type bit has only two possible values, '0' and '1', while an object of type real has many possible values (floating point numbers within a precision and range defined by the VHDL standard and by the specific simulator you are using).

When an explicit value is specified (such as when you are assigning a value to a signal or variable, or when you are passing a value as a parameter to a subprogram), that value is represented in the form of a literal.

Using Signals

Signals are objects that are used to connect concurrent elements (such as components, processes and concurrent assignments), similar to the way that wires are used to connect components on a circuit board or in a schematic. Signals can be declared globally in an external package or locally within an architecture, block or other declarative region.

To declare a signal, you write a **signal** statement such as the following:

```
architecture arch1 of my_design is
    signal Q: std_logic;
begin
    . . .
end arch1;
```

In this simple example, the signal `Q` is declared within the declaration section of the `arch1` architecture. At a minimum, a signal declaration must include the name of the signal (in this case `Q`) and its type (in this case the standard type `std_logic`). If more than one signal of the same type is required, multiple signal names can be specified in a single declaration:

```
architecture arch2 of my_design is
    signal Bus1, Bus2: std_logic_vector(7 downto 0);
begin
    . . .
end declare;
```

In the first example above, the declaration of `Q` was entered in the declaration area of architecture `arch1`. Thus, the signal `Q` will be visible anywhere within the `arch1` design unit, but it will not be visible within other design units. To make the signal `Q` visible to the entire design (a global signal), you would have to move the declaration into an external package, as shown below:

```
package my_package is
    signal Q: std_logic;      -- Global signal
end my_package;
. . .
use work.my_package.Q;    -- Make Q visible to the architecture
architecture arch1 of my_design is
begin
    . . .
end arch1;
```

In this example, the declaration for `Q` has been moved to an external package, and a **use** statement has been specified, making the contents of that package visible to the subsequent architecture.

Signal Initialization

In addition to creating one or more signals and assigning them a type, the signal declaration can also be used to assign an initial value to the signal, as shown below:

```
signal BusA: std_logic_vector(15 downto 0) := (others => 'Z');
```

This particular initialization uses a special kind of assignment, called an aggregate assignment, to assign all signals of the array `BusA` to an initial value of `'Z'`. (The `'Z'` value is defined in the IEEE 1164 standard as a high-impedance state.)

Initialization values are useful for simulation modeling, but they are not recommended for design descriptions that will be processed by synthesis tools. Synthesis tools typically ignore initialization values because they cannot assume that the target hardware will power up in a known state.

Using Signals

You will use signals in VHDL in two primary ways. First, if you want signals to carry information between different functional parts of your design, such as between two components, you will probably use them in a way similar to the following:

```
library ieee;
use ieee.std_logic_1164.all;
entity shiftcomp is port(Clk, Rst, Load: in std_logic;
                                   Init: in std_logic_vector(0 to 7);
                                   Test: in std_logic_vector(0 to 7);
                                   Limit: out std_logic);
end shiftcomp;
architecture structure of shiftcomp is
    component compare
        port(A, B: in std_logic_vector(0 to 7); EQ: out bit);
    end component;
    component shift
        port(Clk, Rst, Load: in std_logic;
               Data: in std_logic_vector(0 to 7);
               Q: out std_logic_vector(0 to 7));
    end component;
    signal Q: std_logic_vector(0 to 7);
begin
    COMP1: compare port map (Q, Test, Limit);
    SHIFT1: shift port map (Clk, Rst, Load, Init, Q);
end structure;
```

This example declares the signal `Q` within the architecture, then uses `Q` to connect the two components `COMP1` and `SHIFT1` together.

A second way of using signals is demonstrated by the following example in which signals are used within logic expressions and are assigned values directly (in this case within a process):

```
library ieee;
use ieee.std_logic_1164.all;
entity synch is
    port (Rst, Clk, Grant, nSelect: in std_logic;
             Request: out std_logic);
end synch;
architecture dataflow of synch is
    signal Q1, Q2, Q3, D3: std_logic;
begin
    dff: process (Rst, Clk)
    begin
        if Rst = '1' then
            Q1 <= '0';
```

```
            Q2 <= '0';
            Q3 <= '0';
        elsif Clk = '1' and Clk'event then
            Q1 <= Grant;
            Q2 <= Select;
            Q3 <= D3;
        end if;
    end process;
    D3 <= Q1 and Q3 or Q2;
    Request <= Q3;
end dataflow;
```

This example (which is a simplified synchronizer circuit) uses three signals, `Q1`, `Q2` and `Q3`, to represent register elements, with the signal `D3` being used as an intermediate signal representing a combinational logic function connecting the outputs of registers `Q1`, `Q2` and `Q3` to the input of `Q3`. The final assignment assigns the `Q3` register output to the `Request` output port. The register behavior is encapsulated into a process, `dff`, simplifying the concurrent statements that follow.

It is important to note that there is no significance to the order in which these concurrent statements occur. Like wires drawn between symbols on a schematic, signals assigned and used within a VHDL architecture are independent of each other and are not position dependent.

Using Variables

Variables are objects used to store intermediate values between sequential VHDL statements. Variables are only allowed in processes, procedures and functions, and they are always local to those functions.

Variables in VHDL are much like variables in a conventional software programming language. They immediately take on and store the value assigned to them, and they can be used to simplify a complex calculation or sequence of logical operations.

The following example is a simplified synchronizer circuit:

```
library ieee;
use ieee.std_logic_1164.all;
entity synch is
    port (Rst, Clk, Grant, nSelect: std_ulogic;
             Request: std_ulogic);
end synch;
architecture behavior of synch is
begin
    process(Rst, Clk)
        variable Q1, Q2, Q3: std_ulogic;
    begin
        if Rst = '1' then   -- Async reset
            Q1 := '0'; Q2 := '0'; Q3 := '0';
        elsif (Clk = '1' and Clk'event) then
            Q1 := Grant;
            Q2 := Select;
            Q3 := Q1 and Q3 or Q2;
        end if;
        Request <= Q3;
    end process;
end behavior;
```

In this example, a single process is used to describe the behavior of the three commonly-clocked register elements. The connections between the three registers are represented by variables that are local to the process, and the result (the output of register `Q3`) is then assigned to the output port `Request`. This design will probably not work as intended, because the registered behavior of `Q1` and `Q2` will be “short circuited” by the fact that variables were used.

Because variables do not always result in registers being generated within otherwise clocked processes, you must be very careful when using them.

Notes

The 1076-1993 language standard adds a new type of global variable that has visibility between different processes and subprograms. Global variables are not generally supported in synthesis tools.

Using Constants and Literals

Constants

Constants are objects that are assigned a value once, when declared, and do not change their value during simulation. Constants are useful for creating more readable design descriptions and they make it easier to change the design at a later time. The following code fragment provides a few examples of constant declarations:

```
architecture sample1 of consts is
    constant SRAM: bit_vector(15 downto 0) := X"F0F0";
    constant PORT: string  := "This is a string";
    constant error_flag: boolean := True;
begin
    . . .
    process( . . .)
        constant CountLimit: integer := 205;
    begin
        . . .
    end process;
end sample1;
```

Constant declarations can be located in any declaration area in your design description. If you want to create constants that are global to your design description, then you will place the constant declarations into external packages. If a constant will be used only within one segment of your design, you can place the constant declaration within the architecture, block, process or subprogram that requires it.

Literals

Explicit data values that are assigned to objects or used within expressions are called literals. Literals represent specific values, but they do not always have an explicit type. (For example, the literal '1' could represent either a bit data type or a character.) Literals do, however, fall into a few general categories:

Character literals

Character literals are 1-character ASCII values that are enclosed in single-quotes, such as the values '1', 'Z', '$' and ':'. The data type of the object being assigned one of these values (or the type implied by the expression in which the value is being used) will dictate whether a given character literal is valid. The literal value '$', for example, is a valid literal when assigned to a character type object, but it is not valid when assigned to a std_logic or bit data type.

String literals

String literals are collections of one or more ASCII characters enclosed in double-quote characters. String literals may contain any combination of ASCII characters, and they may be assigned to appropriately sized arrays of single-character data types (such as bit_vector or std_logic_vector) or to objects of the built-in type string.

Bit string literals

Bit string literals are special forms of string literals that are used to represent binary, octal, or hexadecimal numeric data values.

When representing a binary number, a bit string literal must be preceded by the special character '**B**', and it may contain only the characters '0' and '1'. For example, to represent a decimal value of 36 using a binary format bit string literal, you would write **B"100100"**.

When representing an octal number, the bit string literal must include only the characters '0' through '7', and it must be preceded by the special character '**O**', as in **O"446"**.

When representing a hexadecimal value, the bit string literal must be preceded by the special character '**X**', and it may include only the characters '0' through '9' and the characters 'A' through 'F', as in **X"B295"**. (Lower-case characters are also allowed, so 'a' through 'f' are also valid.)

The underscore character '**_**' may also be used in bit string literals as needed to improve readability. The following are some examples of bit string literals representing a variety of numeric values:

B"0111_1101" (decimal value 125)

O"654" (decimal value 428)

O"146_231" (decimal value 52,377)

X"C300" (decimal value 49,920)

Numeric literals

There are two basic forms of numeric literals in VHDL, **integer** literals and **real** literals.

Integer literals are entered as you would expect, as decimal numbers preceded by an optional negation character ('-'). The range of integers supported is dependent on your particular simulator or synthesis tool, but the VHDL standard does specify a minimum range of **-2,147,483,647** to **+2,147,483,647** (32 bits of precision, including the sign bit).

Real literals are entered using an extended form that requires a decimal point. For large numbers, scientific notation is also allowed using the character '**E**', where the number to the left of the '**E**' represents the mantissa of the real number, while the number to the right of the '**E**' represents the exponent. The following are some examples of real literals:

5.0

-12.9

1.6E10

1.2E-20

The minimum and maximum values of real numbers are defined by the simulation tool vendor, but they must be at least in the range of **-1.0E38** to **+1.0E38** (as defined by the standard). Numeric literals may not include commas, but they may include underscore characters ("_") to improve readability, as in:

1_276_801 `-- integer value 1,276,801`

Type checking is strict in VHDL, and this includes the use of numeric literals. It is not possible, for example, to assign an integer literal of 9 to an object of type real. (You must instead enter the value as 9.0.)

Based literals

Based literals are another form of integer or real values, but they are written in non-decimal form. To specify a based literal, you precede the literal with a base specification (such as **2**, **8**, or **16**) and enclose the non-decimal value with a pair of '#' characters as shown in the examples below:

2#10010001# (integer value 145)

16#FFCC# (integer value 65,484)

8#101# (integer value 65)

Physical literals

Physical literals are special types of literals used to represent physical quantities such as time, voltage, current, distance, etc. Physical literals include both a numeric part (expressed as an integer) and a unit specification. The following examples show how physical literals can be expressed:

300 ns (300 nanoseconds)

900 ps (900 picoseconds)

40 ma (40 milliamps)

Notes

In VHDL standard 1076-1987, bit string literals are only valid for the built-in type bit_vector. In 1076-193, bit string literals can be applied to any string type, including std_logic_vector.

Understanding Types and Subtypes

The VHDL 1076 specification describes four classes of data types:

- **Scalar types** represent a single numeric value or, in the case of enumerated types, an enumeration value. The standard types that fall into this class are integer, real (floating point), physical, and enumerated types. All of these basic types can be thought of as numeric values.
- **Composite types** represent a collection of values. There are two classes of composite types: arrays containing elements of the same type, and records containing elements of different types.
- **Access types** provide references to objects in much the same way that pointer types are used to reference data in software programming languages.
- **File types** reference objects (typically disk files) that contain a sequence of values.

Each type in VHDL has a defined set of values. For example, the value of an integer data type has a defined range of at least -2147483647 to +2147483647. In most cases you will only be interested in a subset of the possible values for a type, so VHDL provides the ability to specify a constraint whenever an object of a given type is declared. The following declaration creates an object of type integer that is constrained to the positive values of 0 to 255:

```
signal ShortInt: integer range 0 to 255;
```

VHDL also provides a feature called a subtype, allowing you to declare an alternate data type that is a constrained version of an existing type. For example, the declaration

```
subtype SHORT integer range 0 to 255;
```

creates an alternate scalar type with a constrained range. Because SHORT is a subtype of integer, it carries with it all operations available for the integer base type.

The four classes of data types are discussed in more detail below.

Scalar Types

Scalar types are those types that represent a single value, and are ordered in some way so that relational operations (such as greater than, less than, etc.) can be applied to them. These types include the obvious numeric types (integer and real) as well as less obvious enumerated types such as Boolean and Character.

Bit Type

The bit data type is the most fundamental representation of a wire in VHDL. The bit type has only two possible values, '0' and '1', that can be used to represent logical 0 and 1 values (respectively) in a digital system. The following example uses bit data types to describe the operation of a full adder:

```
entity fulladder is
    port (X: in bit;
          Y: in bit;
          Cin: in bit;
          Cout: out bit;
          Sum: out bit);
end fulladder;
architecture concurrent of fulladder is
begin
    Sum <= X xor Y xor Cin;
    Cout <= (X and Y) or (X and Cin) or (Y and Cin);
end concurrent;
```

The bit data type supports the following operations: and, or, nand, nor, xor, xnor, not, =, /=, <, <=, >, and >=.

Boolean Type

The Boolean type has two possible values, True and False. Like the bit data type, the Boolean type is defined as an enumerated type. The Boolean type does not have any implied width; it is simply the result of a logical test (such as a comparison operation or an if statement) or the expression of some logical state (such as in the assignment, `ErrorFlag <= True;`).

Integer Type

An integer type includes integer values in a specified range. The only predefined integer type is integer. Integer types have a minimum default range of -2147483647 to +2147483647, inclusive. However, you can restrict that value with a range constraint and/or declare a new integer subtype with a range constrained range, as in the following example:

```
subtype byteint integer range 0 to 255;
```

The predefined subtype natural restricts integers to the range of 0 to the specified (or default) upper range limit. The predefined subtype positive restricts integers to the range of 1 to the specified (or default) upper range limit.

An alternative to the integer data type is provided with IEEE Standard 1076.3. This standard defines the standard data types signed and unsigned, which are array types (based on the IEEE 1164 9-valued data types) that have properties of both array (composite) and numeric (scalar) data types. Like an array, you can perform shifting and masking operations on them and, like integers, you can perform arithmetic operations on them.

Real (Floating Point) Type

Floating point types are used to approximate real number values. The predefined floating point type provided in VHDL is called real. It has possible values in the range of at least -1.0E38 to +1.0E38.

The following declaration describes a signal of type real that has been initialized to a real value of 4589.3:

```
signal F0: real := 4589.3;
```

The real data type supports the following operations: =, /=, <, <=, >, >=, +, -, abs, +, -, *, and /.

Character Type

VHDL's character data type is similar to the character types you might be familiar with from software languages. Characters can be used to represent string data (such as you might use in a test bench), to display messages during simulation, or to represent actual data values in your design description. Unlike many software languages, character values in VHDL have no explicit value. This means that they cannot be simply mapped onto numeric data types or assigned directly to arrays of bits.

There is no specific numeric value associated with a given character literal in VHDL. (You cannot, for example, assign a character literal to an 8-bit array without providing a type conversion function that assigns unique array values – such as ASCII values – to the target array for each character value.)

The character data type is an enumerated type. However, there is an implied ordering (refer to the IEEE 1076-1993 specification for details).

Severity_Level Type

Severity_level is a special data type used in the **report** section of an **assert** statement. There are four possible values for an object of type severity_level: **note**, **warning**, **error** and **failure**. You might use these severity levels in your test bench, for example, to instruct your simulator to stop processing when an error (such as a test vector failure) is encountered during a simulation run. The following **assert** statement makes use of the FAILURE severity level to indicate that the simulator should halt processing if the specified condition evaluates false:

```
assert (error_flag = '1')
    report "There was an error; simulation has halted."
    severity FAILURE;
```

Time and other Physical Types

Time is a standard data type that falls into the category of physical types in VHDL. Physical types are those types that are used for measurement. They are distinguished by the fact that they have units of measurement, such as (in the case of time) seconds, nanoseconds, etc. Each unit in the physical type (with the exception of the base unit) is based on some multiple of the preceding unit. The definition for type time, for example, might have been written as follows (the actual definition is implementation-dependent):

```
type time isrange -2147483647 to 2147483647
       units
              fs;
              ps  = 1000 fs;
              ns  = 1000 ps;
              us  = 1000 ns;
```

```
        ms  = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr  = 60 min;
    end units;
```

Enumerated Types

Enumerated types are used to describe (internally) many of the standard VHDL data types. You can also use enumerated types to describe your own unique data types. For example, if you are describing a state machine, you might want to make use of an enumerated type to represent the various states of the machine, as in the following example:

```
architecture FSM of VCONTROL is
    type states is (StateLive,StateWait,StateSample,StateDisplay);
    signal current_state, next_state: states;
begin
   . . .
    -- State transitions:
    STTRANS: process(current_state,Mode,VS,ENDFR)
    begin
        case current_state is
            when StateLive =>     -- Display live video on the output
              . . .
           when StateWait =>     -- Wait for vertical sync
              . . .
            when StateSample =>  -- Sample one frame of video
            . . .
           when StateDisplay => -- Display the stored frame
              . . .
       end case;
    end process;
end FSM;
```

In this example (the control logic for a video frame grabber), an enumerated type (`states`) is defined in the architecture, and two signals (`current_state` and `next_state`) are declared for use in the subsequent state machine description. Using enumerated types in this way has two primary advantages: first, it is very easy to debug a design that uses enumerated types, because you can observe the symbolic type names during simulation; second, and perhaps more importantly for this state machine description, you can defer the actual encoding of the symbolic values until the time that you implement the design in hardware.

Synthesis tools generally recognize the use of enumerated types in this way and can perform special optimizations, assigning actual binary values to each symbolic name during synthesis. Synthesis tools generally also allow you to override the encoding of enumerated data types, so you have control over the encoding process.

Composite Types

Data Type	Values	Comment
bit_vector	"00100101", "10", etc.	Array of bits
string	"Simulation failed!", etc.	Array of characters
records	Any collection of values	User defined composite data type

Composite types are collections of one or more types of values. An array is a composite data type that contains items of the same type, either in a single dimension (such as a list of numbers or characters) or in multiple dimensions (such as a table of

values). Records, on the other hand, define collections of possibly unrelated data types. Records are useful when you need to represent complex data values that require multiple fields.

Array Types

An array is a collection of one or more values or objects of the same type. Arrays are indexed by a number that falls into the declared range of the array.

The following is an example of an array type declaration:

```
type MyArray isarray (15 downto 0) of std_ulogic;
```

This array type declaration specifies that the new type `MyArray` contains 16 elements, numbered downward from 15 to 0. Arrays can be given ranges that decrement from left to right (as shown) or increment (using the **`to`** keyword instead of **`downto`**). Index ranges do not have to begin or end at zero.

The index range (in this case 15 **`downto`** 0) is what is known as the index constraint. It specifies the legal bounds of the array. Any attempt to assign values to, or read values from, an element outside the range of the array will result in an error during analysis or execution of the VHDL design description.

The index constraint for an array can specify an unbounded array using the following array range syntax:

```
type UnboundedArray is array (natural range <>) of std_ulogic;
```

This array type declaration specifies that the array `UnboundedArray` will have a index constraint matching the range of integer subtype natural, which is defined as 0 to the highest possible integer value (at least 2,147,483,647).

An array type is uniquely identified by the types (and constraints) of its elements, the number of elements (its range), and the direction and order of its indices.

Arrays can have multiple indices, as in the following example:

```
type multi isarray(7 downto 0, 255 downto 0) of bit;
```

The following example (a parity generator) demonstrates how array elements can be accessed, in this case within a loop:

```
entity parity10 is
    port(D: in array(0 to 9) of bit;
         ODD: out bit);
    constant WIDTH: integer := 10;
end parity10;
architecture behavior of parity10 is
begin
    process(D)
        variable otmp: Boolean;
    begin
        otmp := false;
        for i in 0 to D'length - 1 loop
            if D(i) = '1' then
                otmp := not otmp;
            end if;
        end loop;
        if otmp then
            ODD <= '1';
        else
            ODD <= '0';
        end if;
    end process;
end behavior;
```

The direction of an array range has an impact on the index values for each element. For example, the following declarations:

```
signal A: bit_vector(0 to 3);
```

```
signal B: bit_vector(3 downto 0);
```

create two objects, `A` and `B`, that have the same width but different directions. The aggregate assignments:

```
A <= ('1','0','1','0');
B <= ('0','1','0','1');
```

are exactly identical to the assignments:

```
A(0) <= '1';
A(1) <= '0';
A(2) <= '1';
A(3) <= '0';
B(3) <= '0';
B(2) <= '1';
B(1) <= '0';
B(0) <= '1';
```

In this case, the arrays have the same contents when viewed in terms of their array indices. Assigning the value of `B` to `A`, as in:

```
A <= B;
```

which would be exactly equivalent to the assignments:

```
A(0) <= B(3);
A(1) <= B(2);
A(2) <= B(1);
A(3) <= B(0);
```

The leftmost element of array `A` has an index of 0, while the leftmost value of array `B` has an index value of 1.

Record Types

A record is a composite type that has a value corresponding to the composite value of its elements. The elements of a record may be of unrelated types. They may even be other composite types, including other records. You can access data in a record either by referring to the entire record (as when copying the contents of one record object to another record object), or individually by referring to a field name. The following example demonstrates how you might declare a record data type consisting of four elements:

```
type data_in_type is
    record
        ClkEnable: std_logic;
        Din: std_logic_vector(15 downto 0);
        Addr: integer range 0 to 255;
        CS: std_logic;
    end record;
```

The four names, `ClkEnable`, `Din`, `Addr` and `CS` are all field names of the record, representing data of specific types that can be stored as a part of the record. For example, an object of type `data_in_type` could be created and initialized with the following signal declaration:

```
signal test_record: data_in_type := ('0', "1001011011110011", 165, '1');
```

This initialization would be identical to the assignments:

```
test_record.ClkEnable <= '0';
test_record.Din <= "1001011011110011";
test_record.Addr <= 165;
test_record.CS <= '1';
```

Access and Incomplete Types

Access types and incomplete types are used to create data indirection in VHDL. You can think of access types as being analogous to pointers in software programming languages such as C or Pascal. Incomplete types are required to create recursive types such as linked lists, trees and stacks. Access and incomplete types can be useful for creating dynamic

representations of data (such as stacks), but they are not supported in today's synthesis tools. Refer to the IEEE VHDL Language Reference Manual for more information about these language features.

File Types

File types are very useful for writing test benches. File types differ in the VHDL 1076-1987 and 1076-1993 specifications. Discussions and examples of each are presented below.

VHDL 1076-1987 File Types

A file type is a special type of variable that contains sequential data. In the 1987 VHDL standard language, files are implicitly opened when they are declared, and it is not possible to explicitly close them. Objects of type file can be read from and written to using functions and procedures (**read**, **write**, and **endfile**) that are provided in the standard library. Additional functions and procedures for formatting of data read from files is provided in the Text I/O library, which is also part of the 1076 standard. The built-in functions available for reading and writing files in VHDL (the 1987 specification) are:

- **Read(f, object)** – Given a declared file and an object, read one field of data from the file into that object. When the read procedure is invoked, data is read from the file and the file is advanced to the start of the next data field in the file.
- **Write(f, object)** – Given a declared file and an object, write the data contained in the object to the file.
- **Endfile(f)** – Given a declared file, return a boolean true value if the current file marker is at the end of the file.

Files in VHDL are sequential; there is no provision for opening a file and reading from a random location in that file, or for writing specific locations in a file.

To use an object of type file, you must first declare the type of its contents, as shown below:

```
type file_of_characters isfile of character;
```

This declaration creates a new type, called `file_of_characters`, that consists of a sequence of character values. To use this file type, you would then create an object of type `file_of_characters`, as shown below:

```
file testfile: file_of_characters is in "TESTFILE.ASC";
```

This statement creates the object `testfile` and opens the indicated disk file. You can now use the built-in read procedure to access data in the file. A complete architecture that loops through a file and reads each character is shown below:

```
architecture sample87 of readfile is
begin
    Read_input: process
        type character_file is file of character;
        file cfile: character_file is in "TESTFILE.ASC";
        variable C: character;
        variable char_cnt: integer := 0;
    begin
        while not endfile(cfile) loop
            read (cfile, C) ; -- Get a character from cfile into C
            char_cnt = char_cnt + 1; -- Keep track of the number of
-- characters
        end loop;
    end process;
end sample87;
```

VHDL 1076-1993 File Types

In VHDL '93, file types and associated functions and procedures were modified to allow files to be opened and closed as needed. In the 1987 specification, there is no provision for closing a file, and problems can arise when it is necessary for two parts of the same design description to open the same file at different points, or when existing files must be both read from and written to (as when appending data). The built-in functions available for file operations in VHDL '93 are:

- **File_open(f, fname, fmode)** – Given a declared file object, file name (a string value) and a mode (either READ-MODE, WRITE_MODE, or APPEND_MODE), open the indicated file.
- **File_open(status, f, fname, fmode)** – Same as above, but return the status of the file open request in the first parameter, which is of type file_open_status. The status returned is either OPEN_OK (meaning the file was successfully opened),

STATUS_ERROR (meaning the file was not opened because there was already an open file associated with the file object), NAME_ERROR (meaning there was a system error related to the file name specified) or MODE_ERROR (meaning that the specified mode is not valid for the specified file).

- **File_close(f)** – Close the specified file.
- **Read(f, object)** – Given a declared file and an object, read one field of data from the file into that object. When the read procedure is invoked, data is read from the file and the file is advanced to the start of the next data field in the file.
- **Write(f, object)** – Given a declared file and an object, write the data contained in the object to the file.
- **Endfile(f)** – Given a declared file, return a boolean true value if the current file marker is at the end of the file.

A complete architecture that opens a file and loops through it, reading each character in the file, is shown below:

```
architecture sample93 of readfile is
begin
    Read_input: process
        type character_file is file of character;
        file cfile: character_file;
        variable C: character;
        variable char_cnt: integer := 0;
    begin
        file_open(cfile, "TESTFILE.ASC", READ_MODE);
        while not endfile(cfile) loop
            read (cfile, C) ; -- Get a character from cfile into C
            char_cnt = char_cnt + 1; -- Keep track of the number of
-- characters
        end loop;
        file_close(cfile);
    end process;
end sample93;
```

Notes

The IEEE 1164 specification describes an alternative to bit called std_ulogic. Std_ulogic has nine possible values, allowing the values and states of wires (such as high-impedance, unknown, etc.) to be more accurately described.

Floating point types have little use in synthesizable designs, as no synthesis tool available today will accept them.

Multidimensional arrays are not generally supported in synthesis tools. They can, however, be useful for describing test stimulus, memory elements, or other data that require a tabular form.

Records types are not generally synthesizable; however, they can be very useful when describing test stimulus.

Type Conversions and Type Marks

VHDL is a strongly typed language, meaning that you cannot simply assign a literal value or object of one type to an object of another type. To allow the transfer of data between objects of different types, VHDL includes type conversion features for types that are closely related. VHDL also allows type conversion functions to be written for types that are not closely related. In addition, VHDL includes type mark features to help specify (or qualify) the type of a literal value when the context or format of the literal makes its type ambiguous.

Explicit Type Conversions

The simplest type conversions are explicit type conversions, which are only allowed between closely related types. Two types are said to be closely related when they are either abstract numeric types (integers or floating points), or if they are array types of the same dimensions and share the same types (or the element types themselves are closely related) for all elements in the

array. In the case of two arrays, it is not necessary for the arrays to have the same direction. If two subtypes share the same base type, then no explicit type conversion is required.

The following example demonstrates implicit and explicit type conversions:

```
architecture example of typeconv is
    type array1 is array(0 to 7) of std_logic;
    type array2 is array(7 downto 0) of std_logic;
    subtype array3 is std_logic_vector(0 to 7);
    subtype array4 is std_logic_vector(7 downto 0);
    signal a1: array1;
    signal a2: array2;
    signal a3: array3;
    signal a4: array4;
begin
    a2 <= array2(a1);   -- explicit type conversion
    a4 <= a3;           -- no explicit type conversion needed
end example;
```

Type Conversion Functions

To convert data from one type to an unrelated type (such as from an integer type to an array type), you must make use of a type conversion function. Type conversion functions may be obtained from standard libraries (such as the IEEE 1164 library), from vendor-specific libraries (such as those supplied by synthesis tool vendors), or you can write you own type conversion functions.

A type conversion function is a function that accepts one argument of a specified type and returns the equivalent value in another type.

The following two functions are examples of type conversion functions that convert between integer and array (std_ulogic_vector) data types:

```
-------------------------------------------------------------------------
-- Convert a std_ulogic_vector to an unsigned integer
--
function to_uint (a: std_ulogic_vector) return integer is
    alias av: std_ulogic_vector (1 to a'length) is a;
    variable val: integer := 0;
    variable b: integer := 1;
begin
    for i in a'length downto 1 loop
        if (av(i) = '1') then    -- if LSB is '1',
            val := val + b;       -- add value for current bit position
        end if;
        b := b * 2;    -- Shift left 1 bit
    end loop;
    return val;
end to_uint;
-----------------------------------------------------------
-- Convert an integer to a std_ulogic_vector
--
function to_vector (size: integer; val: integer) return std_ulogic_vector is
    variable vec: std_ulogic_vector (1 to size);
    variable a: integer;
begin
    a := val;
```

```
    for i in size downto 1 loop
        if ((a mod 2) = 1) then
            vec(i) := '1';
        else
            vec(i) := '0';
        end if;
        a := a / 2;
    end loop;
    return vec;
end to_vector;
```

The following example (a loadable counter) demonstrates how these two functions could be used:

```
library ieee;
use ieee.std_logic_1164.all;
library types;        -- Type conversions have been compiled into library 'types'
use types.conversions.all;
entity count16 is
    port (Clk,Rst,Load: in std_ulogic;
          Data: in std_ulogic_vector(3 downto 0);
          Count: out std_ulogic_vector(3 downto 0));
end count16;
architecture count16a of count16 is
begin
    process(Rst,Clk)
        variable Q: integer range 0 to 15;
    begin
        if Rst = '1' then                -- Asynchronous reset
            Q := 0;
        elsif rising_edge(Clk) then
            if Load = '1' then
                Q := to_uint(Data);      -- Convert vector to integer
            elsif Q = 15 then
                Q := 0;
            else
                Q := Q + 1;
            end if;
        end if;
        Count <= to_vector(4,Q);         -- Convert integer to vector
                                         -- for use outside the process.
    end process;
end count16a;
```

In this example, the interface specified in the entity port list uses standard logic data types, including a std_ulogic_vector array data type for the counter output. Because there are no arithmetic operations defined for the std_ulogic_vector data type, it is necessary to introduce an intermediate integer variable and convert the `Data` input from a std_ulogic_vector type to an integer when assigning it to the intermediate variable, and to convert the intermediate variable back to a std_ulogic_vector array type when assigning it to the `Count` output.

Another common application of type conversion functions is the conversion of string data read from a file to array or record data types suitable for use as stimulus in a test bench. The following function accepts data in the form of a fixed-length string and converts it, character by character, into a record data type:

```
type test_record is record
    CE: std_ulogic;   -- Clock enable
    Set: std_ulogic;  -- Preset
    Din: std_ulogic;  -- Binary data input
    Doutput: std_ulogic_vector (15 downto 0);  -- Expected output
end record;
function str_to_record(s: string(18 downto 0)) return test_record is
    variable temp: test_record;
begin
    case s(18) is
        when '1' => temp.CE := '1';
        when '0' => temp.CE := '0';
        when others => temp.CE = 'X';
    end case;
    case s(17) is
        when '1' => temp.Set := '1';
        when '0' => temp.Set := '0';
        when others => temp.Set = 'X';
    end case;
    case s(16) is
        when '1' => temp.Din := '1';
        when '0' => temp.Din := '0';
        when others => temp.Din = 'X';
    end case;
    for i in 15 downto 0 loop
        case s(i) is
            when '1' => temp.Doutput := '1';
            when '0' => temp.Doutput := '0';
            when others => temp.Doutput = 'X';
        end case;
    end loop;
    return temp;
end str_to_record;
```

There are many applications of type conversion functions, and many possible ways to write them. If you are writing a synthesizable design description, you should (whenever possible) make use of type conversions that have been provided to you by your synthesis vendor, as type conversion functions can be difficult (in some cases impossible) for synthesis tools to handle.

Ambiguous Literal Types

Functions and procedures in VHDL are uniquely identified not only by their names, but also by the types of their arguments. This means that you can, for example, write two functions to perform similar tasks, but on different types of input data. The ability to overload functions and procedures can lead to ambiguities when functions are called, if the types of one or more arguments are not explicitly stated.

For example, consider two type conversion functions with the following interface declarations:

```
function to_integer (vec: bit_vector) return integer is
    . . .
end to_integer;
function to_integer (s: string) return integer is
    . . .
```

```
end to_integer;
```

If you were to write an assignment statement such as:

```
architecture ambiguous of my_entity is
    signal Int35: integer;
begin
    Int35 <= to_integer("00100011");    -- This will produce an error
    . . .
end ambiguous;
```

then the compiler would produce an error message because it would be unable to determine which of the two functions is appropriate – the literal `"00100011"` could be either a string or bit_vector data type.

To remove data type ambiguity in such cases, you have two options: you can either introduce an intermediate constant, signal or variable, as in:

```
architecture unambiguous1 of my_entity is
    constant Vec35: bit_vector := "00100011";
    signal Int35: integer;
begin
    Int35 <= to_integer(Vec35);
    . . .
end unambiguous1;
```

or introduce a type mark to qualify the argument, as in:

```
architecture unambiguous2 of my_entity is
    signal Int35: integer;
begin
    Int35 <= to_integer(bit_vector'"00100011");
    . . .
end unambiguous2;
```

Resolved and Unresolved Types

A signal requires resolution whenever it is simultaneously driven with more than one value. By default, data types (whether standard types or types you define) are unresolved, resulting in errors being generated when there are multiple values being driven onto signals of those types. These error messages may be the desired behavior, as it is usually a design error when such conditions occur. If you actually intend to drive a signal with multiple values (as in the case of a bus interface), then you will need to use a resolved data type.

Data types are resolved only when a resolution function has been included as a part of their definition. A resolution function is a function that specifies, for all possible combinations of one or more input values (expressed as an array of the data type being resolved), what the resulting (resolved) value will be.

The following sample package defines a resolved data type consisting of four possible values, `'0'`, `'1'`, `'X'` and `'Z'`. The resolution function covers all possible combinations of input values and specifies the resolved value corresponding to each combination:

```
package types is
    type xbit is ( '0',  -- Logical  0
                   '1',  -- Logical  1
                   'X',  -- Unknown
                   'Z' ); -- High Impedance
    -- unconstrained array is required for the resolution function...
    type xbit_vector is array ( natural range <> ) of xbit;

    -- resolution function...
    function resolve_xbit ( v : xbit_vector ) return xbit;
```

```
    -- resolved logic type...
    subtype xbit_resolved is resolve_xbit xbit;
end types;
package body types is
    -- Define resolutions as a table...
    type xbit_table is array(xbit, xbit) of xbit;
    constant resolution_table: xbit_table := (
    --         0    1    X    Z
            ( '0', 'X', 'X', '0' ), --  0
            ( 'X', '1', 'X', '1' ), --  1
            ( 'X', 'X', 'X', 'X' ), --  X
            ( '0', '1', 'X', 'Z' )  --  Z
     );

    function resolve_xbit ( v: xbit_vector ) return xbit is
        variable result: xbit;
    begin
        -- test for single driver
        if (v'length = 1) then
            result := v(v'low); -- Return the same value if only 1 value
        else
            result := 'Z';
            for i in v'range loop
                result := resolution_table(result, v(i));
            end loop;
        end if;
        return result;
    end resolve_xbit;
end types;
```

The resolution function is invoked automatically whenever a signal of the associated type is driven with one or more values. The array argument `v` represents all of the values being driven onto the signal at any given time.

With the types `xbit` and `xbit_resolved` defined in this way, the resolved data type `xbit_resolved` can be used for situations in which resolutions are required. The following example shows how the resolved type `xbit_resolved` could be used to describe the operation of a pair of three-state signals driving a common signal:

```
use work.types.all;
entity threestate is
    port (en1, en2: in xbit_resolved;
            A,B: in xbit_resolved;
            O: out xbit_resolved);
end threestate;
architecture sample of threestate is
    signal tmp1,tmp2: xbit_resolved;
begin
    tmp1 <= A when en1 else 'Z';
    tmp2 <= B when en2 else 'Z';
    O <= tmp1;
    O <= tmp2;
```

```
end sample;
```

In this example, the output `O` could be driven with various combinations of the values of `A` and `B` and the value `'Z'`, depending on the states of the two inputs `en1` and `en2`. The resolution function takes care of calculating the correct value for `O` for any of these combinations during simulation.

Notes

As a practical matter, you should never write an arbitrary-width type conversion function that you intend to use in a synthesizable design description. Instead, you should make use of type conversion functions provided by your synthesis vendor or use the 1076.3 signed or unsigned type.

VHDL Operators

Operator: abs

An absolute value operator which can be applied to any numeric type in an expression.

Example: Delta <= abs(A-B)

Operator: xnor

The logical “both or neither” (equality) operator which can be used in an expression. The expression “A xnor B” returns True only when (1) A is true and B is true, or (2) A is false and B is false.

Operator: and

The logical “and” operator which can be used in an expression. The expression “A and B” returns true only if both A and B are true.

Operator: mod

The modulus operator which can be applied to integer types. The result of the expression “A mod B” is an integer type and is defined to be the value such that:

(1) the sign of (A mod B) is the same as the sign of B, and

(2) abs (A mod B) < abs (B), and

(3) (A mod B) = (A * (B - N)) for some integer N.

Operator: nand

The logical “not and” operator which can be used in an expression. It produces the opposite of the logical “and” operator. The expression “A nand B” returns True only when (1) A is false, or (2) B is false, or (3) both A and B are false.

Operator: nor

The logical “not or” operator which can be used in an expression. It produces the opposite of the logical “or” operator. The expression “A nor B” returns True only when both A and B are false.

Operator: not

The logical “not” operator which can be used in an expression. The expression “not A” returns True if A is false and returns False if A is true.

Operator: or

The logical “or” operator which can be used in an expression. The expression “A or B” returns True if (1) A is true, or (2) B is true, or (3) both A and B are true.

Operator: rem

The remainder operator which can be applied to integer types. The result of the expression “A rem B” is an integer type and is defined to be the value such that:

(1) the sign of (A rem B) is the same as the sign of A, and

(2) abs (A rem B) < abs (B), and

(3) (A rem B) = (A - (A / B) * B).

Operator: rol

Rotate left operator.

Example: Sreg <= Sreg rol 2;

Operator: ror

Rotate right operator.

Example: Sreg <= Sreg ror 2;

Operator: sla

Shift left arithmetic operator.

Example: Addr <= Addr sla 8;

Operator: sll

Shift left logical operator.

Example: Addr <= Addr sll 8;

Operator: sra

Shift right arithmetic operator.

Example: Addr <= Addr sra 8;

Operator: srl

Shift right logical operator.

Example: Addr <= Addr srl 8;

Operator: xor

The logical “one or the other but not both” (inequality) operator which can be used in an expression. The expression “A xor B” returns True only when (1) A is true and B is false, or (2) A is false and B is true.

Operator: =

The equality operator which can be used in an expression on any type except file types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression “A = B” returns True only if A and B are equal.

Operator: /=

The inequality operator which can be used in an expression on any type except file types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression “A /= B” returns True only if A and B are not equal.

Operator: :=

The assignment operator for a variable. The expression “TEST_VAR := 1” means that the variable TEST_VAR is assigned the value 1.

Operator: <

The “less than” operator which can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression “A < B” returns True only if A is less than B.

Operator: <=

This symbol has two purposes. When used in an expression on scalar types and discrete array types, it is the “less than or equal to” operator. The resulting type of an expression using this operator in this context is Boolean (that is, True or False). In this context, the expression “A <= B” returns True only if A is less than or equal to B, for example:

```
LE := '1' when A <= B else '0';
```

In a signal assignment statement, the symbol “<=“ is the assignment operator. Thus, the expression

```
TEST_SIGNAL <= 5;
```

means that the signal TEST_SIGNAL is assigned the value 5.

Operator: >

The “greater than” operator can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (True or False). The expression “A > B” returns True only if A is greater than B.

Operator: >=

The “greater than or equal to” operator which can be used in an expression on scalar types and discrete array types. The resulting type of an expression using this operator is Boolean (that is, True or False). The expression “A >= B” returns True only if A is greater than or equal to B.

Operator: +

The addition operator. Both operands must be numeric and of the same type. The result is also of the same numeric type. Thus, if A = 2 and B = 3, the result of the expression “A + B” is 5.

This operator may also be used as a unary operator representing the identity function. Thus, the expression “+A” would be equal to A.

Operator: -

The subtraction operator. Both operands must be numeric and of the same type. The result is also of the same numeric type. Thus, if A = 5 and B = 3, the result of the expression “A - B” is 2.

This operator may also be used as a unary operator representing the negative function. Thus, the expression “-A” would be equal to the negative of A.

Operator: &

The concatenation operator. Each operand must be either an element type or a 1-dimensional array type. The result is a 1-dimensional array type.

Operator: *

The multiplication operator. Both operands must be of the same integer or floating point type.

The multiplication operator can also be used where one operand is of a physical type and the other is of an integer or real type. In these cases, the result is of a physical type.

Operator: /

The division operator. Both operands must be of the same integer or floating point type.

The division operator can also be used where a physical type is being divided by either an integer type or a real type. In these cases, the result is of a physical type. Also, a physical type can be divided by another physical type, in which case the result is an integer.

Operator: **

The exponentiation operator. The left operand must be of an integer type or a floating point type, and the right operand (the exponent) must be of an integer type. The result is of the same type as the left operand.

Understanding VHDL Operators

The following sections summarize the operators available in VHDL. As indicated, not all operators can be used for all data types, and the data type that results from an operation may differ from the type of object on which the operation is performed.

Logical Operators

The logical operators and, or, nand, nor, xor and xnor are used to describe Boolean logic operations, or perform bit-wise operations, on bits or arrays of bits.

Operator	Description	Operand Types	Result Types
and	AND	Any Bit or Boolean type	Same Type
or	OR	Any Bit or Boolean type	Same Type
nand	NOT AND	Any Bit or Boolean type	Same Type
nor	NOT OR	Any Bit or Boolean type	Same Type

Operator	Description	Operand Types	Result Types
xor	Exclusive OR	Any Bit or Boolean type	Same Type
xnor	Exclusive NOR	Any Bit or Boolean type	Same Type

Relational Operators

Relational operators are used to test the relative values of two scalar types. The result of a relational operation is always a Boolean true or false value.

Operator	Description	Operand Types	Result Type
=	Equality	Any type	Boolean
/=	Inequality	Any type	Boolean
<	Less than	Any scalar type or discrete array	Boolean
<=	Less than or equal	Any scalar type or discrete array	Boolean
>	Greater than	Any scalar type or discrete array	Boolean
>=	Greater than or equal	Any scalar type or discrete array	Boolean

Adding Operators

The adding operators can be used to describe arithmetic functions or, in the case of array types, concatenation operations.

Operator	Description	Operand Types	Result Type
+	Addition	Any numeric type	Same type
-	Subtraction	Any numeric type	Same type
&	Concatenation	Any numeric type	Same type
&	Concatenation	Any array or element type	Same array type

Multiplying Operators

The multiplying operators can be used to describe mathematical functions on numeric types.

Operator	Description	Operand Types	Result Type
*	Multiplication	Left: any integer or floating point type. Right: same type	Same as left
*	Multiplication	Left: any physical type. Right: integer or real type.	Same as left
*	Multiplication	Left: integer or real type. Right: any physical type.	Same as right
/	Division	Left: any integer or floating point type. Right: same type	Same as left
/	Division	Left: any integer or floating point type. Right: same type	Same as left
/	Division	Left: integer or real type. Right: any physical type.	Same as right
mod	Modulus	Any integer type	Same type

Operator	Description	Operand Types	Result Type
rem	Remainder	Any integer type	Same type

Sign Operators

Sign operators can be used to specify the sign (either positive or negative) of a numeric object or literal.

Operator	Description	Operand Types	Result Type
+	Identity	Any numeric type	Same type
-	Negation	Any numeric type	Same type

Miscellaneous Operators

The exponentiation and absolute value operators can be applied to numeric types, in which case they result in the same numeric type. The logical negation operator results in the same type (bit or Boolean), but with the reverse logical polarity. The shift operators provide bit-wise shift and rotate operations for arrays of type bit or Boolean.

Operator	Description	Operand Types	Result Type
**	Exponentiation	Left: any integer type Right: integer type	Same as left
**	Exponentiation	Left: any floating point type Right: integer type	Same as left
abs	Absolute value	Any numeric type	Same as left
not	Logical negation	Any Bit or Boolean type	Same as left
sll	Shift left logical	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left
srl	Shift right logical	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left
sla	Shift left arithmetic	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left
sra	Shift right arithmetic	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left
rol	Rotate left	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left
ror	Rotate right	Left: Any one-dimensional array of Bit or Boolean Right: integer type	Same as left

Notes

Operations defined for types Bit are also valid for type std_ulogic and std_logic.

Synthesis tools vary in their support for multiplying operators.

Understanding VHDL Attributes

Attributes are a feature of VHDL that allow you to extract additional information about an object (such as a signal, variable or type) that may not be directly related to the value that the object carries. Attributes also allow you to assign additional information (such as data related to synthesis) to objects in your design description.

There are two classes of attributes: those that are predefined as a part of the 1076 standard, and those that have been introduced outside of the standard, either by you or by your design tool supplier.

Predefined Attributes

The VHDL specification describes five fundamental kinds of attributes. These five kinds of attributes are categorized by the results that are returned when they are used. The possible results returned from these attributes are: a value, a function, a signal, a type or a range.

Predefined attributes are always applied to a prefix (such as a signal or variable name, or a type or subtype name), as in the statement:

```
wait until Clk = '1' and Clk'event and Clk'last_value = '0';
```

In this statement, the attributes `'event` and `'last_value` have been applied to the prefix `Clk`, which is a signal.

Some attributes also include parameters, so they are written in much the same way you would write a call to a function:

```
variable V: state_type := state_type'val(2);
```

In this case, the attribute `'val` has been applied to the prefix `state_type` (which is a type name) and has been given an attribute parameter, the integer value `2`.

Value kind attributes

Value Kind Attributes: 'Left, 'Right, 'High, 'Low, 'Length, 'Ascending

The value kind attributes that return an explicit value and are applied to a type or subtype include the following:

- **'Left**– this attribute returns the left-most element index (the bound) of a given type or subtype.

  Example:

  ```
  type bit_array isarray (1 to 5) of bit;
        variable L: integer := bit_array'left; -- L has a value of 1
  ```

- **'Right**– this attribute returns the right-most bound of a given type or subtype.

  Example:

  ```
  type bit_array is array (1 to 5) of bit;
        variable R: integer := bit_array'right; -- R has a value of 5
  ```

- **'High**– this attribute returns the upper bound of a given scalar type or subtype.

  Example:

  ```
  type bit_array is array(-15 to +15) of bit;
        variable H: integer := bit_array'high; -- H has a value of 15
  ```

- **'Low**– this attribute returns the upper bound of a given scalar type or subtype.

  Example:

  ```
  type bit_array is array(15 downto 0) of bit;
        variable L: integer := bit_array'low;  -- L has a value of 0
  ```

- **'Length**– this attribute returns the length (number of elements) of an array.

  Example:

  ```
  type bit_array is array (0 to 31) of bit;
  variable LEN: integer := bit_array'length -- LEN has a value of 32
  ```

- **'Ascending**– this attribute (VHDL '93 attribute) returns a boolean true value if the type or subtype is declared with an ascending range.

  Example:

  ```
  type asc_array is array (0 to 31) of bit;
  type desc_array is array (36 downto 4) of bit;
  variable A1: boolean := asc_array'ascending; -- A1 has a value of true
  variable A2: boolean := desc_array'ascending; -- A2 has a value of false
  ```

As you can see from the examples, value kind attributes (and all other predefined attributes) are identified by the ' (single quote) character. They are applied to type names, signals names and other identifiers, depending on the nature of the attribute. The value type attributes are used to determine the upper and lower (or left and right) bounds of a given type.

The following sample architecture uses the 'right and 'left attributes to determine the left- and right-most element indices of an array in order to describe a width-independent shift operation:

```
architecture behavior of shifter is
begin
    reg: process(Rst,Clk)
    begin
        if Rst = '1' then  -- Async reset
            Qreg := (others => '0');
        elsif rising_edge(Clk) then
            Qreg := Data(Data'left+1 to Data'right) & Data(Data'left);
        end if;
    end process;
end behavior;
```

The 'right, 'left, 'high and 'low attributes can be used to return non-numeric values. The following example demonstrates how you can use the 'left and 'right attributes to identify the first and last items in an enumerated type:

```
architecture example of enums is
    type state_type is (Init, Hold, Strobe, Read, Idle);
    signal L, R: state_type;
begin
    L <= state_type'left; -- L has the value of Init
    R <= state_type'right; -- R has the value of Idle
end example;
```

Value Kind Attributes: 'Structure, 'Behavior

There are two additional value kind attributes that can be used to determine information about blocks or attributes in a design. These attributes, 'structure and 'behavior, return true or false values depending on whether the block or architecture being referenced includes references to lower-level components.

- **'Structure**– this attribute returns a true value if the prefix (which must be an architecture name) includes references to lower-level components.
- **'Behavior** – this attribute returns a true value if the prefix (which must be an architecture name) does not include references to lower-level components.

Value Kind Attributes: 'Simple_name, 'Instance_name, 'Path_name

VHDL 1076-1993 added three attributes that can be used to determine the precise configuration of entities in a design description. These attributes return information about named entities, which are various items that become associated with identifiers, character literals or operator symbols as the result of a declaration.

- **'Simple_name** – this attribute returns a string value corresponding to the prefix, which must be a named entity.
- **'Instance_name** – this attribute returns a string value corresponding to the complete path (from the design hierarchy root) to the named entity specified in the prefix, including the names of all instantiated design entities. The string returned by this attribute has a fixed format that is defined in the IEEE VHDL Language Reference Manual.
- **'Path_name** – this attribute returns a string value corresponding to the complete path (from the design hierarchy root) to the named entity specified in the prefix. The string returned by this attribute has a fixed format that is defined in the IEEE VHDL Language Reference Manual.

Function kind attributes

Function Kind Attributes: 'Pos, 'Val, 'Succ, 'Pred, 'Leftof, 'Rightof

Attributes that return information about a given type, signal, or array value are called function kind attributes. VHDL defines the following function kind attributes that can be applied to types:

- **'Pos(value)**– this attribute returns the position number of a type value.

  Example:

  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable P: integer := state_type'pos(Read); -- P has the value of 3
  ```

- **'Val(value)**– this attribute returns the value corresponding to a position number of a type value.
  Example:
  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable V: state_type := state_type'val(2); -- V has the value of Strobe
  ```
- **'Succ(value)**– this attribute returns the value corresponding to position number after a given type value.
  Example:
  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable V: state_type := state_type'succ(Init); -- V has the value of Hold
  ```
- **'Pred(value)**– this attribute returns the value corresponding to position number preceding a given type value.
  Example:
  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable V: state_type := state_type'pred(Hold); -- V has the value of Init
  ```
- **'Leftof(value)**– this attribute returns the value corresponding to position number to the left of a given type value.
  Example:
  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable V: state_type := state_type'leftof(Idle); -- V has the value of Read
  ```
- **'Rightof(value)**– this attribute returns the value corresponding to position number to the right of a given type value.
  Example:
  ```
  type state_type is (Init, Hold, Strobe, Read, Idle);
  variable V: state_type := state_type'rightof(Read); -- V has the value of Idle
  ```

From the above descriptions, it might appear that the 'val and 'succ attributes are equivalent to the attributes 'leftof and 'rightof. One case where they would be different is the case where a subtype is defined that changes the ordering of the base type:

```
type state_type is (Init, Hold, Strobe, Read, Idle);
subtype reverse_state_type is state_type range Idle downto Init;
variable V1: reverse_state_type := reverse_state_type'leftof(Hold);
-- V1 has the value of Strobe
variable V2: reverse_state_type := reverse_state_type'pred(Hold);
-- V2 has the value of Init
```

Function Kind Array Attributes: 'Left, 'Right, 'High, 'Low

The function kind attributes that can be applied to array objects include:

- **'Left(value)**– this attribute returns the index value corresponding to the left bound of a given array range.
  Example:
  ```
  type bit_array is array (15 downto 0) of bit;
  variable I: integer := bit_array'left(bit_array'range); -- I has the value of 15
  ```
- **'Right(value)**– this attribute returns the index value corresponding to the right bound of a given array range.
  Example:
  ```
  type bit_array is array (15 downto 0) of bit;
  variable I: integer := bit_array'right(bit_array'range); -- I has the value of 0
  ```
- **'High(value)**– this attribute returns the index value corresponding to the upper-most bound of a given array range.
  Example:
  ```
  type bit_array is array (15 downto 0) of bit;
  variable I: integer := bit_array'high(bit_array'range); -- I has the value of 15
  ```
- **'Low(value)**– this attribute returns the index value corresponding to the lower bound of a given array range.
  Example:
  ```
  type bit_array is array (15 downto 0) of bit;
  variable I: integer := bit_array'low(bit_array'range); -- I has the value of 0
  ```

Function Kind Attributes: 'Event, 'Active, 'Last_event, 'Last_value, 'Last_active

Function kind attributes that return information about signals (such as whether that signal has changed its value or its previous value) include:

- **'Event**– this attribute returns a true value of the signal had an event (changed its value) in the current simulation delta cycle.

  Example:

```
process(Rst,Clk)
begin
    if Rst = '1' then
        Q <= '0';
    elsif Clk = '1' and Clk'event then -- Look for clock edge
        Q <= D;
    end if;
end process;
```

- **'Active**– this attribute returns true if any transaction (scheduled event) occurred on this signal in the current simulation delta cycle.

  Example:

```
process
variable A,E: boolean;
begin
    Q <= D after 10 ns;
    A := Q'active; -- A gets a value of True
    E := Q'event; -- E gets a value of False
    . . .
end process;
```

- **'Last_event**– this attribute returns the time elapsed since the previous event occurring on this signal.

  Example:

```
process
variable T: time;
begin
    Q <= D after 5 ns;
    wait 10 ns;
    T := Q'last_event;     -- T gets a value of 5 ns
    . . .
end process;
```

- **'Last_value**– this attribute returns the value of the signal prior to the last event.

  Example:

```
process
variable V: bit;
begin
    Q <= '1';
    wait 10 ns;
    Q <= '0';
    wait 10 ns;
    V := Q'last_value;      -- V gets a value of  '1'
    . . .
end process;
```

- **'Last_active**– this attribute returns the time elapsed since the last transaction (scheduled event) of the signal.

  Example:

```
process
variable T: time;
begin
    Q <= D after 30 ns;
    wait 10 ns;
    T := Q'last_active;   -- T gets a value of 10 ns
    . . .
end process;
```

Function Kind Attributes: 'Image, 'Value

The 'image and 'value attributes were added in the 1993 specification to simplify the reporting of information through Text I/O. These attributes both return string results corresponding to their parameter values.

- **'Image(expression)**– this attribute (VHDL '93 attribute) returns a string representation of the expression parameter, which must be of a type corresponding to the attribute prefix.

  Example:

```
assert (Data.Q = '1')
    report "Test failed on vector " & integer'image(vector_idx)
    severity Warning;
```

- **'Value(string)**– this attribute (VHDL '93 attribute) returns a value, of a type specified by the prefix, corresponding to the parameter string.

  Example:

```
write(a_outbuf,string'("Enter desired state (example: S1)"));
writeline(OUTPUT,a_outbuf);
readline(INPUT,a_inbuf);
read(a_inbuf,instate);   -- instate is a string type
next_state <= state_type'value(instate);
-- convert string to type state_type
write(a_outbuf,string'("Enter duration (example: 15)"));
writeline(OUTPUT,a_outbuf);
readline(INPUT,a_inbuf);
read(a_inbuf,induration);     -- induration is a string type
duration <= integer'value(induration);
-- convert string to type integer
```

Signal kind attributes

Signal Kind Attributes: 'Delayed, 'Stable, 'Quiet, 'Transaction

The signal kind attributes are attributes that, when invoked, create special signals that have values and types based on other signals. These special signals can then be used anywhere in the design description that a normally declared signal could be used. One example of where you might use such an attribute is to create a series of delayed clock signals that are all based on the waveform of a base clock signal.

Signal kind attributes include the following:

- **'Delayed(time)**– this attribute creates a delayed signal that is identical in waveform to the signal the attribute is applied to. (The time parameter is optional, and may be omitted.)

  Example:

```
process(Clk'delayed(hold))
-- Hold time check for input Data
  begin
```

```
    if Clk = '1' and Clk'stable(hold) then
        assert(Data'stable(hold))
            report "Data input failed hold time check!"
           severity Warning;
    end if;
end process;
```

- **'Stable (time)**– this attribute creates a signal of type boolean that becomes true when the signal is stable (has no event) for some given period of time.

  Example:

```
process
variable A: Boolean;
begin
    wait for 30 ns;
    Q <= D after 30 ns;
    wait 10 ns;
    A := Q'stable(20 ns);
       -- A gets a value of true (event has not
-- yet occurred)
    wait 30 ns;
    A := Q'stable(20 ns);
-- A gets a value of false (only 10 ns
-- since event)
    . . .
end process;
```

- **'Quiet (time)**– this attribute creates a signal of type boolean that becomes true when the signal has no transactions (scheduled events) or actual events for some given period of time.

  Example:

```
process
variable A: Boolean;
begin
    wait for 30 ns;
    Q <= D after 30 ns;
    wait 10 ns;
    A := Q'quiet(20 ns);
-- A gets a value of false (10 ns since
-- transaction)
    wait 40 ns;
    A := Q'quiet(20 ns);
-- A finally gets a value of true (20 ns
-- since event)
    . . .
end process;
```

- **'Transaction**– this attribute creates a signal of type bit that toggles its value whenever a transaction or actual event occurs on the signal the attribute is applied to.

Type kind attributes

Type Kind Attribute: 'Base

- **'Base**– this attribute returns the base type for a given type or subtype.

  Example:

```
type mlv7 is ('0','1','X','Z','H','L','W');
subtype mlv4 is mlv7 range '0' to 'Z';
variable V1: mlv4 := mlv4'right;
-- V1 has the value of 'Z'
variable V2: mlv7 := mlv4'base'right;
-- V2 has the value of 'W'
variable I1: integer := mlv4'width;
-- I1 has the value of 4
variable I2: integer := mlv4'base'width;
-- I2 has the value of 7
```

Range kind attributes

Range Kind Attributes: 'Range, 'Reverse_range

The range kind attributes return a special value that is a range, such as you might use in a declaration or looping scheme.

- **'Range**– this attribute returns the range value for a constrained array.

  Example:

```
function parity(D: std_logic_vector) return
std_logic is
    variable result: std_logic := '0';
begin
    for i in D'range loop
        result := result xor D(i);
    end loop;
    return result;
end parity;
```

- **'Reverse_range**– this attribute returns the reverse of the range value for a constrained array.

  Example:

```
STRIPX: for i in D'reverse_range loop
        if D(i) = 'X' then
            D(i) = '0';
        else
            exit; -- only strip the terminating Xs
        end if;
    end loop;
```

Custom Attributes

Custom attributes are those attributes that are not defined in the IEEE specifications, but that you (or your simulation or synthesis tool vendor) define for your own use. A good example is the attribute enum_encoding, which is provided by a number of synthesis tool vendors to allow specific binary encodings to be attached to objects of enumerated types.

An attribute such as enum_encoding is declared (again, either by you or by your design tool vendor) using the following method:

```
attribute enum_encoding: string;
```

This attribute could be written directly in your VHDL design description, or it could have been provided to you by the tool vendor in the form of a package. Once the attribute has been declared and given a name, it can be referenced as needed in the design description:

```
type statevalue is (INIT, IDLE, READ, WRITE, ERROR);
attribute enum_encoding of statevalue: type is "000 001 011 010 110";
```

When these declarations are processed by a synthesis tool that supports the enum_encoding attribute, information about the encoding of the type `statevalue` will be used by that tool. When the design is processed by design tools (such as simulators) that do not recognize the enum_encoding attribute, it will simply be ignored.

Custom attributes are a convenient "back door" feature of VHDL, and design tool vendors have created many such attributes to give you more control over the synthesis and simulation process. For detailed information about custom attributes, refer to your design tool documentation.

Notes

The function kind attributes 'active, 'last_event, 'last_value and 'last_active are not generally supported in synthesis tools. Only the 'event attribute should be used when describing synthesizable registered circuits. The 'active, 'last_event, 'last_value and 'last_active attributes should only be used to describe circuits for test purposes (such as for setup and hold checking). If they are encountered by a synthesis program, they will either be ignored, or the program will return an error and halt operation.

Using Standard Logic

This section takes a closer look at two important standards that augment Standard 1076, adding important capabilities for both simulation and synthesis. These two standards are IEEE Standards 1164 and 1076.3.

IEEE Standard 1164

IEEE Standard 1164 was released in the late 1980s, and helped to overcome an important limitation of VHDL and its various commercial implementations. These limitations were created by the fact that VHDL, while being rich in data types, did not include a standard type that would allow multiple values (high-impedance, unknown, etc.) to be represented for a wire. These metalogic values are important for accurate simulation, so VHDL simulation vendors were forced to invent their own proprietary data types using syntactically correct, but non-standard, enumerated types.

IEEE 1164 replaces these proprietary data types (which include systems having four, seven, or even thirteen unique values) with a standard data type having nine values, as shown below:

Value	Description
'U'	Uninitialized
'X'	Unknown
'0'	Logic 0 (driven)
'1'	Logic 1 (driven)
'Z'	High impedance
'W'	Weak 1
'L'	Logic 0 (read)
'H'	Logic 1 (read)
'-'	Don't-care

These nine values make it possible to accurately model the behavior of a digital circuit during simulation. For synthesis users, the standard has additional benefits for describing circuits that involve output enables, as well as for specifying don't-care logic that can be used to optimize the combinational logic requirements of a circuit.

Advantages of IEEE 1164

There are many compelling reasons to adopt IEEE Standard 1164 for all of your design efforts and to use it as a standard data type for all system interfaces. For simulation purposes, the standard logic data types allow you to apply values other than '0' or '1' as inputs and view the results. This capability could be used, for example, to verify that an input with an unknown (uninitialized or don't-care) value does not cause the circuit to behave in an unexpected manner. The resolved standard logic data types can be used to model the behavior of multiple drivers in your circuit. You might use these types to model, for example, the behavior of a three-state bus driver.

The most important reason to use standard logic data types is portability: if you will be interfacing to other components during simulation (such as those obtained from third party simulation model providers) or moving your design description between different simulation environments, then IEEE 1164 gives you a standard, portable style with which to describe your circuit.

Using the Standard Logic Package

To use the IEEE 1164 standard logic data types, you will need to add at least two statements to your VHDL source files. These statements (shown below) cause the IEEE 1164 standard library (named ieee) to be loaded and its contents (the std_logic_1164 package) made visible:

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

In most design descriptions, you will place these two statements at the top of your source file, and repeat them as needed prior to subsequent design units (entity and architecture pairs) in the file. If your source file includes more than one design unit, you

need to repeat the use statement just prior to each design unit in order to make the contents of the standard library visible to each design unit, as shown below:

```
library ieee;
use ieee.std_logic_1164.all;
package my_package is
    . . .
end my_package;
use ieee.std_logic_1164.all;
entity first_one is
    . . .
end first_one;
use ieee.std_logic_1164.all;
architecture structure of first_one is
    . . .
end structure;
use ieee.std_logic_1164.all;
entity second_one is
    . . .
end second_one;
```

Once you have included the ieee library and made the std_logic_1164 package visible in your design description, you can make use of the data types, operators and functions provided for you as a part of the standard.

There are two fundamental data types provided for you in the std_logic_1164 package. These data types, std_logic and std_ulogic, are enumerated types defined with nine symbolic (single character) values. The following definition of std_ulogic is taken directly from the IEEE 1164 standard:

```
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
                   );
```

The std_ulogic data type is an unresolved type, meaning that it is illegal for two values (such as '0' and '1', or '1' and 'Z') to be simultaneously driven onto a signal of type std_ulogic. If you are not describing a circuit that will be driving different values onto a wire (as you might in the case of a bus interface), then you might want to use the std_ulogic data type to help catch errors (such as incorrectly specified, overlapping combinational logic) in your design description. If you are describing a circuit that involves multiple values being driven onto a wire, then you will need to use the type std_logic. Std_logic is a resolved type based on std_ulogic. Resolved types are declared with resolution functions. Resolution functions define the resulting behavior when an object is driven with multiple values simultaneously.

When using either of these data types, you will use them as one-for-one replacements for the built-in type bit. The following example shows how you might use the std_logic data type to describe a simple NAND gate coupled to an output enable:

```
library ieee;
use ieee.std_logic_1164.all;
entity nandgate is
    port (A, B, OE: in std_logic; Y: out std_logic);
end nandgate;
```

```
architecture arch1 of nandgate is
    signal n: std_logic;
begin
    n <= not (A and B);
    Y <= n when OE = '0' else 'Z';
end arch1;
```

As written, it is not actually necessary for this circuit to be described using the resolved type std_logic for correct simulation. Operated as a stand-alone circuit, the output Y will never be driven with two different values. When connected through hierarchy into a larger circuit, however, it is highly likely that such a situation will occur, and std_logic will thus be required.

Std_logic_vector and Std_ulogic_vector

In addition to the single-bit data types std_logic and std_ulogic, IEEE Standard 1164 includes array types corresponding to each of these types. Both std_logic_vector and std_ulogic_vector are defined in the std_logic_1164 package as unbounded arrays similar to the built-in type bit_vector. In practice, you will probably use std_logic_vector or std_ulogic_vector with an explicit width, or you will use a subtype to create a new data type based on std_logic_vector or std_ulogic_vector of the width required. The following sample design description uses a subtype (defined in an external package) to create an 8-bit array based on std_ulogic_vector:

```
library ieee;
use ieee.std_logic_1164.all;
package my_types is
    subtype std_byte is std_ulogic_vector(7 downto 0);
end my_types;
use ieee.std_logic_1164.all;
entity shiftl is
    port (DataIN: in std_byte; DataOUT: out std_byte; Err: out std_ulogic);
end shiftl;
architecture arch1 of shiftl is
    signal n: std_logic;
begin
    DataOUT <= DataIN(DataIN'left - 1 downto 0) & '0';    -- Shift left one bit
    Err <= DataIN(DataIN'left);    -- Check for overflow
end arch1;
```

In this example (an 8-bit shifter), the subtype `std_byte` is defined in terms of std_ulogic_vector and can be used to replace `std_ulogic_vector(7 downto 0)` throughout the design description. The circuit is described in such a way that the width of the shifter is dependent only on the width of the type `std_byte`, so it is easy to modify the width of the circuit later.

Notes

VHDL has special visibility rules for architectures: it is not necessary to place a use statement prior to an architecture declaration if the corresponding entity declaration includes a use statement. In the first example above, the use statement appearing prior to the architecture `structure` is not actually needed and could be omitted.

Type Conversion and Standard Logic

If you need to describe operations such as counters that are not directly supported in the standard logic data types, you will almost certainly have to make use of type conversion functions to convert the standard logic data types at your system interfaces to types such as integers that support such operations.

Type conversion functions are functions that accept an object of one data type and return the equivalent data value represented as a different data type. Some type conversion functions are provided in the IEEE 1164 std_logic_1164 package (functions to convert between std_logic_vector and bit_vector, for example), but no functions are provided in that package to convert between standard logic data types and numeric data types such as integers.

Arithmetic circuits (such as adders and counters) are common elements of modern digital systems, and of design descriptions intended for synthesis. So what do you do if you want to use standard logic data types and describe arithmetic operations? There are actually a number of possible solutions to this problem.

The first solution is to write your own synthesizable type conversion functions, so that you can translate between standard logic values that you will use for your system interfaces (such as the ports for your entities) and the internal numeric type signals and variables you will need to describe your arithmetic function. This is actually a rather poor solution, as it can be quite difficult (perhaps impossible) to write a general-purpose (meaning width-independent) type conversion function that your synthesis tool can handle.

The second solution is to make use of custom type conversion functions or data types that have been provided by your synthesis vendor for use with their tool. An example of such a method (using the std_logic_arith package provided by Synopsys) is shown below:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity COUNT16 is
    port (Clk,Rst,Load: in std_logic;
          Data: in std_logic_vector(3 downto 0);
          Count: out std_logic_vector(3 downto 0)
    );
end COUNT16;
architecture COUNT16_A of COUNT16 is
begin
    process(Rst,Clk)
        -- The unsigned integer type is defined in synopsys.vhd...
        variable Q: unsigned (3 downto 0);
    begin
        if Rst = '1' then
            Q := "0000";
        elsif rising_edge(Clk) then
            if Load = '1' then
                for i in 3 downto 0 loop
                    Q(i) := Data(i);
                end loop;
            elsif Q = "1111" then
                Q := "0000";
            else
                Q := Q + "0001";
            end if;
        end if;
        Count <= conv_std_logic_vector(Q,Data'length);
    end process;
end COUNT16_A;
```

In this example, the `conv_std_logic_vector` function has been provided in the `std_logic_arith` package, which was supplied by a synthesis vendor (in this case, Synopsys).

Using synthesis tool-specific packages such as std_logic_arith can be quite convenient, but may result in a non-portable design description. (This is particularly true if you use tool-specific type conversion functions, which often have completely different naming conventions and function parameters, and are typically incompatible with synthesis tools other than those they were originally written for.)

The best solution is to use the IEEE 1076.3 standard numeric data types.

Standard Logic Data Types

This section describes in detail the contents of the IEEE 1164 Standard Logic package std_logic_1164. The std_logic_1164 package is compiled into a library named ieee, and includes the following data type and function definitions:

Type Std_ulogic

Type std_ulogic is intended to represent a single wire that can have various logical (and metalogical) values. Std_ulogic is the base type for other IEEE 1164 (and related) standard types, including std_logic, std_logic_vector, signed and unsigned. Std_ulogic has the following definition:

```
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
                   );
```

The std_ulogic data type is an enumerated type similar is usage to the bit data type provided in the standard (1076) library. Std_ulogic is an unresolved type.

Type Std_ulogic_vector

Type std_ulogic_vector is intended to represent a collection of wires, or a bus of arbitrary width. Std_ulogic_vector has the following definition:

```
type std_ulogic_vector is array ( natural range <> ) of std_ulogic;
```

Std_ulogic_vector is an unconstrained array of std_ulogic, and is analogous to the standard type bit_vector.

Type Std_logic

Type std_logic is a resolved type based on std_ulogic, and has the following definition:

```
subtype std_logic is resolved std_ulogic;
```

In the case of multiple drivers, the nine values of std_logic are resolved to values as indicated in the chart below.

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	1	0	0	0	0	X
1	U	X	X	X	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Type Std_logic_vector

Std_logic_vector is an unconstrained array of std_logic:

```
type std_logic_vector is array ( natural range <>) of std_logic;
```

Subtypes based on Std_ulogic

```
subtype X01 is resolved std_ulogic range 'X' to '1'; -- ('X','0','1')
```

```
subtype X01Z is resolved std_ulogic range 'X' to 'Z'; -- ('X','0','1','Z')
subtype UX01 is resolved std_ulogic range 'U' to '1'; -- ('U','X','0','1')
subtype UX01Z is resolved std_ulogic range 'U' to 'Z'; -- ('U','X','0','1','Z')
```

The X01, X01Z, UX01, and UX01Z subtypes are used within the std_logic_1164 package to simplify various operations on standard logic data, and may also be used when you have a need for 3-, 4-, or 5-valued logic systems.

Standard logic Operators

The following operators are defined for types std_ulogic, std_logic, std_ulogic_vector and std_logic_vector:

Logical Operators

```
function "and"  ( l : std_ulogic; r : std_ulogic ) return UX01;
function "nand" ( l : std_ulogic; r : std_ulogic ) return UX01;
function "or"   ( l : std_ulogic; r : std_ulogic ) return UX01;
function "nor"  ( l : std_ulogic; r : std_ulogic ) return UX01;
function "xor"  ( l : std_ulogic; r : std_ulogic ) return UX01;
function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
function "not"  ( l : std_ulogic ) return UX01;
```

Array Logical Operators

```
function "and"  ( l, r : std_logic_vector  ) return std_logic_vector;
function "and"  ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "nand" ( l, r : std_logic_vector  ) return std_logic_vector;
function "nand" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "or"   ( l, r : std_logic_vector  ) return std_logic_vector;
function "or"   ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "nor"  ( l, r : std_logic_vector  ) return std_logic_vector;
function "nor"  ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "xor"  ( l, r : std_logic_vector  ) return std_logic_vector;
function "xor"  ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
function "not"  ( l : std_logic_vector  ) return std_logic_vector;
function "not"  ( l : std_ulogic_vector ) return std_ulogic_vector;
```

Standard Logic Type Conversions

Type Conversions

The std_logic_1164 package includes a variety of type conversion functions to help convert data between 1076 standard data types (bit and bit_vector) and IEEE 1164 standard logic data types:

```
function To_bit ( s : std_ulogic;   xmap : bit := '0' ) return bit;
function To_bitvector ( s : std_logic_vector ; xmap : bit := '0' ) return bit_vector;
function To_bitvector ( s : std_ulogic_vector; xmap : bit := '0' ) return bit_vector;
function To_StdULogic ( b : bit ) return std_ulogic;
function To_StdLogicVector ( b : bit_vector ) return std_logic_vector;
function To_StdLogicVector ( s : std_ulogic_vector ) return std_logic_vector;
function To_StdULogicVector ( b : bit_vector ) return std_ulogic_vector;
function To_StdULogicVector ( s : std_logic_vector ) return std_ulogic_vector;
```

Strength Stripping Functions

The strength stripping functions convert the 9-valued types std_ulogic and std_logic to the 3-, 4-, and 5-valued types (X01, X01Z, UX01 and UX01Z), converting strength values ('H', 'L', and 'W') to their '0' and '1' equivalents.

```
function To_X01 ( s : std_logic_vector ) return std_logic_vector;
```

```
function To_X01 ( s : std_ulogic_vector ) return std_ulogic_vector;
function To_X01 ( s : std_ulogic ) return X01;
function To_X01 ( b : bit_vector ) return std_logic_vector;
function To_X01 ( b : bit_vector ) return std_ulogic_vector;
function To_X01 ( b : bit ) return X01;
function To_X01Z ( s : std_logic_vector ) return std_logic_vector;
function To_X01Z ( s : std_ulogic_vector ) return std_ulogic_vector;
function To_X01Z ( s : std_ulogic ) return X01Z;
function To_X01Z ( b : bit_vector ) return std_logic_vector;
function To_X01Z ( b : bit_vector ) return std_ulogic_vector;
function To_X01Z ( b : bit ) return X01Z;
function To_UX01 ( s : std_logic_vector ) return std_logic_vector;
function To_UX01 ( s : std_ulogic_vector ) return std_ulogic_vector;
function To_UX01 ( s : std_ulogic ) return UX01;
function To_UX01 ( b : bit_vector ) return std_logic_vector;
function To_UX01 ( b : bit_vector ) return std_ulogic_vector;
function To_UX01 ( b : bit ) return UX01;
```

Edge Detection and other Functions

Edge Detection Functions

The edge detection functions rising_edge() and falling_edge() provide a concise, portable way to describe the behavior of an edge-triggered device such as a flip-flop:

```
function rising_edge (signal s : std_ulogic) return boolean;
function falling_edge (signal s : std_ulogic) return boolean;
```

Miscellaneous Checking Functions

The following functions can be used to determine if an object or literal is a don't-care, which, for this purpose, is defined as any of the five values 'U', 'X', 'Z', 'W' or '-':

```
function Is_X ( s : std_ulogic_vector ) return  boolean;
function Is_X ( s : std_logic_vector ) return  boolean;
function Is_X ( s : std_ulogic ) return  boolean;
```

Standard 1076.3

IEEE Standard 1076.3 (the numeric standard) was developed to help synthesis tool users and vendors by providing standard, portable data types and operations for numeric data, and by providing more clearly defined meaning for the nine values of the IEEE 1164 std_ulogic and std_logic data types.

IEEE Standard 1076.3 defines the package numeric_std that allows the use of arithmetic operations on standard logic (std_logic and std_logic_vector) data types. (The 1076.3 standard also defines arithmetic forms of the bit and bit_vector data types in a package named numeric_bit, but this alternative package is not described here.)

The numeric_std package defines the numeric types signed and unsigned and corresponding arithmetic operations and functions based on the std_logic (resolved) data type. The package was designed for use with synthesis tools, and therefore includes additional functions (such as std_match) that simplify the use of don't-cares.

There are two numeric data types, unsigned and signed, declared in the numeric_std package, as shown below:

```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

Unsigned represents unsigned integer data in the form of an array of std_logic elements. Signed represents signed integer data.

Notes

In signed or unsigned arrays, the leftmost bit is treated as the most significant bit. Signed integers are represented in the signed array in two's complement form.

Using Numeric Data Types

There are many different applications of the IEEE 1076.3 numeric data types, operators and functions. The following example demonstrates how the unsigned type might be used to simplify the description of a counter:

```
-- COUNT16: 4-bit counter.
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity COUNT16 is
    port (Clk,Rst,Load: in std_logic;
          Data: in std_logic_vector (3 downto 0);
          Count: out std_logic_vector (3 downto 0)
    );
end COUNT16;
architecture COUNT16_A of COUNT16 is
    signal Q: unsigned (3 downto 0);
    constant MAXCOUNT: unsigned (3 downto 0) := "1111";
begin
    process(Rst,Clk)
    begin
        if Rst = '1' then
            Q <= (others => '0');
        elsif rising_edge(Clk) then
            if Load = '1' then
                Q <= UNSIGNED(Data); -- Type conversion
            elsif Q = MAXCOUNT then
                Q <= (others => '0');
            else
                Q <= Q + 1;
            end if;
        end if;
        Count <= STD_LOGIC_VECTOR(Q); -- Type conversion
    end process;
end COUNT16_A;
```

In this example, the type unsigned is used within the architecture to represent the counter data. The add operation ('+') is defined for type unsigned by the 1076-3 standard (in library numeric_std) so the counter can be easily described. Because the unsigned and std_logic_vector data types share the same element type (std_logic), conversion between these types is straightforward, as shown.

Numeric Standard Operators

Arithmetic Operators

```
function "abs" (ARG: signed) return signed;
function "-" (ARG: signed) return signed;
function "+" (L, R: unsigned) return unsigned;
function "+" (L, R: signed) return signed;
function "+" (L: unsigned; R: natural) return unsigned;
function "+" (L: natural; R: unsigned) return unsigned;
```

```
function "+" (L: integer; R: signed) return signed;
function "+" (L: signed; R: integer) return signed;
function "-" (L, R: unsigned) return unsigned;
function "-" (L, R: signed) return signed;
function "-" (L: unsigned;R: natural) return unsigned;
function "-" (L: natural; R: unsigned) return unsigned;
function "-" (L: signed; R: integer) return signed;
function "-" (L: integer; R: signed) return signed;
function "*" (L, R: unsigned) return unsigned;
function "*" (L, R: signed) return signed;
function "*" (L: unsigned; R: natural) return unsigned;
function "*" (L: natural; R: unsigned) return unsigned;
function "*" (L: signed; R: integer) return signed;
function "*" (L: integer; R: signed) return signed;
function "/" (L, R: unsigned) return unsigned;
function "/" (L, R: signed) return signed;
function "/" (L: unsigned; R: natural) return unsigned;
function "/" (L: natural; R: unsigned) return unsigned;
function "/" (L: signed; R: integer) return signed;
function "/" (L: integer; R: signed) return signed;
function "rem" (L, R: unsigned) return unsigned;
function "rem" (L, R: signed) return signed;
function "rem" (L: unsigned; R: natural) return unsigned;
function "rem" (L: natural; R: unsigned) return unsigned;
function "rem" (L: signed; R: integer) return signed;
function "rem" (L: integer; R: signed) return signed;
function "mod" (L, R: unsigned) return unsigned;
function "mod" (L, R: signed) return signed;
function "mod" (L: unsigned; R: natural) return unsigned;
function "mod" (L: natural; R: unsigned) return unsigned;
function "mod" (L: signed; R: integer) return signed;
function "mod" (L: integer; R: signed) return signed;
```

Numeric Logical Operators

```
function "not" (L: unsigned) return unsigned;
function "and" (L, R: unsigned) return unsigned;
function "or" (L, R: unsigned) return unsigned;
function "nand" (L, R: unsigned) return unsigned;
function "nor" (L, R: unsigned) return unsigned;
function "xor" (L, R: unsigned) return unsigned;
function "xnor" (L, R: unsigned) return unsigned;
function "not" (L: signed) return signed;
function "and" (L, R: signed) return signed;
function "or" (L, R: signed) return signed;
function "nand" (L, R: signed) return signed;
function "nor" (L, R: signed) return signed;
function "xor" (L, R: signed) return signed;
```

```
function "xnor" (L, R: signed) return signed;
```

Relational Operators

```
function ">" (L, R: unsigned) return boolean;
function ">" (L, R: signed) return boolean;
function ">" (L: natural; R: unsigned) return boolean;
function ">" (L: integer; R: signed) return boolean;
function ">" (L: unsigned; R: natural) return boolean;
function ">" (L: signed; R: integer) return boolean;
function "<" (L, R: unsigned) return boolean;
function "<" (L, R: signed) return boolean;
function "<" (L: natural; R: unsigned) return boolean;
function "<" (L: integer; R: signed) return boolean;
function "<" (L: unsigned; R: natural) return boolean;
function "<" (L: signed; R: integer) return boolean;
function "<=" (L, R: unsigned) return boolean;
function "<=" (L, R: signed) return boolean;
function "<=" (L: natural; R: unsigned) return boolean;
function "<=" (L: integer; R: signed) return boolean;
function "<=" (L: unsigned; R: natural) return boolean;
function "<=" (L: signed; R: integer) return boolean;
function ">=" (L, R: unsigned) return boolean;
function ">=" (L, R: signed) return boolean;
function ">=" (L: natural; R: unsigned) return boolean;
function ">=" (L: integer; R: signed) return boolean;
function ">=" (L: unsigned; R: natural) return boolean;
function ">=" (L: signed; R: integer) return boolean;
function "=" (L, R: unsigned) return boolean;
function "=" (L, R: signed) return boolean;
function "=" (L: natural; R: unsigned) return boolean;
function "=" (L: integer; R: signed) return boolean;
function "=" (L: unsigned; R: natural) return boolean;
function "=" (L: signed; R: integer) return boolean;
function "/=" (L, R: unsigned) return boolean;
function "/=" (L, R: signed) return boolean;
function "/=" (L: natural; R: unsigned) return boolean;
function "/=" (L: integer; R: signed) return boolean;
function "/=" (L: unsigned; R: natural) return boolean;
function "/=" (L: signed; R: integer) return boolean;
```

Shift and Rotate Functions

```
function shift_left (ARG: unsigned; COUNT: natural) return unsigned;
function shift_right (ARG: unsigned; COUNT: natural) return unsigned;
function shift_left (ARG: signed; COUNT: natural) return signed;
function shift_right (ARG: signed; COUNT: natural) return signed;
function rotate_left (ARG: unsigned; COUNT: natural) return unsigned;
function rotate_right (ARG: unsigned; COUNT: natural) return unsigned;
```

```
function rotate_left (ARG: signed; COUNT: natural) return signed;
function rotate_right (ARG: signed; COUNT: natural) return signed;
function "sll" (ARG: unsigned; COUNT: integer) return unsigned;
function "sll" (ARG: signed; COUNT: integer) return signed;
function "srl" (ARG: unsigned; COUNT: integer) return unsigned;
function "srl" (ARG: signed; COUNT: integer) return signed;
function "rol" (ARG: unsigned; COUNT: integer) return unsigned;
function "rol" (ARG: signed; COUNT: integer) return signed;
function "ror" (ARG: unsigned; COUNT: integer) return unsigned;
function "ror" (ARG: signed; COUNT: integer) return signed;
```

Numeric Resize Functions

The resize functions are used to convert a fixed-sized signed or unsigned array to a new (larger or smaller) size. If the resulting array is larger than the input array, the result is padded with '0's. In the case of a signed array, the sign bit is extended to the least significant bit.

```
function resize (ARG: signed; NEW_SIZE: natural) return signed;
function resize (ARG: unsigned; NEW_SIZE: natural) return unsigned;
```

Numeric Type Conversion Functions

The numeric type conversion functions are used to convert between integer data types and signed and unsigned data types.

```
function to_integer (ARG: unsigned) return natural;
function to_integer (ARG: signed) return integer;
function to_unsigned (ARG, SIZE: natural) return unsigned;
function to_signed (ARG: integer; SIZE: natural) return signed;
```

Numeric Matching Functions

The matching functions (std_match) are used to determine if two values of type std_logic are logically equivalent, taking into consideration the semantic values of the 'X' (uninitialized) and '-' (don't-care) literal values. The following table (derived from the match_table constant declaration found in the numeric_std package) defines the matching of all possible combinations of the std_logic enumerated values:

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	-
U	F	F	F	F	F	F	F	F	T
X	F	F	F	F	F	F	F	F	T
0	F	F	T	F	F	F	T	F	T
1	F	F	F	T	F	F	F	T	T
Z	F	F	F	F	F	F	F	F	T
W	F	F	F	F	F	F	F	F	T
L	F	F	T	F	F	F	T	F	T
H	F	F	F	T	F	F	F	T	T
-	T	T	T	T	T	T	T	T	T

```
function std_match (L, R: STD_ULOGIC) return boolean;
function std_match (L, R: unsigned) return boolean;
function std_match (L, R: signed) return boolean;
function std_match (L, R: std_logic_vector) return boolean;
function std_match (L, R: STD_ULOGIC_vector) return boolean;
```

Numeric Translation Functions

The numeric translation functions convert the nine std_logic values to numeric binary values ('0' or '1') for use in signed and unsigned arithmetic operations. These translation functions convert the values of 'L' and 'H' to '0' and '1', respectively. Any other values ('U', 'X', 'Z', '-', or 'W') result in a warning error (assertion) being generated.

```
function to_01 (S: unsigned; XMAP: std_logic := '0') return unsigned;
function to_01 (S: signed; XMAP: std_logic := '0') return signed;
```

Concurrent Statements

The following links are to more detailed topics within the area of concurrent statements. These include a look at the concept of concurrency as it is implemented in the VHDL language and in VHDL simulators; an exploration of some of the concurrent language features of VHDL in more detail and how combinational and registered logic can be described using these features; a look at how timing delays are annotated to concurrent assignments in VHDL, so you will have a better understanding of how simulation models are constructed.

The Concurrent Area

In VHDL, there is only one place where you will normally enter concurrent statements. This place, the concurrent area, is found between the **begin** and **end** statements of an architecture declaration. The following VHDL example shows where the concurrent area of a VHDL architecture is located:

```
architecture arch1 of my_circuit is
    signal Reset, DivClk: std_logic;
    constant MaxCount: std_logic_vector(15 downto 0) := "10001111";
    component count port (Clk, Rst: in std_logic;
                                        Q: out std_logic_vector(15 downto 0));
begin
    Reset <= '1' when Qout  = MaxCount else '0';
    CNT1: count port map(GClk, Reset, DivClk);
    Control: process(DivClk)
    begin
        . . .
    end process;
     . . .
end arch1;
```

All statements within the concurrent area are considered to be parallel in their execution and of equal priority and importance. Processes also obey this rule, executing in parallel with other assignments and processes appearing in the concurrent area.

There is no order dependency to statements in the concurrent area, so the following architecture declaration:

```
architecture arch1 of my_circuit is
    signal A, B, C: std_logic_vector(7 downto 0);
    constant Init: std_logic_vector(7 downto 0) := "01010101";
begin
    A <= B and C;
    B <= Init when Select = '1' else C;
    C <= A and B;
end arch1;
```

is exactly equivalent to:

```
architecture arch2 of my_circuit is
    signal A, B, C: std_logic_vector(7 downto 0);
    constant Init: std_logic_vector(7 downto 0) := "01010101";
begin
    C <= A and B;
    A <= B and C;
    B <= Init when Select = '1' else C;
end arch2;
```

The easiest way to understand this concept of concurrency is to think of concurrent VHDL statements as a kind of netlist, in which the various assignments being made are nothing more than connections between different types of objects.

If you think of the signals, constants, components, literals – and even processes – available in concurrent VHDL statements as distinct objects (such as you might find on a schematic or block diagram), and think of operations (such as and, not, and when-else) and assignments as logic gates and wiring specifications, respectively, then you will have no trouble understanding how VHDL's concurrent statements can be mapped to actual digital logic.

Concurrent Signal Assignments

The most common and simple concurrent statements you will write in VHDL are concurrent signal assignments. Concurrent signal assignments specify the logical relationships between different signals in a digital system.

If you have used PLD-oriented design languages (such as PALASM, ABEL, CUPL or Altera's AHDL), then concurrent signal assignments will be quite familiar to you. Just like the Boolean equations that you write using a PLD language, concurrent signal assignments in VHDL describe logic that is inherently parallel.

Because all signal assignments in your design description are concurrent, there is no relevance to the order in which the assignments are made within the concurrent area of the architecture.

In most cases, you will use concurrent signal assignments to describe either combinational logic (using logic expressions of arbitrary complexity), or you will use them to describe the connections between lower-level components. In some cases (though not typically for designs that will be synthesized) you will use concurrent signal assignments to describe registered logic as well.

The following example includes two simple concurrent signal assignments that represent NAND and NOR operations:

```
architecture arch3 of nand_circuit is
    signal A, B: std_logic;
    signal Y1, Y2: std_logic;
begin
    Y1 <= not (A and B);
    Y2 <= not (A or B);
end arch3;
```

In this example, there is no significance to the order in which the two assignments have been made. Also, keep in mind that the two signals being assigned (`Y1` and `Y2`) could just as easily have been ports of the entity rather than signals declared in the architecture. In all cases, signals declared locally (within an architecture, for example) can be used in exactly the same ways as can ports of the corresponding entity. The only difference between ports and locally-declared signals is that ports have a direction, or mode (**in**, **out** or **inout**), limiting whether they can have values assigned to them (in the case of **in**), or whether they can be read as inputs (in the case of **out**). If a port is declared as mode **out**, its value cannot be read. It can only be assigned a value. A port of mode **in** is the opposite; it can be read, but it cannot be assigned a value. A port of mode **inout** has both capabilities.

Conditional Signal Assignment

A conditional signal assignment is a special form of signal assignment, similar to the if-then-else statements found in software programming languages, that allows you to describe a sequence of related conditions under which one or more signals are assigned values. The following example (a simple multiplexer) demonstrates the basic form of a conditional assignment:

```
entity my_mux is
    port (Sel: in std_logic_vector (0 to 1);
              A, B, C, D: in std_logic_vector (0 to 3);
              Y: out std_logic_vector (0 to 3));
end my_mux;
architecture mux1 of my_mux is
begin
    Y <= A when Sel = "00" else
              B when Sel = "01" else
              C when Sel = "10" else
              D when others;
end mux1;
```

A conditional signal assignment consists of an assignment to one output (or a collection of outputs, such as an array of any type) and a series of conditional **when** statements, as shown. To ensure that all conditions are covered, you can use a terminating **when others** clause, as was done for the multiplexer description above.

The conditional signal assignment also provides a concise method of describing a list of conditions that have some priority. In the case of the multiplexer just described, there is no priority required or specified, since the four conditions (the possible values of the 2-bit input `Sel`) are all mutually exclusive. In some design descriptions, however, the priority implied by a series of **when-else** statements can cause some confusion (and additional logic being generated). For this reason, you might want to use a selected signal assignment as an alternative.

Notes

It is very important that all conditions in a conditional assignment are covered, as unwanted latches can be easily generated from synthesis for those conditions that are not covered. In the preceding multiplexer example, you might be tempted to replace the clause `D when others` with `D when Sel = "11"` (to improve readability). This would not be correct, however, because the data type being used in the design (std_logic_vector) has nine possible values for each bit. This means that there are actually 81 possible unique values that the input `Sel` could have at any given time, rather than four.

Selected Signal Assignment

A selected signal assignment is similar to a conditional signal assignment but differs in that the input conditions specified have no implied priority. The following is an example of a selected signal assignment:

```
entity my_mux is
    port (Sel: in std_logic_vector (0 to 1);
              A, B, C, D: in std_logic_vector (0 to 3);
              Y: out std_logic_vector (0 to 3));
end my_mux;
architecture mux1 of my_mux is
begin
    with Sel select
        Y <= A when "00",
             B when "01",
             C when "10",
             D when others;
end mux1;
```

In this simple multiplexer example, the selected signal assignment has exactly the same function as the conditional signal assignment presented earlier. This is not always the case, however, and you should carefully evaluate which type of assignment is most appropriate for a given application.

Conditional vs. Selected Signal Assignment

How to choose between a conditional assignment and a selected assignment? Consider this: a conditional assignment always enforces a priority on the conditions. For example, the conditional expression:

```
Q1 <= "01" when A = '1' else
          "10" when B = '1' else
          "11" when C = '1' else
          "00";
```

is identical to the selected assignment:

```
with std_logic_vector'(A,B,C) select
     Q2 <= "01" when "100",
                "01" when "101",
                "01" when "110",
                "01" when "111",
                "10" when "010",
```

```
                "10" when "011",
                "11" when "001",
                "00" when others;
```

Notice that input A takes priority. In the conditional assignment, that priority is implied by the ordering of the expressions. In the selected assignment, you must specify all possible conditions, so there can be no priority implied.

Why is this important for synthesis? Consider a circuit in which you know in advance that only one of the three inputs (A, B, or C) could ever be active at the same time. Or perhaps you don't care what the output of your circuit is under the condition where more than one input is active. In such cases, you can reduce the amount of logic required for your design by eliminating the priority implied by the conditional expression. You could instead write your description as:

```
with std_logic_vector'(A,B,C) select
         Q2 <= "01" when "100",
                    "10" when "010",
                    "11" when "001",
                    "00" when others;
```

This version of the description will, in all likelihood, require less logic to implement than the earlier version. This kind of optimization can save dramatic amounts of logic in larger designs.

In summary, while a conditional assignment may be more natural to write, a selected signal assignment may be preferable to avoid introducing additional, unwanted logic in your circuit.

Notes

You must include all possible conditions in a selected assignment. If not all conditions are easily specified, you can use the others clause as shown above to provide a default assignment.

The selection expressions may include ranges and multiple values. For example, you could specify ranges for a bit_vector selection expression as follows:

```
with Address select
      CS <= SRAM when 0x"0000" to 0x"7FFF",
            PORT when 0x"8000" to 0x"81FF",
            UART when 0x"8200" to 0x"83FF",
            PROM when others;
```

VHDL `93 adds the following feature to the selected signal assignment: You can use the keyword **unaffected** to specify that the output does not change under one or more conditions. For example, a multiplexer with two selector inputs could be described as:

```
with Sel select
        Y <= A when "00",
             B when "01",
             C when "10",
             unaffected when others;
```

The preceding multiplexer description may result in a latch being generated from synthesis. This is because the synthesized circuit will have to maintain the value of the output Y when the value of input Sel is "11".

Procedure Calls

Procedures may be called concurrently within an architecture. When procedures are called concurrently, they must appear as independent statements within the concurrent area of the architecture.

You can think of procedures in the same way you think of processes within an architecture: as independent sequential programs that execute whenever there is a change (an event) on any of their inputs. The advantage of a procedure over a process is that the body of the procedure (its sequential statements) can be kept elsewhere (in a package, for example) and used repeatedly throughout the design.

In the following example, the procedure dff is called within the concurrent area of the architecture:

```
architecture shift2 of shift is
    signal D,Qreg: std_logic_vector(0 to 7);
begin
    D <= Data when (Load = '1') else
                 Qreg(1 to 7) & Qreg(0);
    dff(Rst, Clk, D, Qreg);
    Q <= Qreg;
end shift2;
```

Generate Statements

Generate statements are provided as a convenient way to create multiple instances of concurrent statements, most typically component instantiation statements. There are two basic varieties of generate statements.

The for-generate Statement

The following example shows how you might use a **for-generate** statement to create four instances of a lower-level component (in this case a RAM block):

```
architecture generate_example of my_entity is
    component RAM16X1
        port(A0, A1, A2, A3, WE, D: in std_logic;
                O: out std_logic);
    end component;
begin
    . . .
    RAMGEN: for i in 0 to 3 generate
        RAM: RAM16X1 port map ( . . . );
    end generate;
    . . .
end generate_example;
```

When this generate statement is evaluated, the VHDL compiler will generate four unique instances of component `RAM16X1`. Each instance will have a unique name that is based on the instance label provided (in this case `RAM`) and the index value.

For-generate statements can be nested, so it is possible to generate multi-dimensional arrays of component instances or other concurrent statements.

The if-generate Statement

The **if-generate** statement is most useful when you need to conditionally generate a concurrent statement. A typical example of this occurs when you are generating a series of repetitive statements or components and need to supply different parameters, or generate different components, at the beginning or end of the series. The following example shows how a combination of a **for-generate** statement and three **if-generate** statements can be used to describe a 10-bit parity generator constructed of cascaded exclusive-OR gates:

```
library ieee;
use ieee.std_logic_1164.all;
entity parity10 is
    port(D: in std_logic_vector(0 to 9);
            ODD: out std_logic);
    constant width: integer := 10;
end parity10;
library gates;
use gates.all;
architecture structure of parity10 is
    component xor2
```

```
        port(A,B: in std_logic;
             Y: out std_logic);
    end component;
    signal p: std_logic_vector(0 to width - 2);
begin
    G: for I in 0 to (width - 2) generate
        G0: if I = 0 generate
             X0: xor2 port map(A => D(0), B => D(1), Y => p(0));
        end generate G0;
        G1: if I > 0 and I < (width - 2) generate
             X0: xor2 port map(A => p(i-1), B => D(i+1), Y => p(i));
        end generate G1;
        G2: if I = (width - 2) generate
             X0: xor2 port map(A => p(i-1), B => D(i+1), Y => ODD);
        end generate G2;
    end generate G;
end structure;
```

Concurrent Processes

Process statements contain sequential statements but are themselves concurrent statements within an architecture. In most VHDL design descriptions, there are multiple processes that execute concurrently during simulation and describe hardware that is inherently concurrent in its operation.

In the following example, two processes are used to describe a background clock (process CLOCK) and a sequence of stimulus inputs in a test bench:

```
architecture Stim1 of TEST_COUNT4EN is
component COUNT4EN
    port ( CLK,RESET,EN : in  std_logic;
           COUNT : out std_logic_vector(3 downto 0)
    );
end component;
constant CLK_CYCLE : Time := 20 ns;
signal CLK,INIT_RESET,EN : std_logic;
signal COUNT_OUT : std_logic_vector(3 downto 0);
begin
   U0: COUNT4EN port map ( CLK=>CLK,RESET=>INIT_RESET,
                           EN=>EN, COUNT=>COUNT_OUT);
   process begin
      CLK <= '1';
      wait for CLK_CYCLE/2;
      CLK <= '0';
      wait for CLK_CYCLE/2;
   end process;
   process begin
      INIT_RESET <= '0'; EN <= '1';
      wait for  CLK_CYCLE/3;
      INIT_RESET <= '1';
      wait for  CLK_CYCLE;
      INIT_RESET <= '0';
      wait for  CLK_CYCLE*10;
      EN <= '0';
```

```
      wait for  CLK_CYCLE*3;
      EN <= '1';
      wait;
   end process;
end Stim1;
```

The inter-relationships between multiple processes in a design description can be complex. For the purpose of understanding concurrency however, you must never assume that any process you write will be executed in simulation prior to any other process. This means that you cannot count on signals or shared variables being updated between two processes.

Component Instantiations

Component instantiations are statements that reference lower-level components in your design, in essence creating unique copies (or instances) of those components. A component instantiation statement is a concurrent statement, so there is no significance to the order in which components are referenced. You must, however, declare any components that you reference in either the declarative area of the architecture (before the **begin** statement) or in an external package that is visible to the architecture.

The following example demonstrates how component instantiations can be written. In this example, there are two lower-level components (`half_adder` and `full_adder`) that are referenced in component instantiations to create a total of four component instances. When simulated or synthesized, the four component instances (`A0`, `A1`, `A2` and `A3`) will be processed as four independent circuit elements. In this example, the two lower-level components `half_adder` and `full_adder` have been declared, right in the architecture. To make your design descriptions more concise, you may choose to place component declarations in separate packages instead.

```
library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
    port(A,B: in std_logic_vector(3 downto 0);
            S: out std_logic_vector(3 downto 0);
            Cout: out std_logic);
end adder4;
architecture structure of adder4 is
    component half_adder
         port (A, B: in std_logic; Sum, Carry: out std_logic);
    end component;
    component full_adder
         port (A, B, Cin: in std_logic; Sum, Carry: out std_logic);
    end component;
    signal C: std_logic_vector(0 to 2);
begin
    A0: half_adder port map(A(0), B(0),         S(0), C(0));
    A1: full_adder port map(A(1), B(1), C(0), S(1), C(1));
    A2: full_adder port map(A(2), B(2), C(1), S(2), C(2));
    A3: full_adder port map(A(3), B(3), C(2), S(3), Cout);
end structure;
```

Port and Generic Mapping

The mapping of ports in a component can be described in one of two ways. The simplest method is called positional association. Positional association simply maps signals in the architecture (the actuals) to corresponding ports in the lower-level entity declaration (the formals) by their position in the port list. When using positional association, you must provide exactly the same number and types of ports as are declared for the lower-level entity.

Positional association is quick and easy to use, and it is tempting to use this method almost exclusively. However, there are potential problems with positional association. The most troublesome problem is the lack of error checking. It is quite easy, for

example, to inadvertently reverse the order of two ports in the list. The result is a circuit that may compile with no errors, but fail to simulate properly. After the first few times you accidentally swap the reset and clock lines to one of your lower-level components, you may decide that it is worth the extra typing to provide a more complete specification of your port mappings. The method you will use in this case is called named association.

Named association is an alternate form of port mapping that includes both the actual and formal port names in the port map of a component instantiation. (Named association can also be used in other places, such as in the parameter lists for generics and subprograms.)

An example using named association, for a 4-bit adder, is as follows:

```
architecture structure of adder4 is
  component half_adder
       port (A, B: in std_logic; Sum, Carry: out std_logic);
  end component;
  component full_adder
       port (A, B, Cin: in std_logic; Sum, Carry: out std_logic);
  end component;
  signal C: std_logic_vector(0 to 2);
begin
  A0: half_adder port map(A => A(0), B => B(0), Sum => S(0), Carry => C(0));
  A1: full_adder port map(A => A(1), B => B(1), Cin => C(0), Sum => S(1), Carry => C(1));
  A2: full_adder port map(A => A(2), B => B(2), Cin => C(1), Sum => S(2), Carry => C(2));
  A3: full_adder port map(A => A(3), B => B(3), Cin => C(2), Sum => S(3), Carry => Cout);
end structure;
```

When you specify port mappings using named association, lower-level names (the formal ports of the component) are written on the left side of the => operator, while the top-level names (the actuals) are written on the right.

The benefits of named association go beyond simple error checking. Because named association removes the requirement for any particular order of the ports, you can enter them in whatever order you want. You can even leave one or more ports unconnected if you have provided default values in the lower-level component specification.

Because named association is so much more flexible (and less error prone) than positional association, it is strongly recommend that you get in the habit of typing in the few extra characters required to use named association.

Generic Mapping

If the lower-level entity being referenced includes generics, you can specify a generic map in addition to the port map to pass actual generic parameters to the lower-level entity:

```
architecture timing of adder4 is
  component half_adder
       port (A, B: in std_logic; Sum, Carry: out std_logic);
  end component;
  component full_adder
       port (A, B, Cin: in std_logic; Sum, Carry: out std_logic);
  end component;
  signal C: std_logic_vector(0 to 2);
begin
  A0: half_adder
        generic map(tRise => 1 ns, tFall => 1 ns);
        port map(A => A(0), B => B(0), Sum => S(0), Carry => C(0));
  A1: full_adder
        generic map(tRise => 1 ns, tFall => 1 ns);
        port map(A => A(1), B => B(1), Cin => C(0), Sum => S(1), Carry => C(1));
  A2: full_adder
```

```
          generic map(tRise => 1 ns, tFall => 1 ns);
          port map(A => A(2), B => B(2), Cin => C(1), Sum => S(2), Carry => C(2));
  A3: full_adder
          generic map(tRise => 1 ns, tFall => 1 ns);
          port map(A => A(3), B => B(3), Cin => C(2), Sum => S(3), Carry => Cout);
end timing;
```

Just as with port maps, generic maps can be written using either positional or named association.

Notes

The rules of VHDL allow you to mix positional and named association in the same port, generic or parameter list. Doing so has little or no benefit however, and it may confuse other potential users of your design description.

Delay Specifications

VHDL allows signal assignments to include delay specifications, in the form of an **after** clause. The **after** clause allows you to model the behavior of gate and wire delays in a circuit. This is very useful if you are developing simulation models or if you want to include estimated delays in your synthesizable design description. The following are two examples of delay specifications associated with signal assignments:

```
Y1 <= not (A and B) after 7 ns;
Y2 <= not (A and B) transport after 7 ns;
```

These two assignments demonstrate the two fundamental types of delay specifications available in VHDL: inertial and transport.

Inertial delay is intended to model the delay through a gate, in which there is some minimum pulse length that must be maintained before an event is propagated.

Transport delay, on the other hand, models the delay on a wire, so pulses of any width are propagated.

For design descriptions intended for synthesis, you will probably not bother to use delay specifications such as these. A circuit produced as a result of synthesis is unlikely to have timing characteristics that can be accurately predicted (or specified) up front. In fact, all synthesis tools in use as of this writing ignore the **after** clause completely. (If you have a general idea of the timing characteristics of your synthesis target – be it an FPGA chip or a high-complexity ASIC – you can use delay specifications to improve the accuracy of your initial simulation. Just be aware that anything you annotate prior to synthesis will be little more than a guess.)

When you are writing test benches, you will also probably not use **after** clauses to specify timing of input events. Instead, you will likely rely on a series of **wait** statements entered within a process to accurately specify your test stimulus.

The IEEE 1076-1993 standard added an additional feature called a **reject** time. For inertial delays (the default delay type if transport is not specified), a minimum inertial pulse time can be specified as follows:

```
Y1 <= reject 3 ns not (A and B) after 7 ns;
```

In this example, any event greater than `3 ns` in width will be propagated to the output. In the absence of a specified reject time, the specified delay time (in this case `7 ns`) is used as the default reject time.

Signal Drivers

VHDL includes an elaborate set of rules and language features to resolve situations in which the same signal is driven to multiple values simultaneously. These situations can be caused unintentionally (by an incomplete or incorrect design specification), or they may represent a desired circuit condition, such as a three-state driver connected to a bus, or they may represent a simple output enable used in a loadable bi-directional register.

To handle such situations, VHDL introduces the concept of a signal driver. A signal driver is a conceptual circuit that is created for every signal assignment in your circuit. By default, this conceptual circuit provides a comparison function to ensure that only one driver is active at any given time. The following architecture demonstrates a circuit description that does not meet this requirement:

```
architecture arch4 of nand_circuit is
    signal Sel, A, B: std_logic;
    signal Y: std_logic;
begin
    Y <= not (A and B) and Sel;
    Y <= not (A or B) and not Sel;
end arch4;
```

The intent of this circuit is to provide a single output (`Y`) that functions either as a NAND gate or as a NOR gate based on the value of `Sel`. Unfortunately, each of the two assignments results in a driver being created, resulting in a multiple-driver situation.

The solution to this, of course, is to completely specify the output `Y` using only one signal assignment, as in the following:

```
architecture arch4 of nand_circuit is
    signal Sel, A, B: std_logic;
    signal Y,Y1,Y2: std_logic;
begin
    Y1 <= not (A and B);
    Y2 <= not (A or B);
    Y <= Y1 and Sel or Y2 and not Sel;
end arch4;
```

In this example, two intermediate signals have been introduced (`Y1` and `Y2`) and the output `Y` has been more completely described as a function of these two values. Another method might be to simply combine the three assignments into a larger combinational expression

```
(not (A and B) and Sel or not (A or B) and not Sel
```

or to use a more concise statement such as a conditional assignment:

```
architecture arch5 of nand_circuit is
    signal Sel, A, B: std_logic;
    signal Y,Y1,Y2: std_logic;
begin
    Y <= not (A and B) when Sel = '1' else
            not (A or B);
end arch5;
```

Of course, these simple examples only show how you might resolve multiple driver situations that have been inadvertently created. You will find that VHDL's signal driver rules can actually help to detect and correct errors in your design that might otherwise go unnoticed. For situations that are intentional, however, how can you get around the rules? The answer is a feature of VHDL called a resolution function. A resolution function is a special type of function that you (or someone else, such as the IEEE committee that defined the resolved type std_logic) can write to resolve multiple-driver situations for a specific type. For example, the resolution function for a four-value data type consisting of the values '1', '0', 'X' (unknown) and 'Z' (high impedance) might have a resolution function that specifies:

- that simultaneous values of '1' and '0' appearing on a signal's drivers will result in an 'X' value,
- that both 'Z' and 'X' can be over-ridden by values of '1' or '0', and
- that 'Z' is over-ridden by 'X'.

For most design descriptions and test benches, you will not need to use resolved types such as these. (In many synthesis tools, resolution functions are not supported anyway. They serve only to let the compiler know whether multiple drivers are allowed for an output.)

Sequential Statements

Sequential VHDL statements allow you to describe the operation, or behavior, of your circuit as a sequence of related events. Such descriptions are natural for order-dependent circuits such as state machines and for complex combinational logic that involves some priority of operations. The use of sequential statements to describe combinational logic implies that the use of the term sequential in VHDL is somewhat different from the term as it is often used to describe digital logic. Specifically, sequential statements written in VHDL do not necessarily represent sequential digital logic circuits. It is possible (and quite common) to write sequential VHDL statements, using processes and subprograms, to describe what is essentially combinational logic.

Sequential statements are found within processes, functions, and procedures. Sequential statements differ from concurrent statements in that they have order dependency. This order dependency may or may not imply a sequential circuit (one involving memory elements).

The Process Statement

VHDL's **process** statement is the primary way you will enter sequential statements. A **process** statement, including all declarations and sequential statements within it, is actually considered to be a single concurrent statement within a VHDL architecture. This means that you can write as many processes and other concurrent statements as are necessary to describe your design, without worrying about the order in which the simulator will process each concurrent statement.

Anatomy of a Process

The general form of a process statement is:

```
process_name: process (sensitivity_list)
    declarations
begin
    sequential_statements
end process;
```

The easiest way to think of a VHDL process is to relate it to event-driven software – like a program that executes (in simulation) any time there is an event on one of its inputs (as specified in the sensitivity list). A process describes the sequential execution of statements that are dependent on one or more events having occurred. A flip-flop is a perfect example of such a situation. It remains idle, not changing state, until there is a significant event (either a rising edge on the clock input or an asynchronous reset event) that causes it to operate and potentially change its state.

Although there is a definite order of operations within a process (from top to bottom), you can think of a process as executing in zero time. This means that a process can be used to describe circuits functionally, without regard to their actual timing, and multiple processes can be "executed" in parallel with little or no concern for which processes complete their operations first.

A process can be thought of as a single concurrent statement written within a VHDL architecture, extending from the **process** keyword (or from the optional process name that precedes it) to the terminating **end process** keyword pair and semicolon.

The process name (process_name) appearing before the **process** keyword is optional and can be used to: (1) identify specific processes that are executing during simulation, and (2) more clearly distinguish elements such as local variables that may have common names in different processes.

Immediately following the **process** statement is an optional list of signals enclosed by parentheses. This list of signals, called the sensitivity list, specifies the conditions under which the process is to begin executing. When a sensitivity list is associated with a process, any change in the value of any input in the list will result in immediate execution of the process.

In the absence of a sensitivity list, the process will execute continuously, but must be provided with at least one **wait** statement to cause the process to suspend periodically.

The order in which statements are written in a process is significant. You can think of a process as a kind of software program that is executed sequentially, from top to bottom, each time it is invoked during simulation. Consider, for example, the following process describing the operation of a counter:

```
process(Clk)
begin
    if Clk = '1' and Clk'event then
        if Load = '1' then
```

```
            Q <= Data_in;
        else
            Q <= Q + 1;
        end if;
    end if;
end process;
```

When this process is executed, the statements appearing between the **begin** and **end** process statements are executed in sequence. In this example, the first statement is an if test that will determine if there was a rising edge on the `Clk` clock input. A second, nested if test determines if the counter should be loaded with `Data_in` or incremented, depending on the value of the `Load` input.

Processes with Sensitivity Lists

A process with a sensitivity list is executed during simulation whenever an event occurs on any of the signals in the sensitivity list. An event is defined as any change in value of a signal, such as when a signal of type Boolean changes from True to False, or when the value of an integer type signal is incremented or otherwise modified.

Processes that include sensitivity lists are most often used to describe the behavior of circuits that respond to external stimuli. These circuits, which may be either combinational, sequential (registered), or a combination of the two, are normally connected with other sub-circuits or interfaces, via signals, to form a larger system. In a typical circuit application, such a process will include in its sensitivity list all inputs that have asynchronous behavior. These inputs may include clocks, reset signals, or inputs to blocks of combinational logic.

The following is an example of a process that includes a sensitivity list. This process describes the operation of a clocked shift register with an asynchronous reset; note the use of the `` `event `` signal attribute to determine which of the two signals (`Clk` and `Rst`) had an event:

```
process(Rst, Clk)
begin
    if Rst = '1' then
        Q <= "00000000";
    elsif Clk = '1' and Clk'event then
        if Load = '1' then
            Q <= Data_in;
        else
            Q <= Q(1 to 7) & Q(0);
        end if;
    end if;
end process;
```

During simulation, whenever there is an event on either `Rst` or `Clk`, this **process** statement will execute from the **begin** statement to the **end process** statement pair. If the `Rst` input is `'1'` (regardless of whether the event that triggered the process execution was `Rst` or `Clk`), then the output `Q` is set to a reset value of `"00000000"`. If the value of `Rst` is not `'1'`, then the `Clk` input is checked to determine if it has a value of `'1'` and had an event. This checking for both a value and an event is a common (and synthesizable) way of detecting transitions, or edges, on signals such as clocks.

After all of the statements in the process have been analyzed and executed, the process is suspended until a new event occurs on one of the process's sensitivity list entries.

For design descriptions intended for input to synthesis software, you should follow the above example and write **process** statements that include sensitivity lists, as this is the most widely used synthesis convention for registers.

Processes without Sensitivity Lists

A process that does not include a sensitivity list executes somewhat differently than a process with a sensitivity list. Rather than executing from the **begin** statement at the top of the process to the **end process** statement, a process with no sensitivity list executes from the beginning of the process to the first occurrence of a **wait** statement, then suspends until the condition specified in the **wait** statement is satisfied. If the process only includes a single **wait** statement, the process re-activates when the condition is satisfied and continues to the **end process** statement, then begins executing again from the beginning. If there are multiple **wait** statements in the process, the process executes only until the next **wait** statement is encountered.

The following example demonstrates how this works, using a simplified Manchester encoder as an example:

```
process
begin
    wait until Clk = '1' and Clk'event;
    M_out <= data_in;
    wait until Clk = '1' and Clk'event;
    M_out <= not data_in;
end process;
```

This process will suspend its execution at two points. The first **wait until** statement suspends the process until there is a rising edge on the clock (a transition to a value of `'1'`). When this rising edge condition has been met, the process continues execution by assigning the value of `data_in` to `M_out`. Next, the second **wait until** statement suspends the process until another rising edge has been detected on `Clk`. When this condition has been met, the process continues and assigns the inverted value of `data_in` to `M_out`. The process does not suspend at the **end process** statement, but instead loops back to the beginning and immediately starts processing over again.

The use of multiple **wait** statements within a process makes it possible to describe very complex multiple-clock circuits and systems. Unfortunately, such design descriptions usually fall outside of the scope of today's synthesis tools. Rather than use multiple **wait** statements to describe such logic, you will probably use **wait** statements only when describing test stimulus.

Using Processes for Combinational Logic

Concurrent signal assignments can be used to create combinational logic. When you write a sequence of concurrent signal assignments, each statement that you write is independent of all other statements and results in a unique combinational function (unless a guarded block or some other special feature is used to imply memory).

If you wish, you can use sequential VHDL statements (in the form of a process or subprogram) to create combinational logic as well. Sequential VHDL statements can actually be more clear and concise for many types of combinational functions, as they allow the priority of operations to be clearly expressed within a combinational logic function.

The following is an example of a simple combinational logic function (a 4-into-1 multiplexer) described using a process:

```
entity simple_mux is
    port (Sel: in bit_vector (0 to 1);
              A, B, C, D: in bit;
              Y: out bit);
end simple_mux;
architecture behavior of simple_mux is
begin
    process(Sel, A, B, C, D)
    begin
        if Sel = "00" then
            Y <= A;
        elsif Sel = "01" then
            Y <= B;
        elsif Sel = "10" then
            Y <= C;
        elsif Sel = "11" then
            Y <= D;
        end if;
    end process;
end simple_mux;
```

This simple process describes combinational logic because it conforms to the following rules:

- The sensitivity list of the process includes all signals that are being read (i.e., used as inputs) within the process.
- Assignment statements written for the process outputs (in this case only output `Y`) cover all possible combinations of the process inputs (in this case `Sel`, `A`, `B`, `C` and `D`).

These two rules dictate whether the signal assignment logic generated from a process is strictly combinational or will require some form of memory element (such as a flip-flop or latch).

For processes that include variable declarations, there is an additional rule that comes into play:

- All variables used in the process must have a value assigned to them before they are read (i.e., used as inputs).

An example of when an apparently combinational logic description actually describes registered logic is demonstrated by the modified (6-into-1) multiplexer description shown below:

```
entity simple_mux is
    port (Sel: in bit_vector (0 to 2);
             A, B, C, D, E, F: in bit;
             Y: out bit);
end simple_mux;
architecture behavior of simple_mux is
begin
    process(Sel, A, B, C, D, E, F)
    begin
        if Sel = "000" then
            Y <= A;
        elsif Sel = "001" then
            Y <= B;
        elsif Sel = "010" then
            Y <= C;
        elsif Sel = "011" then
            Y <= D;
        elsif Sel = "100" then
            Y <= E;
        elsif Sel = "101" then
            Y <= F;
        end if;
    end process;
end simple_mux;
```

This modified version of the multiplexer has only six of the eight possible values for `Sel` described in the **if-then-elsif** statement chain. What happens when `Sel` has a value of `"110"` or `"111"`? Unlike many simpler hardware description languages (most notably languages such as ABEL or CUPL that are intended for programmable logic use), the default behavior in VHDL is to hold the values of unspecified signals. For output `Y` to hold its value when `Sel` has a value of `"110"` or `"111"`, a memory element (such as a latch) will be required. The result is that the circuit as described is no longer a simple combinational logic function.

Understanding what types of design descriptions will result in combinational logic and what types will result in latches and flip-flops is very important when writing VHDL for synthesis.

Using Processes for Registered Logic

Perhaps the most common use of VHDL processes is to describe the behavior of circuits that have memory and must save their state over time. The sequential nature of VHDL processes (and subprograms) make them ideal for the description of such circuits.

If your goal is to create registered logic (using either flip-flop or latch elements), then you will describe your design using one or more of the following methods:

- Write a process that does not include all of its inputs in the sensitivity list.
- Use incompletely specified **if-then-elsif** logic to imply that one or more signals must hold their values under certain conditions.

- Use one or more variables in such a way that they must hold a value between iterations of the process. (For example, specify a variable as an input to an assignment before that variable has been assigned a value itself.)

To ensure the highest level of compatibility with synthesis tools, you should use a combination of methods 1 and 2. The following example demonstrates how registered logic can be described using a process:

```
-- Eight-bit  shifter
--
library ieee;
use ieee.std_logic_1164.all;
entity rotate is
    port( Clk, Rst, Load: in std_logic;
              Data: in std_logic_vector(0 to 7);
              Q: out std_logic_vector(0 to 7));
end rotate;
architecture rotate1 of rotate is
    signal Qreg: std_logic_vector(0 to 7);
begin
    reg: process(Rst,Clk)
    begin
        if Rst = '1' then  -- Async reset
            Qreg <= "00000000";
        elsif (Clk = '1' and Clk'event) then
            if (Load = '1') then
                Qreg <= Data;
            else
                Qreg <= Qreg(1 to 7) & Qreg(0);
            end if;
        end if;
    end process;
    Q <= Qreg;
 end rotate1;
```

In this example, the incomplete **if-then-elsif** statement implies that signal `Qreg` will hold its value when the two conditions (a reset or clock event) are false.

Using Processes for State Machines

State machines are a common form of sequential logic circuits that are used for generating or detecting sequences of events. To describe a synthesizable state machine in VHDL, you should follow a well-established coding convention that makes use of enumerated types and processes. The following example demonstrates how to write a synthesizable state machine description using this coding convention.

The circuit described is a simple freeze-frame unit that grabs and holds a single frame of NTSC color video image. This design description includes the frame detection and capture logic. The complete circuit requires an 8-bit D-A/A-D converter and a 256K X 8 static RAM.

The design description makes use of a number of independent processes. The first process (which has been given the name of `ADDRCTR`), describes a large counter corresponding to the frame address counter in the circuit. This counter description makes use of the IEEE Standard 1076.3 numeric data type unsigned.

The second process, `SYNCCTR`, also describes a counter using the unsigned data type. This counter is used to detect the vertical blanking interval, which indicates the start of one frame of video.

The third and fourth processes (`STREG` and `STTRANS`) describe the operation of the video frame grabber controller logic, using the most common (and most easily synthesized) form for state machines. First, an enumerated type called `states` is declared that consists of the values `StateLive`, `StateWait`, `StateSample`, and `StateDisplay`. Two intermediate signals

(current_state and next_state) are then introduced to represent the current state and calculated next state of the machine. In the processes that follow, signal current_state represents a set of state registers, while next_state represents a combinational logic function.

Process STREG describes the operation of the state registers, and simply loads the value of the calculated next state (signal next_state) into the state registers (current_state) whenever there is a synchronous clock event. This process also includes asynchronous reset logic that will set the machine to its initial state (StateLive) when the Rst input is asserted.

The actual transition logic for the state machine is described in process STTRANS. In this process, a **case** statement is used to decode the current state of the machine (as represented by signal current_state) and define the transitions between states. This is an example where sequential VHDL statements are used to describe non-sequential (combinational) logic.

```
-- A Video Frame Grabber.
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity video is
    port (Reset, Clk: in std_logic;
          Mode: in std_logic;
          Data: in std_logic_vector(7 downto 0);
          TestLoad: in std_logic;
          Addr: out std_logic_vector(17 downto 0);
          RAMWE: out std_logic;
          RAMOE: out std_logic;
          ADOE: out std_logic);
end video;
architecture control1 of video is
    constant FRAMESIZE: integer := 253243;
    constant TESTADDR: integer := 253000;
    signal ENDFR: std_logic;
    signal INCAD: std_logic;
    signal VS: std_logic;
    signal Sync: unsigned (6 downto 0);
begin
    -- Address counter. This counter increments until we reach the end of
    -- the frame (address 253243), or until the input INCAD goes low.
    ADDRCTR: process(Clk)
        variable cnt: unsigned (17 downto 0);
    begin
        if rising_edge(Clk) then
            if TestLoad = '1' then
                cnt := to_unsigned(TESTADDR,18);
                ENDFR <= '0';
            else
                if INCAD = '0' or cnt = FRAMESIZE then
                    cnt := to_unsigned(0,18);
                else
                    cnt := cnt + to_unsigned(1,18);
                end if;
```

```
            if cnt = FRAMESIZE then
                ENDFR <= '1';
            else
                ENDFR <= '0';
            end if;
        end if;
    end if;
    Addr <= std_logic_vector(cnt);
end process;
-- Vertical sync detector. Here we look for 128 bits of zero, which
-- indicates the vertical sync blanking interval.
SYNCCTR: process(Reset,Clk)
begin
    if Reset = '1' then
        Sync <= to_unsigned(0,7);
    elsif rising_edge(Clk) then
        if Data /= "00000000" or Sync = 127 then
            Sync <= to_unsigned(0,7);
        else
            Sync <= Sync + to_unsigned(1,7);
        end if;
    end if;
end process;
VS <= '1' when Sync = 127 else '0';
STATEMACHINE: block
    type states is (StateLive,StateWait,StateSample,StateDisplay);
    signal current_state, next_state: states;
begin
   -- State register process:
 STREG: process(Reset,Clk)
    begin
        if Reset = '1' then
            current_state <= StateLive;
        elsif rising_edge(Clk) then
            current_state <= next_state;
        end if;
    end process;

    -- State transitions:
    STTRANS: process(current_state,Mode,VS,ENDFR)
    begin
        case current_state is
            when StateLive =>    -- Display live video on the output
                RAMWE <= '1';
                RAMOE <= '1';
                ADOE <= '0';
```

```
                INCAD <= '0';
                if Mode = '1' then
                   next_state <= StateWait;
                else
                   next_state <= StateLive
                end if;
            when StateWait =>    -- Wait for vertical sync
                RAMWE <= '1';
                RAMOE <= '1';
                ADOE <= '0';
                INCAD <= '0';
                if VS = '1' then
                   next_state <= StateSample;
                else
                   next_state <= StateWait
                endif;
            when StateSample =>  -- Sample one frame of video
                RAMWE <= '0';
                RAMOE <= '1';
                ADOE <= '0';
                INCAD <= '1';
                if ENDFR = '1' then
                   next_state <= StateDisplay;
                else
                   next_state <= StateSample
                end if;
            when StateDisplay => -- Display the stored frame
                RAMWE <= '1';
                RAMOE <= '0';
                ADOE <= '1';
                INCAD <= '1';
                if Mode = '1' then
                   next_state <= StateLive;
                else
                   next_state <= StateDisplay
                end if;
        end case;
    end process;
end block;
end control1;
```

Specifying State Machine Encodings

The preceding video frame grabber has been described in an implementation-independent fashion, with the assumption that whatever synthesis tool is used to process this design will come up with an optimal solution, in terms of the state encodings selected. For small designs such as this, or when you are not tightly constrained for space, it is probably fine to let the synthesis tool encode your states for you. In many cases, however, you will have to roll up your sleeves and work on improving the synthesis results yourself, by creating your own optimal state encodings. Determining an optimal encoding for a large state

machine can be a long and tedious process. It is important to understand the various coding styles for manually-encoded machines, however, to get the most out of synthesis.

Using Constants for State Encodings

The easiest way to specify an explicit encoding for a state machine is to replace the declaration and use of an enumerated type with a series of constant declarations. For the video frame grabber, for example, you could replace the declarations:

```
type states is (StateLive,StateWait,StateSample,StateDisplay);
signal current_state, next_state: states;
```

with:

```
type states is std_logic_vector(1 downto 0);
constant StateLive: states := "00";
constant StateWait: states := "01";
constant StateSample: states := "11";
constant StateDisplay: states := "10";
signal current_state, next_state: states;
```

Using these declarations will result in the precise encodings that you have specified in the synthesized circuit. There is one additional modification that must be made to this frame grabber state machine if you specify the states using declarations based on std_logic_vector, however. Because the base type of std_logic_vector (std_logic) has nine unique values, the four constants that have been declared (`StateLive`, `StateWait`, `StateSample` and `StateDisplay`) do not represent all possible values for the state type. For this reason, an **others** clause will have to be added to the **case** statement describing the transitions of the machine, as in:

```
when others =>
    null;
```

Using the Enum_encoding Synthesis Attribute

An alternate method of specifying state machine encodings is provided in some synthesis tools. This method makes use of a non-standard (but widely supported) attribute called enum_encoding. The following modified declarations (again, using the video frame grabber state machine as an example) uses the enum_encoding attribute to specify the same state encoding used in the previous example:

```
type states is (StateLive,StateWait,StateSample,StateDisplay);
attribute enum_encoding of states: type is "00 01 11 10";
signal current_state, next_state: states;
```

The `enum_encoding` attribute used in this example has been defined elsewhere (most probably in a special library package provided by the synthesis vendor) as a string:

```
attribute enum_encoding: string;
```

This attribute is recognized by the synthesis tool, which encodes the generated state machine circuitry accordingly. During simulation, the `enum_encoding` attribute is ignored, and the enumerated values are displayed instead.

Specifying a One-hot Encoding

One common technique for optimizing state machine logic is to use what is called a one-hot encoding, in which there is one register dedicated to each state in the machine. One-hot machines require more register resources than more typical, maximally-encoded machines, but can result in tremendous savings in the combinational logic required for next-state and output decoding. This trade-off can be particularly effective in device technologies that have an abundance of built-in registers, but that suffer from limited (or relatively slow) routing resources.

When you first try to use a one-hot approach to state encoding, it is tempting to describe the machine using the same methods that you might have used for your other state machines. The following declarations represent an attempt to encode the video frame grabber state machine one-hot using constant declarations:

```
type states is std_logic_vector(3 downto 0);
constant StateLive: states := "0001";
constant StateWait: states := "0010";
constant StateSample: states := "0100";
```

```
constant StateDisplay: states := "1000";
signal current_state, next_state: states;
```

At first glance this looks correct; each state is represented by a single bit being asserted, and when simulated and synthesized, the machine will indeed transition to the appropriate encoded state for each transition described in the **case** statement shown earlier. In terms of the logic required for state decoding, however, a genuine one-hot machine has not been achieved. This is because the **case** statement written describing the state transitions implicitly refers to all four state registers when decoding the current state of the machine. A true, optimal one-hot machine only requires that one register be observed to determine if the machine is in a given state.

To generate the correct logic, optimized as a one-hot encoded machine, the description has to be modified somewhat, so that only one state register is examined for each possible transition. The easiest way to do this is to replace the **case** statement with a series of **if** statements, as follows:.

```
-- State transitions for one-hot encoding:
STTRANS: process(current_state,Mode,VS,ENDFR)
begin
        if current_state(0) = '1' then   -- StateLive
                RAMWE <= '1';
                RAMOE <= '1';
                ADOE <= '0';
                INCAD <= '0';
                if Mode = '1' then
                   next_state <= StateWait;
                else
                   next_state <= StateLive
                end if;
        end if;
        if current_state(1) = '1' then   -- StateWait
                RAMWE <= '1';
                RAMOE <= '1';
                ADOE <= '0';
                INCAD <= '0';
                if VS = '1' then
                   next_state <= StateSample;
                else
                   next_state <= StateWait
                end if;
        end if;
        if current_state(2) = '1' then   -- StateSample
                RAMWE <= '0';
                RAMOE <= '1';
                ADOE <= '0';
                INCAD <= '1';
                if ENDFR = '1' then
                   next_state <= StateDisplay;
                else
                   next_state <= StateSample
                end if;
        end if;
```

```
        if current_state(3) = '1' then   -- StateDisplay
                RAMWE <= '1';
                RAMOE <= '0';
                ADOE <= '1';
                INCAD <= '1';
                if Mode = '1' then
                   next_state <= StateLive;
                else
                   next_state <= StateDisplay
                end if;
        end if;
end process;
```

This description can be made more readable by introducing constants for the index values for each state register.

Using Processes for Test Stimulus

In addition to their use for describing combinational and registered circuits to be synthesized or modeled for simulation, VHDL processes are also important for describing the test environment in the form of sequential application of stimulus and (if desired) checking of resulting circuit outputs.

A process that is intended for testing (as part of a test bench) will normally have no sensitivity list. Instead, it will have a series of **wait** statements that provide time for the unit under test to stabilize between the assignment of test inputs. Because a process intended for use as a test bench does not describe hardware to be synthesized, you are free to use any legal features and style of VHDL without regard to the limitations of synthesis.

The following is a simplistic test bench example written with a single **process** statement. This **process** statement might be used to apply a sequence of input values to a lower-level circuit and check the state of that circuit's outputs at various points in time.

```
-- A simple process to apply various stimulus over time...
process
    constant PERIOD: time := 40 ns;
begin
    Rst <= '1';
    A <= "00000000";
    B <= "00000000";
    wait for PERIOD;
    CheckState(Q, "00000000");
    Rst <= '0';
    A <= "10101010";
    B <= "01010101";
    wait for PERIOD * 4;
    CheckState(Q, "11111111");
    A <= "11111010";
    B <= "01011111";
    wait for PERIOD * 2;
    CheckState(Q, "00110101");
    wait;
end process;
```

In this example, the process executes just once before suspending indefinitely (as indicated by the final **wait** statement). The stimulus is described by a sequence of assignments to signals `A` and `B`, and by calls to a procedure (defined elsewhere) named `CheckState`. **Wait** statements are used to describe a delay between each test sequence.

Sequential Statements in Subprograms

We've seen examples of how sequential statements are written in a **process** statement. The **process** statement is relatively easy to understand if you think of it as a small software program that executes independent of other processes and concurrent statements during simulation.

Functions and procedures (which are collectively called subprograms) are very similar to processes in that they contain sequential statements executed as independent 'programs' during simulation. The parameters you pass into a subprogram are analogous to the sensitivity list of a process; whenever there is an event on any object (signal or variable) being passed as an argument to a subprogram, that subprogram is executed and its outputs (whether they are output parameters, in the case of a procedure, or the return value, in the case of a function) are recalculated.

The following example includes a procedure declared within the architecture. The procedure counts the number of ones and zeroes there are in a std_logic_vector input (of arbitrary width) and returns the count values as output parameters. The procedure is used to build two result strings containing the appropriate number of ones and zeroes, left justified and padded with 'X' values. (For example, an input with the values "1010001001" would result in the values "1111XXXXXX" and "000000XXXX".)

```
entity proc is
    port (Clk: in std_logic;
          Rst: in std_logic;
          InVector: in std_logic_vector(0 to 9);
          OutOnes: out std_logic_vector(0 to 9);
          OutZeroes: out std_logic_vector(0 to 9));
end proc;
architecture behavior of proc is
    procedure CountBits(InVector: in std_logic_vector;
                        ones,zeroes: out natural) is
        variable cnt1: natural := 0;
        variable cnt0: natural := 0;
    begin
        for I in 0 to InVector'right loop
            case InVector(I) is
                when '1' => cnt1 := cnt1 + 1;
                when '0' => cnt0 := cnt0 + 1;
                when others => null;
            end case;
        end loop;
        ones := cnt1;
        zeroes := cnt0;
    end CountBits;
    signal Tmp1, Tmp0: std_logic_vector(0 to 9);
begin
    process(Rst, Clk)
    begin
        if Rst = '1' then
            OutOnes <= (others => '0');
            OutZeroes <= (others => '0');
        elsif rising_edge(Clk) then
            OutOnes <= Tmp1;
            OutZeroes <= Tmp0;
```

```
        end if;
    end process;
    process(InVector)
        variable ones, zeroes: natural;
    begin
        Countbits(InVector,ones,zeroes);
        Tmp0 <= (others => 'X');
        Tmp1 <= (others => 'X');
        for I in 0 to ones - 1 loop
            Tmp1(I) <= '1';
        end loop;
        for I in 0 to zeroes - 1 loop
            Tmp0(I) <= '0';
        end loop;
    end process;
end behavior;
```

This example shows that a procedure containing sequential statements can be invoked from within a process – or even from within another procedure. The calling process simply suspends until the procedure has completed execution.

Notes

The example above is theoretically synthesizable, but the fact that the procedure has been written without regard to the width of the inputs will probably make it impossible to process by synthesis tools. If this design were to be synthesized, the variables `cnt1` and `cnt0` would have to be given range constraints.

Signal and Variable Assignments

One important aspect of VHDL you should clearly understand is the relationship between sequential statements (in a process or subprogram) and the scheduling of signal and variable assignments. Signals within processes have fundamentally different behavior from variables. Variables are assigned new values immediately, while signal assignments are scheduled and do not occur until the current process (or subprogram) has been suspended. When you describe complex logic using sequential assignments, you must carefully consider which type of object (signal or variable) is appropriate for that part of your design.

An example of where signal assignments would be appropriate, is an 8-bit serial cyclic-redundancy-check (CRC) generator. Signals are required because a chain of registers are being constructed. Each register in the chain is clocked from a common source, and data moves from one register to the next only when there is an event on `Clk`. The data could be described as being “scheduled.”

```
-- 8-bit Serial CRC Generator.
--
library ieee;
use ieee.std_logic_1164.all;
entity crc8s is
   port (Clk,Set, Din: in std_logic;
         CRC_Sum: out std_logic_vector(15 downto 0));
end crc8s;
architecture behavior of crc8s is
    signal X: std_logic_vector(15 downto 0);
begin
    process(Clk,Set)
    begin
        if Set = '1' then
            X <= "1111111111111111";
```

```
        elsif rising_edge(Clk) then
            X(0)  <= Din xor X(15);
            X(1)  <= X(0);
            X(2)  <= X(1);
            X(3)  <= X(2);
            X(4)  <= X(3);
            X(5)  <= X(4) xor Din xor X(15);
            X(6)  <= X(5);
            X(7)  <= X(6);
            X(8)  <= X(7);
            X(9)  <= X(8);
            X(10) <= X(9);
            X(11) <= X(10);
            X(12) <= X(11) xor Din xor X(15);
            X(13) <= X(12);
            X(14) <= X(13);
            X(15) <= X(14);
        end if;
    end process;
    CRC_Sum <= X;
end behavior;
```

Because the data moving from register to register is scheduled, this example would not work if `X` was described using a variable instead of a signal. If a variable was substituted for `X`, the assignments for each stage of the CRC generation would be immediate and thus would not describe a chain of registers.

Also, the assignment of `X` to `CRC_Sum` must be placed outside the process. If you were to write the assignment to `CRC_Sum` within the process, as in:

```
      . . .
            X(14) <= X(13);
            X(15) <= X(14);
        end if;
       CRC_Sum <= X;
    end process;
end behavior;
```

the result would not be what you intended. This is because the assignment of `CRC_Sum` will be subject to the execution and signal assignment rules of a process. In this case, the assignment of a final value to `X` will be delayed until the process suspends, and `CRC_Sum` will not be updated until the next time the process executes. (As it turns out, the next time the process executes may well be on the falling edge of the clock, meaning that `CRC_Sum` would be delayed by half a clock cycle.)

If-then-else Statements

VHDL includes a variety of control statements that can be used to describe combinational functions, indicate priorities of operations, and specify other high-level behavior.

The **if-then-else** construct is the most common form of control statement in VHDL. The general form of the **if-then-else** construct is:

```
if first_condition then
    statements
elsif second_condition then
   statements
else
```

```
   statements
end if;
```

The conditions specified in an **if-then-else** construct must evaluate to a Boolean type. This means that the following example is incorrect:

```
procedure Mux(signal A, B, S: in std_logic; signal O: out std_logic) is
begin
    if S then      -- Error:  S is not Boolean!
        O <= B;
    else
        O <= A;
    end if;
end Mux;
```

Instead, this example must be modified so that the **if** statement condition evaluates to a Boolean expression:

```
if S = '1' then     -- Now it will work...
        O <= B;
    else
        O <= A;
    end if;
end Mux;
```

The statement parts of an **if-then-else** construct can contain any sequential VHDL statements, including other **if-then-else** statement constructs. This means that you can nest multiple levels of **if-then-else** statements, in the following form:

```
if outer_condition then
    statements
else
    if inner_condition then
        statements
    end if;
end if;
```

Case Statements

Case statements are a type of control statement that can be used as alternatives to **if-then-else** constructs. **Case** statements have the following general form:

```
case control_expression is
    when test_expression1  =>
        statements
    when test_expression2  =>
        statements
    when others  =>
        statements
end case;
```

The test expressions of a **case** statement must be mutually exclusive, meaning that no two test expressions are allowed to be true at the same time. **Case** statements must also include all possible conditions of the control expression. (The **others** expression can be used to guarantee that all conditions are covered.)

The primary difference between descriptions written using **case** statements from those written using **if-then-else** statements is that **if-then-else** statements imply a priority of conditions, while a **case** statement does not imply any priority. (This is similar to the difference between conditional and selected assignments).

Loops

Loop statements are a category of control structures that allow you to specify repeating sequences of behavior in a circuit. There are three primary types of loops in VHDL: **for** loops, **while** loops, and **infinite** loops.

For Loop

The **for** loop is a sequential statement that allows you to specify a fixed number of iterations in a behavioral design description. The following architecture demonstrates how a simple 8-bit parity generator can be described using a **for** loop:

```
library ieee;
use ieee.std_logic_1164.all;
entity parity10 is
    port(D: in std_logic_vector(0 to 9);
         ODD: out std_logic);
    constant WIDTH: integer := 10;
end parity10;
architecture behavior of parity10 is
begin
    process(D)
        variable otmp: Boolean;
    begin
        otmp := false;
        for i in 0 to D'length - 1 loop
            if D(i) = '1' then
                otmp := not otmp;
            end if;
        end loop;
        if otmp then
            ODD <= '1';
        else
            ODD <= '0';
        end if;
    end process;
end behavior;
```

The **for** loop includes an automatic declaration for the index (`i` in this example). You do not need to separately declare the index variable.

The index variable and values specified for the loop do not have to be numeric types and values. In fact, the index range specification does not even have to be represented by a range. Instead, it can be represented by a type or sub-type indicator. The following example shows how an enumerated type can be used in a loop statement:

```
architecture looper2 of my_entity is
    type stateval is Init, Clear, Send, Receive, Error;      -- States of a machine
begin
    . . .
    process(a)
    begin
        for state in stateval loop
            case state is
                when Init =>
                    ...
```

```
                when Clear =>
                    ...
                when Send =>
                    ...
                when Receive =>
                    ...
                when Error =>
                    ...
            end case;
        end loop;
    end process;
    . . .
end looper2;
```

For loops can be given an optional name, as shown in the following example:

```
loop1: for state in stateval loop
    if current_state = state  then
         valid_state <= true;
    end if;
end loop loop1;
```

The loop name can be used to help distinguish between the loop index variable and other similarly-named objects, and to specify which of the multiple nested loops is to be terminated. Otherwise, the loop name serves no purpose.

While Loop

A **while** loop is another form of sequential loop statement that specifies the conditions under which the loop should continue, rather than specifying a discrete number of iterations. The general form of the **while** loop is shown below:

```
architecture while_loop of my_entity is
begin
    . . .
    process(. . .)
    begin
        . . .
        loop_name: while (condition) loop
             -- repeated statements go here
        end loop loop_name;
        . . .
    end process;
    . . .
end while_loop;
```

Like the **for** loop, a **while** loop can only be entered and used in sequential VHDL statements (i.e., in a process, function or procedure). The loop name is optional.

The following example uses a **while** loop to describe a constantly running clock that might be used in a test bench. The loop causes the clock signal to toggle with each loop iteration, and the loop condition will cause the loop to terminate if either of two flags (`error_flag` or `done`) are asserted.

```
process
begin
    while error_flag /= '1'  and done /= '1' loop
        Clock <= not Clock;
        wait for CLK_PERIOD/2;
```

```
    end loop;
end process;
```

Notes

Although **while** loops are quite useful in test benches and simulation models, you may have trouble if you attempt to synthesize them. Synthesis tools may be unable to generate a hardware representation for a **while** loop, particularly if the loop expression depends on non-static elements such as signals and variables. Because support for **while** loops varies widely among synthesis tools, it is recommend that you not use them in synthesizable design descriptions.

Infinite Loop

An infinite loop is a loop statement that does not include a **for** or **while** iteration keyword (or iteration scheme). An infinite loop will usually include an **exit** condition, as shown in the template below:

```
architecture inifinite_loop of my_entity is
begin
    . . .
    process(. . .)
        . . .
        loop_name: loop
            . . .
            exit when (condition);
        end loop loop_name;
    end process;
    . . .
end infinite_loop;
```

An infinite loop using a **wait** statement is shown in the example below. This example exhibits exactly the same behavior as a **while** loop:

```
process
begin
    loop
        Clock <= not Clock;
        wait for CLK_PERIOD/2;
        if done = '1' or error_flag = '1' then
            exit;
        end if;
    end loop;
end process;
```

As with a **while** loop, an infinite loop probably has no equivalent in hardware and is therefore not synthesizable.

Loop Termination

There are many possible reasons for wanting to jump out of a loop before its normal terminating condition has been reached. The three types of loops previously described all have the ability to be terminated prematurely. Loop termination is performed through the use of an **exit** statement. When an **exit** statement is encountered, its condition is tested and, if the condition is true, the simulator skips the remaining statements in the loop and all remaining loop iterations, and continues execution at the statement immediately following the **end loop** statement.

The following example demonstrates how loop termination can be used to halt a sequence of test vectors that are being executed when an error is detected:

```
for i in 0 to VectorCount loop
    ApplyVector(InputVec(i), ResultVec);
     exit when CheckOutput(OutputVec(i), ResultVec) = FatalError;
end loop;
```

The exit condition is optional; an **exit** statement without an exit condition will unconditionally terminate when the **exit** statement is encountered. The following example shows an unconditional **exit** termination specified in combination with an **if-then** statement to achieve the same results as in the previous example:

```
for i in 0 to VectorCount loop
    ApplyVector(InputVec(i), ResultVec);
     if CheckOutput(OutputVec(i), ResultVec) = FatalError then
        exit;
end loop;
```

When multiple loops are nested, the **exit** statement will terminate only the innermost loop. If you need to terminate a loop that is not the innermost loop, you can make use of loop labels to specify which loop is being terminated. The following example shows how loop labels are specified in **exit** statements:

```
LOOP1: while (StatusFlag = STATUS_OK) loop
       GenerateSequence(InputVec,OutputVec,VectorCount,Seed);
        LOOP2: for i in 0 to VectorCount loop
            ApplyVector(InputVec(i), ResultVec);
            ErrStatus := CheckOutput(OutputVec(i), ResultVec) = TestError;
            if ErrStatus = ERR_COMPARE then
                ReportError();
                exit LOOP2;
            elsif ErrStatus = ERR_FATAL then
                ReportFatal();
                exit LOOP1;
            end if;
        end loop LOOP2;
end loop LOOP1;
```

Modularity Features

Modular (or structured) programming is a technique that you can use to enhance your own design productivity, as well as that of your design team. A modular design approach allows commonly-used segments of VHDL code to be re-used. It also enhances design readability.

VHDL includes many features that can help you create modular designs. The following links look at features that allow you to quickly and easily create reusable segments of your design, based on methods similar to those used in software programming languages.

Functions and Procedures

Functions and procedures in VHDL, which are collectively known as subprograms, are directly analogous to functions and procedures in a high-level software programming language such as C or Pascal. A procedure is a subprogram that has an argument list consisting of inputs and outputs, and no return value. A function is a subprogram that has only inputs in its argument list, and has a return value.

Subprograms are useful for isolating commonly-used segments of VHDL source code. They can either be defined locally (within an architecture, for example), or they can be placed in a package and used globally throughout the design description or project.

Statements within a subprogram are sequential (like a process), regardless of where the subprogram is invoked. Subprograms can be invoked from within the concurrent area of an architecture or from within a sequential process or higher-level subprogram. They can also be invoked from within other subprograms.

Subprograms are very much like processes in VHDL. In fact, any statement that you can enter in a VHDL process can also be entered in a function or procedure, with the exception of a **wait** statement (since a subprogram executes once each time it is invoked and cannot be suspended while it is executing). It is therefore useful to think of subprograms as processes that (1) have been located outside the body of an architecture, and (2) operate only on their input and (in the case of procedures) their output parameters.

Nesting of functions and procedures is allowed to any level of complexity, and recursion is also supported in the language. (Of course, if you expect to generate actual hardware from your VHDL descriptions using synthesis tools, then you will need to avoid writing recursive functions and procedures, as such descriptions are not synthesizable).

Functions

A function is a subprogram that accepts zero or more input arguments and returns a single output value. Because a function returns a value, it has a type associated with it. The following is an example of a function that accepts two integer arguments and returns the greater of the two as an integer value:

```
function maxval (arg1, arg2: integer) return integer is
     variable result: integer;
begin
     if arg1 > arg2 then
         result := arg1;
     else
         result := arg2;
     end if;
      return result;
end maxval;
```

The arguments to a function are all inputs to the function. They cannot be modified or otherwise assigned values within the function. By default, the arguments are of a constant kind. This means that the arguments are interpreted within the function as if they had been supplied as constants declared in the function itself. An alternative type of argument, indicated by the use of the **signal** keyword, allows the use of signal attributes (such as 'event) within the function. The following function (which is provided in the IEEE 1164 standard library) demonstrates the use of a signal argument in a function:

```
function rising_edge (signal s: std_logic) return boolean is
begin
```

```
    return (s'event and (To_X01(s) = '1') and
        (To_X01(s'last_value) = '0'));
end rising_edge;
```

In this example, the keyword **signal** is critical to the correct operation of the function. In the absence of the **signal** keyword, the `'event` attribute would not be preserved.

Functions are most commonly used in situations where you require a calculation or conversion based on the subprogram inputs. Examples of this include arithmetic or logic functions (such as the one just presented), type conversion functions, and value checks such as you might use when writing a test bench.

Because they return a value, functions must be used as part of a larger expression. The following VHDL code fragment demonstrates a type conversion function being used in an expression to convert an array data type to an integer:

```
signal Offset: integer range (0 to 1023);
signal BUS1: std_logic_vector(11 downto 0);
 . . .
Offset <= to_integer(BUS1) + 136;
```

Operators as Functions

One interesting feature of VHDL is its support for operator overloading. Operator overloading allows you to specify custom functions representing symbolic operations for your own data types. To define a new operation (or modify an existing one), you simply write a function and enclose its name (which can be a non-numeric name such as an operator symbol) in double-quote characters.

The following operator function is taken directly from the IEEE 1164 standard logic package, and demonstrates how operator overloading works:

```
function "and" (l : std_logic; r : std_logic ) return UX01 is
begin
    return(and_table(l, r));
end "and";
```

In this example, the function `and` is declared as a function returning the type `UX01` (a four-valued logic type used internally in the standard logic package). The function is identified during compilation by its name (`and`) and by the types and number of its arguments. For example, in the expression:

```
architecture simple of and_operation is
    signal Y, A, B: std_logic;
begin
    Y <= A and B;
end simple;
```

the **and** operation is actually a function defined using the previously listed statements. In fact, all of the standard operations that you use in VHDL (including such operators as and, or, not, +, -, *, & and <) are actually functions declared in libraries such as std and ieee.

Notes

In source code listings presented in this document the typographic convention of listing all VHDL keywords in bold face has been used. As you have just seen, however, many of the keywords that are listed in bold face are actually functions defined in a standard library.

Procedures

Procedures differ from functions in that they do not have a return value, and their arguments may include both inputs and outputs to the subprogram. Because each argument to a procedure has a mode (**in**, **out**, or **inout**), they can be used very much like you would use an entity/architecture pair to help simplify and modularize a large and complex design description.

Procedures are used as independent statements, either within the concurrent area of an architecture or within the sequential statement area of a process or subprogram.

The following sample procedure defines the behavior of a clocked JK flip-flop with an asynchronous reset:

```
procedure jkff (signal Rst, Clk: in std_logic;
```

```
                        signal J, K: in std_logic;
                        signal Q,Qbar: inout std_logic) is
begin
    if Rst = '1' then
        Q <= '0';
    elsif Clk = '1' and Clk'event then
        if J = '1' and K = '1' then
            Q <= Qbar;
        elsif J = '1' and K = '0' then
            Q <= '1';
        elsif J = '0' and K = '1' then
            Q <= '0';
        end if;
    end if;
    Qbar <= not Q;
end jkff;
```

A procedure may include a **wait** statement, unless it has been called from within a process that has a sensitivity list.

Notes

Variables declared and used within a procedure are not preserved between different executions of the procedure. This is unlike a process, in which variables maintain their values between executions. Variables within a procedure therefore do not maintain their values over time, unless the procedure is suspended with a **wait** statement.

Declaring a Global Subprogram

Functions and procedures can be declared either globally, so they are usable throughout a design description, or they can be declared locally within the declarative region of an architecture, block, process, or even within another subprogram. If you are writing a subprogram that will be used throughout your design, you will write the subprogram declaration in an external package, as shown in the following example:

```
package my_package is
    function my_global_function(...)
        return bit;
end my_package;
package body my_package is
    function my_global_function(...)
        return bit is
    begin
        . . .
    end my_global_function;
end my_package;
. . .
use work.my_package.my_global_function;
entity my_design is
begin
    . . .
end my_design;
```

In this example, the function `my_global_function()` has been declared within the package `my_package`. The actual body of the function – the sequence of statements that define its operation – is placed into a package body. (The reasons why a subprogram requires a package body in addition to a package are somewhat obscure, but they have to do with the fact that the statements in a subprogram must be executed when the design description is simulated, while other declarations appearing in a

package can be completely resolved at the time the VHDL description is first analyzed by the VHDL compiler.) To use the global function in subsequent architectures (such as the architecture associated with entity `my_design` in this example), a **use** statement (and **library** statement, if the package has been compiled into a named library) must precede the declaration for that architecture or its parent entity.

Declaring a Local Subprogram

Another way of using subprograms is to declare them locally, such as within an architecture or block declaration. In the following example, `my_local_function()` has been declared entirely within the architecture `my_architecture`:

```
architecture my_architecture of my_design is
begin
    my_process: process(...)
        function my_local_function(...)
            return bit is
        begin
            . . .
        end my_local_function;
    begin
        . . .
    end process my_process;
end my_architecture;
```

This example demonstrates the concept of local scoping. VHDL objects (such as signals, variables and constants) can be declared at many points in a design, and that the visibility, or scoping, of those objects depends on where they have been declared. Subprograms (functions and procedures) also have scoping. In this example, the function `my_local_function` can only be referenced within the architecture in which it has been declared and defined.

Consistent scoping of objects and subprograms is an important part of modular VHDL coding and of structured programming in general. If you will only be using an object or subprogram in one section of your overall design, then you should keep the declaration of that object or subprogram local to that section of the design. This will make it possible to re-use that section of the design elsewhere with a minimum of fuss (since you won't have to remember to declare the object or subprogram globally in the new design).

Subprogram Overloading

Because a function or procedure is uniquely identified by its name in combination with its argument types, there can be more than one function or procedure defined with the same name, depending on the types of the operands required. This feature (called subprogram overloading) is important because the function required to perform a given operation on one type of data may be quite different than the function required for another type.

It is unlikely that you will need to use subprogram overloading in your own design efforts. Instead, you will use the standard data types provided for you in the language standards, and you will use the predefined operators for those data types exclusively. You might find it useful, however, to look over the operators defined in the standard libraries so you have a better idea of the capabilities of each standard data type provided.

Parameter Types

Subprograms operate on values or objects that are passed in as parameters to the subprogram. Procedures differ from functions in that they can also pass information out on the parameter list. (The parameters of a procedure have directions, or modes.)

There are three classes of parameters available for subprograms: **constant**, **variable** and **signal**. The default class, if no other class is specified, is **constant**. The parameters that are used within the function or procedure are called the formal parameters, while the parameters passed into the function or procedure are called the actual parameters.

The primary difference between **constant**, **variable** and **signal** parameters is the type of actual parameters that can be passed into the subprogram when it is called. If the formal parameter of a subprogram is of class **constant**, the actual parameter can be any expression that evaluates to a data type matching that of the formal parameter. For parameters of class **variable** or **signal**, the actual parameters must be **variable** or **signal** objects, respectively.

Parameters of subprograms transfer only the value of the actual parameters (those parameters specified when the subprogram is called) for the formal parameters (the parameters specified in the subprogram declaration). Attribute information is not passed directly into the subprogram. (The attributes that you will most often be concerned with, such as 'event, will be available if you are using parameters of class **signal**.)

Mapping of Parameters

Many of the examples used in the topics within this sub-folder have used what is referred to as positional association to describe how actual parameters are paired with formal parameters of the subprogram.

Positional association is a quick and convenient way to describe the mapping of parameters, but it can be error-prone.

For this reason, you might want to write your subprogram references using an alternate form of port map called named association. Named association guarantees that the correct parameters are connected, and it also gives you the ability to re-order the parameters as needed.

The following example shows how the same subprogram might be referenced using both positional and named association:

```
dff(Rst,Clk,Data,Result);
dff(Rst=>Rst,C=>Clk,D=>Data,a=>Result);
```

The special operator => indicates exactly which lower-level ports are to be connected to which higher-level signals.

Partitioning Features

VHDL provides many high-level features to help you manage a complex design description. In fact, design management is one of VHDL's key strengths when compared to alternative design entry languages and methods.

The modularity features (procedures and functions) are one aspect of design management, allowing commonly-used declarations and sequential statements to be collected in one place. Design partitioning is another important aspect of design management. Design partitioning goes beyond simpler design modularity methods to provide comprehensive design management across multiple projects and allow alternative structural implementations to be tried out with minimal effort.

Design partitioning is particularly useful for those designs being developed in a team environment, as it promotes cooperative design efforts and well-defined system interfaces.

Blocks

Blocks are the simplest form of design partitioning. They provide an easy way to segment a large VHDL architecture into multiple self-contained parts. **Blocks** allow the logical grouping of statements within an architecture, and provide a place to declare locally-used signals, constants, and other objects as needed.

VHDL blocks are analogous to sheets in a multi-sheet schematic. They do not represent re-usable components (unless you re-use them by copying them with your text editor or by using configurations), but do enhance readability by allowing declarations of objects to be kept close to where those objects are actually used.

The general form of the **block** statement is shown below:

```
architecture my_arch of my_entity is
begin
    BLOCK1: block
        signal a,b: std_logic;
    begin
        -- some local statements here
    end block BLOCK1;
    BLOCK2: block
        signal a,b std_logic;
    begin
        -- some other local statements here
        -- Note that 'a' and 'b' are unique to this block!
    end block BLOCK2;
end my_arch;
```

This simple example includes two blocks, named `BLOCK1` and `BLOCK2`, that each include declarations for local signals. In the first block, `BLOCK1`, the signals `a` and `b` are declared prior to the **begin** statement of the block. These signals are therefore local to block `BLOCK1` and are not visible outside of it. The second block, `BLOCK2`, also has declarations for local signals named `a` and `b`, but these are not the same signals as those declared in block `BLOCK1`.

This concept of local declarations is important to understand and is probably familiar to you if you have used high-level programming languages. One of the most important techniques of structured programming (whether you are describing software or hardware) is to minimize the overall complexity of your design description by localizing the declarations as much as is practical. Keeping signals local will make the design description easier to read, allow it to be modified more easily in the future, and also enhance design re-use, since it will be easier to copy one portion of the design to another project or source file.

Nested Blocks

Blocks can be nested, as shown in the following example:

```
architecture my_arch of my_entity is
begin
    BLOCK1: block
```

```
        signal a,b: std_logic;
    begin
        BLOCK2: block
            signal c,d std_logic;
        begin
            -- This block is now local to block BLOCK1 and has
            -- access to 'a' and 'b'
        end block BLOCK2;
    end block BLOCK1;
end my_arch;
```

In this example, block BLOCK2 has been placed within block BLOCK1. This means that all declarations made within BLOCK1 (signals a and b, in this example) are visible both within block BLOCK1 and block BLOCK2. The reverse is not true, however. The declarations for c and d within block BLOCK2 are local only to BLOCK2 and are not visible outside that block. What happens when the same signals are declared in two blocks that are nested? Consider the following:

```
architecture my_arch of my_entity is
begin
    BLOCK1: block
        signal a,b: std_logic;
    begin
        BLOCK2: block
            signal a,b std_logic;
        begin
            -- This a and b overrides previous
        end block BLOCK2;
    end block BLOCK1;
end my_arch;
```

In this example, the signals a and b are declared both in the outer block (BLOCK1) and in the inner block (BLOCK2). The result is that the signals a and b in the outer block are hidden (but not replaced or overwritten) by the declarations of a and b in the inner block.

Guarded Blocks

Guarded blocks are special forms of block declarations that include an additional expression known as a guard expression. The guard expression enables or disables drivers within the block, allowing circuits such as latches and output enables to be easily described using a dataflow style of VHDL.

The following example shows how a guarded block can be used to describe the operation of a latch:

```
use ieee.std_logic_1164.all;
entity latch is
    port( D, LE: in std_logic;
             Q, QBar: out std_logic);
end latch;
architecture mylatch of latch is
begin
    L1: block (LE = '1')
    begin
        Q <= guarded D after 5 ns;
        QBar <= guarded not(D) after 7 ns;
    end block L1;
end mylatch;
```

In this example, the guard expression `LE = '1'` applies to all signal assignments that include the **guarded** keyword. (Guard expressions are placed in parentheses after the **block** keyword.) The signal assignments for `Q` and `QBar` therefore depend on the value of `LE` being `'1'`. When `LE` is not `'1'`, the guarded signals hold their values.

Notes

If you need to access a signal that has been effectively hidden by a declaration of the same name, you can qualify the signal name with a block name prefix, as in `BLOCK1.a` or `BLOCK1.b`.

Guarded blocks are not supported by all synthesis tools, so it is not recommended that you use them for designs intended for synthesis. Instead, you should use a process or subprogram to describe the behavior of registered or latched circuits.

Packages

Packages are intended to hold commonly-used declarations such as constants, type declarations and global subprograms. Packages can be included within the same source file as other design units (such as entities and architectures) or may be placed in a separate source file and compiled into a named library. This latter method is useful when you will be using the contents of a package throughout a large design or in multiple projects.

Packages may contain the following types of objects and declarations:

- Type and subtype declarations
- Constant declarations
- File and alias declarations
- Component declarations
- Attribute declarations
- Functions and procedures
- Shared variables

When items from the package are required in other design units, you must include a **use** statement to make the package and its contents visible for each design unit.

The following is an example of a package declaration and its corresponding **use** statements:

```
library ieee;
use ieee.std_logic_1164.all;
package my_types is
    subtype byte is std_logic(0 to 7);
    constant CLEAR: byte := (others=>'0');
end my_types;
use work.my_types.all;
use ieee.std_logic_1164.all;
entity rotate is
    port(Clk, Rst, Load: in std_logic;
            Data: in byte;
            Q: out byte);
end rotate;
architecture rotate4 of rotate is
    signal Qreg: byte;
begin
    Qreg <= Data when (Load = '1') else
                Qreg(1 to byte'LENGTH-1) & Qreg(0);
    dff(Rst, Clk, Qreg, Q);
end rotate4;
```

In this example, the package `my_types` includes declarations for a subtype (`byte`) and constant (`CLEAR`) that will be used throughout the subsequent design description. The statement **use** `work.my_types.all` specifies that all contents of the

package `my_types` should be loaded from the default library (`work`). (The `work` library is a special library described in the VHDL specification as one that does not require a **library** statement and into which all design units are analyzed by default.) An alternative to using the **all** keyword in the **use** statement would be to specify precisely which items in the default library are to be made visible, as in **use** `work.my_types.byte` and **use** `work.my_types.CLEAR`.

How Are Packages Used?

When you create your own VHDL design descriptions, you can use packages in a number of ways. First, you can dramatically simplify your designs by placing commonly-used declarations (such as `byte` and `CLEAR` in the previous example) into packages that are used throughout your project. You will probably find that using libraries to collect such packages in one place will simplify the design even further and make it easier to share commonly-used declarations between different design descriptions.

Another way you can use packages is to reference pre-written packages that have been provided for you. One example of such a package is found in the IEEE 1164 Standard Logic standard. The IEEE 1164 standard provides a standard package named std_logic_1164 that includes declarations for the types std_logic, std_ulogic, std_logic_vector and std_ulogic_vector, as well as many useful functions related to those data types.

Packages may also be provided to you by vendors of synthesis and simulation tools. Synthesis tools, for example, often include packages containing synthesizable type conversion functions, synthesizable procedures for flip-flops and latches, and other useful design elements.

Finally, there is a standard package that includes declarations for all the standard data types (bit, bit_vector, integer and so on). This standard package is defined by the IEEE 1076 standard and automatically made visible to all design units. (You do not have to specify a **use** clause for the standard package.)

Package Bodies

Packages that include global subprograms (functions or procedures) or deferred constants must defer part of their declaration (the part that must be analyzed during simulation) to a separate design unit called a **package body**. Every package can have, at most, one corresponding package body. Package bodies are optional and are only required when a package includes subprograms or deferred constants.

The following example shows how a package body must be used when a subprogram (in this case, a procedure describing the behavior of a D flip-flop) is declared in a package:

```
package my_reg8 is
    subtype byte8 is std_logic_vector(0 to 7);
    constant CLEAR8: byte8 := (others=>'0');
    procedure dff8 (signal Rst, Clk: in std_logic;
                                signal D: in byte;
                                signal Q: out byte);
end my_reg8;
package body my_reg8 is
    procedure dff8 (signal Rst, Clk: in std_logic;
                                 signal D: in byte8;
                                 signal Q: out byte8) is
    begin
        if Rst = '1' then
            Q <= CLEAR8;
        elsif Clk = '1' and Clk'event then
            Q <= D;
        end if;
    end dff;
end my_reg8;
```

In this example, the procedure `dff8` is declared initially in the package `my_reg8`. This first declaration is somewhat akin to a "function prototype" as used in the C or C++ languages, and it defines the interface to the procedure. The package body that corresponds to package `my_reg8` (and shares its name) contains the complete description of the procedure.

Design Libraries

A design library is defined in the VHDL 1076 standard as "an implementation-dependent storage facility for previously analyzed design units". This rather loose definition has resulted in many different implementations in synthesis and simulation tools. In general, however, you will find that design libraries are used to collect commonly-used design units (typically packages and package bodies) into uniquely-named areas that can be referenced from multiple source files in your design.

In a typical simulation environment, you will specify to the simulator the library into which you want each design unit compiled (or analyzed, to use the terminology of the VHDL standard). If you do not specify a library, the design units are compiled into a default library named **work**.

For simple design descriptions (such as those that are completely represented within a single source file), you will use the **work** library exclusively and will not have to put much thought into how libraries are implemented in the set of tools you are using. When you use the **work** library exclusively, all you need to do is specify a **use** statement such as:

```
use work.my_package.all;
```

prior to each entity declaration in your design for each package that you have declared in your source file. (You do not have to place **use** statements prior to an architecture declaration if the corresponding entity declaration is preceded by a **use** statement.)

If, however, you choose to use named libraries in your designs (and you are encouraged to do so, as it can dramatically improve your design productivity), then you should follow a few simple rules to avoid compatibility problems when moving between different simulation and synthesis environments. First, you should not use the **work** library to contain packages that are shared between design units located in different source files. Although some simulation environments allow previously-compiled contents of the **work** library to be accessed at any time (such as during the separate compilation of a source file), this is not actually defined by the VHDL standard and may not work in other simulation and synthesis environments.

Some synthesis and simulation tools actually define the **work** library to be only those design units that are included in the source file currently being compiled. This is a simple rule of usage and is the recommended use of **work**.

To keep your use of libraries as simple as possible, it is recommended that you make consistent use of VHDL source file names and corresponding library file names, and avoid the use of **work** for all but the simplest packages.

Package Visibility

The **library** statement described in the previous section is used to load a library so that its contents are available when compiling a source file. However, the **library** statement does not actually make the contents of the specified library (the packages or other design units found in the library) visible to design units in the current source file. Visibility is created when you specify one or more **use** statements prior to the design units requiring access to items in the library.

The **use** statement is quite flexible. You can specify exactly which items within a package are to be made visible, specify that all items in a package are to be made visible, or specify that all items in all packages for a specific library are to be made visible. The following examples demonstrate some of the possible uses of **use** statements:

```
use mylib.my_package.all; -- All items in my_package are visible
use mylib.my_package.dff; -- Just using the dff procedure
use mylib.all;    -- Make everything in the library visible
```

In general, you will find that it is most convenient to place a **library** statement (one for each external library being used) at the beginning of your source file, and place **use** statements just prior to those design units requiring visibility of items in the library. To prevent compatibility problems as described above, you should avoid using **work** for shared packages or other design units that cross source file boundaries.

For clarity, it is recommended that you specify both the library and package name in your **use** statements, even if you are using all items in the library.

Components

Components are used to connect multiple VHDL design units (entity/architecture pairs) together to form a larger, hierarchical design. Using hierarchy can dramatically simplify your design description and can make it much easier to re-use portions of the design in other projects. Components are also useful when you want to make use of third-party design units, such as simulation models for standard parts, or synthesizable core models obtained from a company specializing in such models.

The following describes the relationship between the three design units in a shift and compare design example:

```
architecture structure of shiftcomp is
    component compare
```

```
        port(A, B: in bit_vector(0 to 7); EQ: out bit);
    end component;
    component shift
        port(Clk, Rst, Load: in bit;
               Data: in bit_vector(0 to 7);
               Q: out bit_vector(0 to 7));
    end component;
    signal Q: bit_vector(0 to 7);
begin
    COMP1: compare port map (Q, Test, Limit);
    SHIFT1: shift port map (Clk, Rst, Load, Init, Q);
end structure;
```

In this example, the two lower-level components (`shift` and `compare`) were instantiated in the higher-level module (`shiftcomp`) to form a hierarchy of design units. Each component instantiation is represented by a component name that is unique within the architecture or block.

Component instantiations are concurrent statements and therefore have no order-dependency. A design unit (such as this one) that includes only component instantiation statements can be thought of as a netlist, such as might be written (or generated) to represent the connections on a schematic.

Mapping of Ports

The previous example of component instantiation used positional association to describe how signals at the higher level (in this case `shiftcomp`) are to be matched with (i.e., connected to) ports of the entities in the lower-level modules (`shift` and `compare`).

Positional association is a quick and convenient way to describe the mapping of signals to ports in a component instantiation, but it can be error-prone. Consider, for example, what would have happened if the component instantiation for the shift module had been written as follows:

```
SHIFT1: shift port map (Rst, Clk, Load, Init, Q);
```

Because the `Rst` and `Clk` signals are of the same type (std_logic), the simulator or synthesis tool would accept this port mapping without complaint, and it would connect the reset signal to the clock and connect the clock to the reset. The circuit would not operate as expected, and the problem might be difficult to debug.

For this reason, it is generally recommend that you write component instantiations using an alternate form of port map called named association. Named association guarantees that the correct signals and ports are connected through the hierarchy, and it also gives you the ability to re-order the ports as needed.

The following example shows how the same component (a NAND gate) might be instanced using both positional and named association:

```
U1: nand2 port map (a, b, y);                  -- Positional association
U2: nand2 port map (a=>in1,b=>in2,y=>out1);    -- Named association
```

The special operator `=>` indicates exactly which lower-level ports (`a`, b and `y`, in this case) are to be connected to which higher-level signals (`in1`, `in2` and `out1`).

Named association also makes it possible to leave one or more lower-level ports unconnected using the keyword **open**, as shown below:

```
U2: count8 port map (C => Clk1, Rst => Clr, L => Load, D => Data,
                     Q => , Cin => open);
```

Generics

It is possible to pass instance-specific information other than actual port connections to an entity using a feature called generics. Generics are very useful for making design units more general-purpose or for annotating information (such as timing specifications) to an entity at the time the design is analyzed for simulation or synthesis.

The following example shows how generics can be used to create a parameterized model of a D-type flip-flop:

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

```
entity dffr is
    generic (wid: positive);
    port (Rst,Clk: in std_logic;
          signal D: in std_logic_vector(wid-1 downto 0);
          signal Q: out std_logic_vector(wid-1 downto 0));
end dffr;
architecture behavior of dffr is
begin
    process(Rst,Clk)
        variable Qreg: std_logic_vector(wid-1 downto 0);
    begin
        if Rst = '1' then
            Qreg := (others => '0');
        elsif Clk = '1' and Clk'event then
            for i in Qreg'range loop
                Qreg(i) := D(i);
            end loop;
        end if;
        Q <= Qreg;
    end process;
end behavior;
```

In this example, the `dffr` entity has a generic list in addition to a port list. This generic list contains one entry, a positive integer, that corresponds to the width of the `D` input and `Q` output. The architecture declaration uses a **for** loop in conjunction with the **generic** (`wid`) to describe the operation of the D-type flip-flops.

When instantiated in a higher-level design unit, a generic map must be provided in addition to the port map, as shown below:

```
architecture sample of reg is
    component dffr
        generic (wid: positive);
        port (Rst,Clk: in std_logic;
              signal D: in std_logic_vector(wid-1 downto 0);
              signal Q: out std_logic_vector(wid-1 downto 0));
    end component;
    constant WID8: positive := 8;
    constant WID16: positive := 16;
    constant WID32: positive := 32;
    signal D8,Q8: std_logic_vector(7 downto 0);
    signal D16,Q16: std_logic_vector(15 downto 0);
    signal D32,Q32: std_logic_vector(31 downto 0);
begin
    FF8:  dffr generic map(WID8)  port map(Rst,Clk,D8,Q8);
    FF16: dffr generic map(WID16) port map(Rst,Clk,D16,Q16);
    FF32: dffr generic map(WID32) port map(Rst,Clk,D32,Q32);
end sample;
```

The example shows how three instances of the `dffr` design unit can be created using different values for the generic.

Notes

When using named associations, it is a good idea to place each one on a separate line. This simplifies debugging because the debugger will identify the exact line where an association error occurred.

Configurations

Configurations are features of VHDL that allow large, complex design descriptions to be managed during simulation. One example of how you might use configurations is to construct two versions of a system-level design, one of which makes use of high-level behavioral descriptions of the system components, while a second version substitutes in a post-synthesis timing model of one or more components.

A configuration declaration is a primary design unit that defines the binding of some or all of the component instances in your design description to corresponding lower-level entities and architectures. The configuration declaration can form a simple parts list for your design, or it can be written to contain detailed information about how each component is "wired into" the rest of the design (through specific port mappings) and the values for generics being passed into each entity.

If you think of the configuration declaration as a parts list for your design, you can perhaps visualize it better as follows: consider a design description in which you have described an entity named `Board` with an architecture named `structure`. In the architecture structure you have described one instance (`U1`) of a component called `Chip`. Moving down in the hierarchy of your design, let's suppose that the entity `Chip` has been written with four alternative architectures named `A1`, `A2`, `A3` and `A4`. There are many reasons why you might have done this. For example, the default architecture might be the final synthesizable version of the chip, while the remaining three are versions intended strictly for high-level simulation.

There are many applications of configurations in simulation. For large projects involving many engineers and many design revisions, configurations can be used to manage versions and specify how a design is to be configured for system simulation, detailed timing simulation, and synthesis. Because simulation tools allow configurations to be modified and recompiled without the need to recompile other design units, it is easy to construct alternate configurations of a design very quickly without having to recompile the entire design.

Notes

Configurations are not generally supported in synthesis tools.

Test Benches

One of the primary reasons to use VHDL is its power as a test stimulus language. As logic designs become more complex, comprehensive, up-front verification becomes critical to the success of a design project. In fact, as you become proficient with simulation, you will quickly find that your VHDL simulator becomes your primary design development tool. When simulation is used right at the start of the project, you will have a much easier time with synthesis, and you will spend far less time re-running time-intensive processes, such as FPGA place-and-route tools and other synthesis-related software.

To simulate your project, you will need to develop an additional VHDL program called a test bench. (Some VHDL simulators include a command line stimulus language, but these features are no replacement for a true test bench.) Test benches emulate a hardware breadboard into which you will "install" your synthesizable design description for the purpose of verification. Test benches can be quite simple, applying a sequence of inputs to the circuit over time. They can also be quite complex, perhaps even reading test data from a disk file and writing test results to the screen and to a report file. A comprehensive test bench can, in fact, be more complex and lengthy (and take longer to develop) than the synthesizable circuit being tested. As you will begin to appreciate, test bench development will be where you make use of the full power of VHDL and your own skills as a VHDL “coder”.

Depending on your needs (and whether timing information related to your target device technology is available), you may develop one or more test benches to verify the design functionally (with no delays), to check your assumptions about timing relationships (using estimates or unit delays), or to simulate with annotated post-route timing information so you can verify that your circuit will operate in-system at speed.

During simulation, the test bench will be the top level of a design hierarchy. To the simulator, there is no distinction between those parts of the design that are being tested and the test bench itself.

When writing test benches, you will most likely use a broader range of language features. You may use records and multi-dimensional arrays to describe test stimuli, write loops, create subprograms to simplify repetitive actions, and/or use VHDL's text I/O features to read and write files of data.

A Simple Test Bench

The simplest test benches are those that apply some sequence of inputs to the circuit being tested (the Unit Under Test, or UUT) so that its operation can be observed in simulation. Waveforms are typically used to represent the values of signals in the design at various points in time. Such a test bench must consist of a component declaration corresponding to the unit under test, and a description of the input stimulus being applied to the UUT.

The following example demonstrates the simplest form of a test bench, and tests the operation of a NAND gate:

```
library ieee;   -- Load the ieee 1164 library
use ieee.std_logic_1164.all; -- Make the package 'visible'
use work.nandgate;    -- We'll use the NAND gate model from 'work'
-- The top level entity of the test bench has no ports...
--
entity testnand is
end testnand;
architecture stimulus of testnand is
    -- First, declare the lower-level entity...
    component nand
        port  (A,B: in std_logic;
               Y: out std_logic);
    end component;
    -- Next, declare some local signals to assign values to and observe...
    signal A,B: std_logic;
    signal Y: std_logic;
begin
```

```
    -- Create an instance of the comparator circuit...
    NAND1: nandgate port map(A => A,B => B,Y => Y);
    -- Now define a process to apply some stimulus over time...
    process
        constant PERIOD: time := 40 ns;
    begin
        A <= '1';
        B <= '1';
        wait for PERIOD;
        assert (Y = '0')
            report "Test failed!" severity ERROR;
        A <= '1';
        B <= '0';
        wait for PERIOD;
            assert (Y = '1')
            report "Test failed!" severity ERROR;
        A <= '0';
        B <= '1';
        wait for PERIOD;
            assert (Y = '1')
            report "Test failed!" severity ERROR;
        A <= '0';
        B <= '0';
        wait for PERIOD;
            assert (Y = '1')
            report "Test failed!" severity ERROR;
        wait;
    end process;
end stimulus;
```

Reading from the top of this test bench, the key areas of VHDL are:

- **Library** and **use** statements making the standard logic package available for use (the lower-level NAND gate model has been described using standard logic).
- An optional **use** statement referencing the lower-level design unit `nand` from the **work** library.
- An entity declaration for the test bench.
- An architecture declaration, containing:
- A component declaration corresponding to the unit under test.
- Signal declarations for `A`, `B`, and `Y`. These local signals will be used to (1) apply inputs to the unit under test, and (2) observe the behavior or the output during simulation.
- A component instantiation statement and corresponding **port map** statement that associates the top-level signals `A`, `B` and `Y` with their equivalent ports in the lower-level entity. The component name used is not significant; any valid component name could have been chosen.
- A **process** statement describing the inputs to the circuit over time. This process has been written without the use of a sensitivity list. It uses **wait** statements to provide a specific amount of delay (defined using constant `PERIOD`) between each new combination of inputs. **Assert** statements are used to verify that the circuit is operating correctly for each combination of inputs. Finally, a **wait** statement without any condition expression is used to suspend simulation indefinitely after the desired inputs have been applied. (In the absence of the final **wait** statement, the process would repeat forever, or for as long as the simulator had been instructed to run.)

Notes

Test benches do not generally include an interface (port) list, as they are the highest-level design unit when simulated.

Using Assert Statements

VHDL's **assert** statement provides a quick and easy way to check expected values and display messages from your test bench. An **assert** statement has the following general format:

```
assert condition_expression
    report text_string
    severity severity_level ;
```

When analyzed (either during execution as a sequential statement, or during simulator initialization in the case of a concurrent **assert** statement), the condition expression is evaluated. As in an **if** statement, the condition expression of an **assert** statement must evaluate to a boolean (true or false) value. If the condition expression is false (indicating the assertion failed), the text that you have specified in the optional **report** statement clause is displayed in your simulator's transcript (or other) window. The **severity** statement clause then indicates to the simulator what action (if any) should be taken in response to the assertion failure (or assertion violation, to use the language of the VHDL specification).

The severity level can be specified using one of the following predefined severity levels: NOTE, WARNING, ERROR, or FAILURE. The actions that result from the use of these severity levels will depend on the simulator you are using, but you can generally expect the simulator to display a file name and line number associated with the **assert** statement, keep track of the number of assertion failures, and print a summary at the end of the simulation run. **Assert** statements that specify FAILURE in their **severity** statement clauses will normally result in the simulator halting.

Displaying Complex Strings in Assert Statements

A common use of **assert** and **report** statements is to display information about signals or variables dynamically during a simulation run. Unfortunately, VHDL's built-in support for this is somewhat limited. The problem is twofold: first, the **report** clause only accepts a single string as its argument, so it is necessary to either write multiple **assert** statements to output multiple lines of information (as when formatting and displaying a table), or you must make use of the string concatenation operator & and the special character constant CR (carriage return) and/or LF (line feed) to describe a single, multi-line string as shown below:

```
assert false
    report "This is the first line of the message." & CR & LF &
               "This is the second line of the message.";
```

The second, more serious limitation of the **report** statement clause is that it only accepts a string, and there is no built-in provision for formatting various types of data (such as arrays, integers and the like) for display. This means that to display such data in an **assert** statement, you must provide type conversion functions that will convert from the data types you are using to a formatted string. The following example demonstrates how you might write a conversion function to display a std_logic_vector array value as a string of characters:

```
architecture stimulus of testfib is
    . . .
    function vec2str(vec: std_logic_vector) return string is
    variable stmp: string(vec'left+1 downto 1);
    begin
        for i in vec'reverse_range loop
            if (vec(i) = 'U') then
                stmp(i+1) := 'U';
            elsif (vec(i) = 'X') then
                stmp(i+1) := 'X';
            elsif (vec(i) = '0') then
                stmp(i+1) := '0';
            elsif (vec(i) = '1') then
                stmp(i+1) := '1';
            elsif (vec(i) = 'Z') then
```

```
                stmp(i+1) := 'Z';
            elsif (vec(i) = 'W') then
                stmp(i+1) := 'W';
            elsif (vec(i) = 'L') then
                stmp(i+1) := 'L';
            elsif (vec(i) = 'H') then
                stmp(i+1) := 'H';
            else
                stmp(i+1) := '-';
            end if;
        end loop;
    return stmp;
    end;
    . . .
    signal S: std_logic_vector(15 downto 0);
    signal S_expected: std_logic_vector(15 downto 0);
begin
    . . .
    process
    begin
        . . .
        assert (S /= S_expected)    -- report an error if different
                    report "Vector failure!" & CR & LF &
                    "Expected S to be  " & vec2str(S_expected) & CR & LF &
                    "but its value was " & vec2str(S)
                    severity ERROR;
```

In this example, a type conversion function has been written (`vec2str`) that converts an object of type std_logic_vector to a string of the appropriate format and size for display. As you develop more advanced test benches, you will probably find it useful to collect such type conversion functions into a library for use in future test benches.

Using Loops and Multiple Processes

Test benches can be dramatically simplified through the use of loops, constants and other more advanced features of VHDL. Using multiple concurrent processes in combination with loops can result in very concise descriptions of complex input and expected output conditions.

The following example demonstrates how a loop (in this case a **while** loop) might be used to create a background clock in one process, while other loops (in this case **for** loops) are used to apply inputs and monitor outputs over potentially long periods of time:

```
  Clock1: process
        variable clktmp: std_logic := '1';
    begin
        while done /= true loop
            wait for PERIOD/2;
            clktmp := not clktmp;
            Clk <= clktmp;
        end loop;
        wait;
    end process;
    Stimulus1: Process
```

```
    Begin
        Reset <= '1';
        wait for PERIOD;
        Reset <= '0';
        Mode <= '0';
        wait for PERIOD;
        Data <= (others => '1');
        wait for PERIOD;
        Mode <= '1';
        -- Check to make sure we detect the vertical sync...
        Data <= (others => '0');
        for i in 0 to 127 loop
            wait for PERIOD;
            assert (VS = '1')
                report "VS went high at the wrong place!" severity ERROR;
        end loop;
        assert (VS = '1')
             report "VS was not detected!" severity ERROR;
        -- Load in the test counter value to check the end of frame detection...
        TestLoad <= '1';
        wait for PERIOD;
        TestLoad <= '0';
        for i in 0 to 300 loop
            Data <= RandomData();
            wait for PERIOD;
        end loop;
        assert (EOF = '1')
             report "EOF was not detected!" severity ERROR;
        done <= true;
        wait;
    End Process;
End stimulus;
```

In this example, the process labeled `Clock1` uses a local variable (`clktmp`) to describe a repeating clock with a period defined by the constant `PERIOD`. This clock is described with a **while** loop statement, and it runs independent of all other processes in the test bench until the `done` signal is asserted true. The second process, `Stimulus1`, describes a sequence of inputs to be applied to the unit under test. It also makes use of loops – in this case **for** loops – to describe lengthy repeating stimuli and expected value checks.

Writing Test Vectors

Another approach to creating test stimuli is to describe the test bench in terms of a sequence of fixed input and expected output values. This sequence of values (sometimes called test vectors) could be described using multi-dimensional arrays or using arrays of records. The following example makes use of a record data type, `test_record`, which consists of the record elements `CE`, `Set`, `Din` and `CRC_Sum`. An array type (`test_array`) is then declared, representing an unconstrained array of `test_record` type objects. The constant `test_vectors`, of type `test_array`, is declared and assigned values corresponding to the inputs and expected output for each desired test vector.

The test bench operation is described using a **for** loop within a process. This **for** loop applies the input values `Set` and `Din` (from the test record corresponding to the current iteration of the loop) to the unit under test. (The `CE` input is used within the test bench to enable or disable the clock, and is not passed into the unit under test.) After a certain amount of time has elapsed (as

indicated by a **wait** statement), the CRC_Sum record element is compared against the corresponding output of the unit under test, using an **assert** statement.

```
library ieee;
use ieee.std_logic_1164.all;
use work.crc8s;     -- Get the design out of library 'work'
entity testcrc is
end testcrc;
architecture stimulus of testcrc is
    component crc8s
        port (Clk,Set,Din: in std_logic;
              CRC_Sum: out std_logic_vector(15 downto 0));
    end component;
    signal CE: std_logic;
    signal Clk,Set: std_logic;
    signal Din: std_logic;
    signal CRC_Sum: std_logic_vector(15 downto 0);
    signal vector_cnt: integer := 1;
    signal error_flag: std_logic := '0';
    type test_record is record -- Declare a record type
        CE: std_logic; -- Clock enable
        Set: std_logic;  -- Register preset signal
        Din: std_logic;  -- Serial Data input
        CRC_Sum: std_logic_vector (15 downto 0);   -- Expected result
    end record;
    type test_array is array(positive range <>) of test_record; -- Collect them
-- in an array
    -- The following constant declaration describes the test vectors to be
    -- applied to the design during simulation, and the expected result after a
    -- rising clock edge.
    constant test_vectors : test_array := (
         -- CE, Set, Din, CRC_Sum
('0', '1', '0', "----------------"),  -- Reset
('1', '0', '0', "----------------"),  -- 'H'
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "0010100000111100"),  -- x283C
('1', '0', '0', "----------------"),  -- 'e'
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '1', "----------------"),
```

```
('1', '0', '0', "----------------"),
('1', '0', '1', "1010010101101001"),  -- xA569
('1', '0', '0', "----------------"),  -- 'l'
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "0010000101100101"),  -- x2165
('1', '0', '0', "----------------"),  -- 'l'
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '0', "1111110001101001"),  -- xFC69
('1', '0', '0', "----------------"),  -- 'o'
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '0', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '1', "----------------"),
('1', '0', '1', "1101101011011010")   -- xDADA
    );
begin
    -- instantiate the component
    UUT: crc8s port map(Clk,Set,Din,CRC_Sum);
    -- provide stimulus and check the result
    testrun: process
        variable vector : test_record;
    begin
        for index in test_vectors'range loop
             vector_cnt <= index;
             vector := test_vectors(index); -- Get the current test vector
-- Apply the input stimulus...
             CE <= vector.CE;
             Set <= vector.Set;
             Din <= vector.Din;
        -- Clock (low-high-low) with a 100 ns cycle...
             Clk <= '0';
             wait for 25 ns;
             if CE = '1' then
                 Clk <= '1';
```

```
            end if;
            wait for 50 ns;
            Clk <= '0';
            wait for 25 ns;
        -- Check the results...
            if (vector.CRC_Sum /= "----------------"
                  and CRC_Sum  /= vector.CRC_Sum) then
               error_flag <= '1';
               assert false
                   report "Output did not match!"
                   severity WARNING;
            else
                error_flag <= '0';
            end if;
        end loop;
        wait;
    end process;
end stimulus;
```

Notes

VHDL 1076-1993 broadened the scope of bit string literals somewhat, making it possible to enter std_logic_vector data in non-binary forms as in the constant hexadecimal value x"283C".

Reading and Writing Files with Text I/O

The text I/O features of VHDL make it possible to open one or more data files, read lines from those files, and parse the lines to form individual data elements, such as elements in an array or record. To support the use of files, VHDL has the concept of a file data type, and includes standard, built-in functions for opening, reading from, and writing to file data types. The `textio` package, which is included in the standard library, expands on the built-in file type features by adding text parsing and formatting functions, functions and special file types for use with interactive (“std_input” and “std_output”) I/O operations, and other extensions.

The following example demonstrates how you can use the text I/O features of VHDL to read test data from an ASCII file, using the standard text I/O features.

```
-- Test bench, VHDL '93 style
--
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.fib;    -- Get the design out of library 'work'
entity testfib is
end entity testfib;
architecture stimulus of testfib is
    component fib is
       port (Clk,Clr: in std_logic;
             Load: in std_ulogic;
             Data_in: in std_ulogic_vector(15 downto 0);
             S: out std_ulogic_vector(15 downto 0));
    end component fib;
    function str_to_stdvec(inp: string) return std_ulogic_vector is
```

```
        variable temp: std_ulogic_vector(inp'range) := (others => 'X');
    begin
        for i in inp'range loop
            if (inp(i) = '1') then
                temp(i) := '1';
            elsif (inp(i) = '0') then
                temp(i) := '0';
            end if;
        end loop;
        return temp;
    end function str_to_stdvec;
    function stdvec_to_str(inp: std_ulogic_vector) return string is
        variable temp: string(inp'left+1 downto 1) := (others => 'X');
    begin
        for i in inp'reverse_range loop
            if (inp(i) = '1') then
                temp(i+1) := '1';
            elsif (inp(i) = '0') then
                temp(i+1) := '0';
            end if;
        end loop;
        return temp;
    end function stdvec_to_str;
    signal Clk,Clr: std_ulogic;
    signal Load: std_ulogic;
    signal Data_in: std_ulogic_vector(15 downto 0);
    signal S: std_ulogic_vector(15 downto 0);
    signal done: std_ulogic := '0';
    constant PERIOD: time := 50 ns;
begin
    UUT: fib port map(Clk=>Clk,Clr=>Clr,Load=>Load,
                      Data_in=>Data_in,S=>S);
    Clock: process
        variable c: std_ulogic := '0';
    begin
        while (done = '0') loop
            wait for PERIOD/2;
            c := not c;
            Clk <= c;
        end loop;
    end process Clock;
    Read_input: process
        file vector_file: text;
        variable stimulus_in: std_ulogic_vector(33 downto 0);
        variable S_expected: std_ulogic_vector(15 downto 0);
        variable str_stimulus_in: string(34 downto 1);
```

```
        variable err_cnt: integer := 0;
        variable file_line: line;
    begin
        file_open(vector_file,"tfib93.vec",READ_MODE);
        wait until rising_edge(Clk);
        while not endfile(vector_file) loop
            readline (vector_file,file_line);
            read (file_line,str_stimulus_in) ;
            assert false
                report "Vector: " & str_stimulus_in
                severity note;
            stimulus_in := str_to_stdvec (str_stimulus_in);
            wait for 1 ns;
            --Get input side of vector...
            Clr <= stimulus_in(33);
            Load <= stimulus_in(32);
            Data_in <= stimulus_in(31 downto 16);
            --Put output side (expected values) into a variable...
            S_expected := stimulus_in(15 downto 0);
            wait until falling_edge(Clk);
            -- Check the expected value against the results...
            if (S /= S_expected) then
                err_cnt := err_cnt + 1;
                assert false
                    report "Vector failure!" & lf &
                    "Expected S to be  " & stdvec_to_str(S_expected) & lf &
                    "but its value was " & stdvec_to_str(S) & lf
                    severity note;
            end if;
        end loop;
        file_close(vector_file);
        done <= '1';
        if (err_cnt = 0) then
            assert false
                report "No errors." & lf & lf
                severity note;
        else
            assert false
                report "There were errors in the test." & lf
                severity note;
        end if;
        wait;
    end process Read_input;
end architecture stimulus;
-- Add a configuration statement. This statement actually states the
-- default configuration, and so it is optional.
```

```
configuration build1 of testfib is
    for stimulus
        for DUT: fib use entity work.fib(behavior)
            port map(Clk=>Clk,Clr=>Clr,Load=>Load,
                     Data_in=>Data_in,S=>S);
        end for;
    end for;
end configuration build1;
```

This test bench reads lines from an ASCII file and applies the data contained in each line as a test vector to stimulate and test a simple Fibonacci sequence generator circuit. It begins with the by-now-familiar entity-architecture pair:

This test bench reads files of text “dynamically” during simulation, so the test bench does not have to be recompiled when test stimulus is added or modified. This is a big advantage for very large designs.

What does the test vector file that this test bench reads look like? The following example file content describes one possible sequence of tests that could be performed using this test bench:

```
10000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
00000000000000000000000000001101
00000000000000000000000000010101
00000000000000000000000000100010
00000000000000000000000000110111
00000000000000000000000001011001
00000000000000000000000010010000
00000000000000000000000011101001
00000000000000000000000101111001
00000000000000000000001001100010
00000000000000000000001111011011
00000000000000000000011000111101
00000000000000000000101000011000
00000000000000000001000001010101
00000000000000000001101001101101
00000000000000000010101011000010
00000000000000000100010100101111
00000000000000000110111111110001
00000000000000001011010100100000
00000000000000000100101000010001
00000000000000000000000000000001
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000101
00000000000000000000000000001000
```

This file could have been entered manually, using a text editor. Alternatively, it could have been generated from some other software package or from a program written in C, Basic or any other language. Reading text from files opens many new possibilities for testing and for creating interfaces between different design tools.

Reading Non-Tabular Data from Files

You can use VHDL's text I/O features to read and write many different built-in data types, including such data types as characters, strings, and integers. This is a powerful feature of the language that you will make great use of as you become proficient with the language.

VHDL's text I/O features are somewhat limited, however, when it comes to reading data that is not expressed as one of the built-in types defined in Standard 1076. The primary example of this is when you wish to read or write standard logic data types. In the previous example (the Fibonacci sequence generator), type conversion functions were used to read standard logic input data as characters. This method works fine, but it is somewhat clumsy. A better way to approached this common problem is to develop a reusable package of functions for reading and writing standard logic data. Writing a comprehensive package of such functions is not a trivial task. It would probably require a few days of coding and debugging.

VHDL Keywords

The following is a list of all keywords that exist in the standard VHDL language.

Keywords

Keyword: ABS
Keyword: ACCESS
Keyword: AFTER
Keyword: ALIAS
Keyword: ALL
Keyword: AND
Keyword: ARCHITECTURE
Keyword: ARRAY
Keyword: ASSERT
Keyword: ATTRIBUTE
Keyword: BEGIN
Keyword: BLOCK
Keyword: BODY
Keyword: BUFFER
Keyword: BUS
Keyword: CASE
Keyword: COMPONENT
Keyword: CONFIGURATION
Keyword: CONSTANT
Keyword: DISCONNECT
Keyword: DOWNTO
Keyword: ELSE
Keyword: ELSIF
Keyword: END
Keyword: ENTITY
Keyword: EXIT
Keyword: FILE
Keyword: FOR
Keyword: FUNCTION
Keyword: GENERATE
Keyword: GENERIC
Keyword: GROUP
Keyword: GUARDED
Keyword: IF
Keyword: IMPURE
Keyword: IN
Keyword: INERTIAL
Keyword: INOUT
Keyword: IS

Keyword: LABEL
Keyword: LIBRARY
Keyword: LINKAGE
Keyword: LITERAL
Keyword: LOOP
Keyword: MAP
Keyword: MOD
Keyword: NAND
Keyword: NEW
Keyword: NEXT
Keyword: NOR
Keyword: NOT
Keyword: NULL
Keyword: OF
Keyword: ON
Keyword: OPEN
Keyword: OR
Keyword: OTHERS
Keyword: OUT
Keyword: PACKAGE
Keyword: PORT
Keyword: POSTPONED
Keyword: PROCEDURE
Keyword: PROCESS
Keyword: PURE
Keyword: RANGE
Keyword: RECORD
Keyword: REGISTER
Keyword: REJECT
Keyword: REM
Keyword: REPORT
Keyword: RETURN
Keyword: ROL
Keyword: ROR
Keyword: SELECT
Keyword: SEVERITY
Keyword: SHARED
Keyword: SIGNAL
Keyword: SLA
Keyword: SLL
Keyword: SRA
Keyword: SRL
Keyword: SUBTYPE
Keyword: THEN
Keyword: TO
Keyword: TRANSPORT

Keyword: TYPE
Keyword: UNAFFECTED
Keyword: UNITS
Keyword: UNTIL
Keyword: USE
Keyword: VARIABLE
Keyword: WAIT
Keyword: WHEN
Keyword: WHILE
Keyword: WITH
Keyword: XNOR
Keyword: XOR

Keyword: ABS

The **abs** keyword is an absolute value operator which can be applied to any numeric type in an expression.

Example

```
Delta <= abs(A-B)
```

Keyword: ACCESS

The **access** keyword declares an access subtype. Access subtypes are used like pointers to refer to other objects. The objects which an access subtype can reference are array objects, record objects, and scalar type objects.

An access declaration includes the reserved word **access**, followed by a subtype.

Example

```
type AddressPtr is access RAM;
```

Keyword: AFTER

The **after** keyword is used in signal assignment statements to indicate a delay value before a signal assignment takes place.

A signal assignment statement containing an **after** clause includes – in this order – the name of the signal object, the reserved signal assignment symbol "**<=**", the optional keyword "**transport**", an expression specifying the value to be assigned to the signal, the reserved word "**after**", and the delay value (of type "time") after which the signal assignment is to take place.

If no **after** clause is present in a signal assignment statement, an implicit "**after** 0ns" clause is assumed.

Example

```
Clk <= not Clk after 50 ns;
...
Waveform <= transport '1' after 100 ps;
```

Keyword: ALIAS

An **alias** is an alternate name for an object. An alias is primarily used to create a slice (a one-dimensional array referring to all or part) of an existing array. An alias is not a new object, but only an alternate name for all or part of an existing object.

Example

```
alias LOWBYTE  :std_logic_vector(7 downto 0) is Data1(7 downto 0);
...
alias HIGHBYTE  :std_logic_vector(7 downto 0) is Data1(15 downto 8);
```

Keyword: ALL

The **all** keyword is used in the following ways:

- in a **use** statement, to make all the items in a package visible,
- in an **attribute** specification, to refer to all the names in a name class,

- in a **configuration** specification (**for**) statement, to refer to all instances of a component, and
- in a signal disconnection specification, to refer to all signal drivers of the same type.

Example

```
use ieee_std_logic_1164.all;
...
for DUT: compare use entity work.compare(compare1);
```

Keyword: AND

The **and** keyword represents a logical “and” operator which can be used in an expression. The expression “A and B” returns true only if both A and B are true.

Example

```
while error_flag /= '1'  and done /= '1' loop
```

Keyword: ARCHITECTURE

The **architecture** keyword defines the internal details of a design entity.

An architecture body defines the relationships between the input and output elements of the entity. An architecture body consists of a series of concurrent statements. An architecture body can also include processes, functions, and procedures, each of which may include sequential statements. Although the statements inside a process, for example, are executed sequentially, the process itself is treated within the architecture body as a concurrent statement.

A given architecture can be associated with only one entity. However, a given entity may have more than one architecture body.

An **architecture** statement includes – in this order – the following:

- the reserved word “**architecture**”, followed by :
  (a) the name of the architecture,
  (b) the reserved word “**of**”,
  (c) the entity name, and
  (d) the reserved word “**is**”,
- a declarations section,
- the reserved word “**begin**”,
- the architecture body (a series of concurrent statements as described above), and
- the reserved word “**end**”, followed optionally by the name of the architecture from (1)(a) above.

Example

```
architecture sample_architecture of compare is
begin
GT <= '1' when A > B else '0';
LT <= '1' when A < B else '0';
EQ <= '1' when A = B else '0';
end sample_architecture;
```

Keyword: ARRAY

The **array** keyword is used to declare an array data type. An array is an object containing a collection of elements that are all of the same type.

An array can be either constrained or unconstrained. A constrained array is defined with an index defining the number of array elements. In an unconstrained array, the number of elements in the array is specified in the array's object declaration, or the index definition for the array may be given in a subtype declaration. Arrays may be one-dimensional (single index) or multi-dimensional (multiple indices).

An array definition includes – in this order – the following:

- the reserved word “**array**”, followed by a definition(s) of the elements in the array, and

- the reserved word "**of**", followed by the subtype of the array's elements.

Example

```
type DataWord is array (15 downto 0) of DataBit;
--Constrained
...
type BigWord is array (integer range <>) of DataBit;
Unconstrained
```

Keyword: ASSERT

The **assert** keyword indicates the beginning of an assert statement. An assert statement checks to see if a given condition is true and, if the statement is not true, performs some action.

An assert statement includes two options, either or both of which may be used:

- **report** – which displays a user-defined message if the given condition is false, and
- **severity** – which allows the user to choose a severity level if the given condition is false.

The four possible severity levels are: Note, Warning, Error, and Failure. The value of severity is typically used to control the actions of a simulation in the event the given condition is false. For example, a severity level of Failure may be used to stop the simulation.

Example

```
assert (S = S_expected)
report "S does not match the expected value!"
severity Error;
```

Keyword: ATTRIBUTE

An **attribute specification** describes a characteristic of a given object. An attribute is most often used to get additional information about an object. For example, an attribute may be used to find the width of an array or to determine if a signal is in transition (i.e., has an event occurring on it).

```
if Clk'event then...
...
W = Data'width;
```

An attribute can be a constant, function, range, signal, type, or value. User-defined attributes are always constants, no matter what type. The other five possibilities – function, range, signal, type, and value – are pre-defined attributes.

An **attribute declaration** is used to declare an attribute name and its type. It includes – in this order – the reserved word "**attribute**", the name of the attribute, and the attribute's type.

```
attribute enum_encoding: string;
```

An attribute specification assigns a value to the attribute. It includes – in this order – the reserved word "**attribute**", the attribute's name, the reserved word "**of**", an item name (which can be an architecture, component, configuration, constant, entity, function, label, package, procedure, signal, subtype, type, or variable), the name class of the item (e.g., architecture, component, configuration, etc.), the reserved word "**is**", and an expression.

Example

```
attribute enum_encoding of StateReg is
"0001 0011 0010 0110 0100 1100 1000";
```

Notes

An attribute name must be declared in an attribute declaration before it can be used in an attribute specification.

Keyword: BEGIN

The **begin** keyword specifies the start of the main body of statements in an architecture, function, procedure, process or block.

Example

```
architecture example of control_stmts is
begin
    m <= b when a else  c;
end example;
```

Keyword: BLOCK

Block is a concurrent statement used to represent a portion of a design. Block statements may also include an optional Guard feature which allows the user to disable signal drivers within the block when a specified Guard condition is false.

A block statement includes – in this order – the following:

- block label,
- the reserved word "**block**",
- optionally, a Boolean guard expression (for example, `TESTCOUNT<5`),
- a block header, which specifies the interface of the block with its environment,
- a block declarations section,
- the reserved word "**begin**",
- the block statements, and
- the reserved words "**end block**", optionally followed by the block label (which, if used, must be the same as the block label declared above).

When a guard expression is used, a signal driver can be disabled by inserting the reserved word "**guarded**" at the beginning of the right side of the signal driver statement. For example, based on the example in (3) above, the block statement:

```
SAMPLE <= guarded D;
```

will cause the signal `SAMPLE` to take on the value of `D` only when `TESTCOUNT<5`. Otherwise, no action on that assignment statement will be taken.

Example

```
TESTPARITY: block
    signal Atmp,Btmp;  -- Local signals
begin
    Atmp <= gen_parity(A);
    Btmp <= gen_parity(B);
    ParityEQ <= '1' when Atmp = Btmp else '0';
end block TESTPARITY;
```

Keyword: BODY

The **body** keyword is used in conjunction with the **package** keyword to declare a package body. A package body specifies the definitions of the various subprograms (components, functions, etc.) that are declared by its associated package declaration.

The package body must have the same name as the package declaration. Only one package body can be associated with each package declaration.

Example

```
package body conversions is
    function to_unsigned (a: std_ulogic_vector) return
        integer is
        ...
    begin
```

```
        ...
    end to_unsigned;
    function to_vector (size: integer; num: integer) return
        std_ulogic_vector is
        ...
    begin
        ...
    end to_vector;
end conversions;
```

Keyword: BUFFER

Buffer is one of five possible modes for an interface port. (The other four are **in**, **out**, **inout**, and **linkage**.) The buffer mode indicates a port which can be used for both input and output, and it can have only one source. A buffer port can only be connected to another buffer port or to a signal that also has only one source.

Example

```
entity ent5 is
  port (clk,reset : in std_logic;
     p : buffer std_logic_vector(1 downto 0));
end ent5 ;
```

Keyword: BUS

Bus specifies one of two kinds of signals used in a signal declaration (the other is **register**). A bus signal represents a hardware bus and defaults to a user-specified value when all of the signal's drivers are turned off.

Example

```
entity tbuf is
   port (enable: boolean; a: bundle; m: out bundle bus);
end tbuf;
```

Keyword: CASE

Case is a sequential statement used within a process, procedure or function that selects and executes one statement sequence among a list of alternatives, based on the value of a given expression. The expression must be of a discrete type or a one-dimensional array type.

A **case** statement includes – in this order – the following:

- the reserved word “**case**”,
- the expression to be evaluated,
- the reserved word “**is**”,
- the reserved word “**when**” followed by a choice and the sequence of statements to be executed if the expression evaluates to be that choice,
- optionally, subsequent “**when**” statements similar to above,
- optionally, the reserved words “**when others**” followed by the sequence of statements to be executed if the expression evaluates to be any value other than those specified in the “**when**” statements above,
- the reserved words “**end case**”.

Because the case statement chooses one and only one alternative for execution, all possible values for the expression must be covered in “**when**” statements.

A **case** statement is distinguished from a chain of **if-then-else** statements in that no priority is implied for the conditions specified.

Example

```
case current_state is
     when IDLE =>
       if start_key = '1' then
       current_state <= READ0;
       end if;
     when READ0 =>
       current_state <= READ1;
     when READ1 =>
       current_state <= READX;
     when READX =>
       current_state <= WRITE0;
     when WRITE0 =>
       current_state <= WRITEX;
     when WRITEX =>
       current_state <= IDLE;
end case;
```

Keyword: COMPONENT

A **component** declaration is used to define the interface to a lower-level design entity. The component may then be included in a component instantiation statement which itself is included in an architecture body, thus allowing one entity to be used as part of another entity. The **component** declaration must be placed in the declaration section of the architecture body, or in a package visible to the architecture.

Example

```
component my_adder
    port(A,B,Cin: in std_ulogic;
             Sum,Cout: out std_ulogic);
end component;
```

Keyword: CONFIGURATION

A declaration used to create a configuration for an entity. A **configuration** declaration for a given entity binds one architecture body to the entity and can bind components of architecture bodies within that entity to other entities. In a given configuration declaration for an entity, only one architecture body can be bound to that entity. However, one entity can have many configurations.

Example

```
configuration this_build of adder is
use work.all;
for structure
  for A1,A2,A3: AddBlock
    use entity FullAdd(behavior);
  end for;
end for;
end this_build;
```

Keyword: CONSTANT

The **constant** keyword declares a constant of a type specified in the constant declaration.

A constant declaration includes – in this order – the reserved word “**constant**”, the name of the constant, the optional reserved word “**in**”, the type of the constant, and, optionally, an expression for the value of the constant.

If an expression for the value of the constant is not included in the constant declaration, then the constant is referred to as a deferred constant. A deferred constant may only be included in a package declaration, while the complete constant declaration, including the expression which defines its value, must be included in the package body.

Example

```
constant RESET: std_ulogic_vector(7 downto 0) := "00000000";
...
constant PERIOD: time := 80 ns;
```

Keyword: DISCONNECT

The **disconnect** keyword specifies the time delay to disconnect the guarded feature of a signal which is part of a guarded signal statement.

A disconnect statement includes – in this order – the reserved word "**disconnect**", the name of the guarded signal, the guarded signal's type, the reserved word "**after**", and a time expression specifying the time after which the guard feature should be disconnected.

In place of the guarded signal's name, the reserved words "**others**" or "**all**" may be used. "**Others**" refers to all other signal statements in the immediately enclosing declarative region which have not been specified in a separate **disconnect** statement. "**All**" refers to all other signal statements in the declarative region.

Example

```
architecture sample_architecture of test1 is
signal input_data_bus : resolved_word bus;
disconnect input_data_bus : resolved_word after 6ns;
begin
...
end sample_architecture;
```

Notes

A given signal driver can have only one **disconnect** statement.

Keyword: DOWNTO

The **downto** keyword is used to indicate a descending range in a range statement or other statement which includes a range (for example, an array type declaration). (The reserved word "**to**" is used to indicate an ascending range.)

Example

```
signal A0,A1: std_logic_vector(15 downto 0);
```

Keyword: ELSE

The **else** keyword is used to identify the final alternative in an **if** or **when** statement.

Example

```
if A > B then
    Compare <= GT;
elsif A < B then
    Compare <= LT;
else
    Compare <= EQ;
end if;
```

Keyword: ELSIF

The **elsif** keyword is used to identify an interim alternative in an **if** statement.

Example

```
if A > B then
    Compare <= GT;
elsif A < B then
    Compare <= LT;
else
    Compare <= EQ;
end if;
```

Keyword: END

The **end** keyword specifies the end of an architecture, configuration, entity, function, package, package body or procedure.

Example

```
architecture sample_architecture of compare is
begin
GT <= '1' when A > B else '0';
LT <= '1' when A < B else '0';
EQ <= '1' when A = B else '0';
end sample_architecture;
```

Notes

The **end** keyword is also used in conjunction with other keywords to signify the end of a specific declaration or statement. The following sections illustrate examples of such usage:

END BLOCK

```
TESTPARITY: block
    signal Atmp,Btmp;  -- Local signals
begin
    Atmp <= gen_parity(A);
    Btmp <= gen_parity(B);
    ParityEQ <= '1' when Atmp = Btmp else '0';
end block TESTPARITY;
```

END CASE

```
case current_state is
     when IDLE =>
       if start_key = '1' then
       current_state <= READ0;
       end if;
     when READ0 =>
       current_state <= READ1;
     when READ1 =>
       current_state <= READX;
     when READX =>
       current_state <= WRITE0;
     when WRITE0 =>
       current_state <= WRITEX;
     when WRITEX =>
       current_state <= IDLE;
end case;
```

END COMPONENT

```
component my_adder
    port(A,B,Cin: in std_ulogic;
            Sum,Cout: out std_ulogic);
end component;
```

END FOR

```
configuration build1 of testfib is
    for stimulus
        for DUT: fib use entity work.fib(behavior)
            port map(Clk=>Clk,Clr=>Clr,Load=>Load,
                      Data_in=>Data_in,S=>S);
        end for;
    end for;
end configuration build1;
```

END GENERATE

```
G: for I in 0 to (WIDTH - 2) generate
    -- This generate statement creates the first
    -- XOR gate in the series...
    G0: if I = 0 generate
        X0: xor2 port map(A => D(0), B => D(1), Y => p(0));
    end generate G0;
    -- This generate statement creates the middle
    -- XOR gates in the series...
    G1: if I > 0 and I < (WIDTH - 2) generate
        X0: xor2 port map(A => p(i-1), B => D(i+1), Y => p(i));
    end generate G1;
    -- This generate statement creates the last
    -- XOR gate in the series...
    G2: if I = (WIDTH - 2) generate
        X0: xor2 port map(A => p(i-1), B => D(i+1), Y => ODD);
    end generate G2;
end generate G;
```

END IF

```
if A > B then
    Compare <= GT;
elsif A < B then
    Compare <= LT;
else
    Compare <= EQ;
end if;
```

END LOOP

```
loop1: for state in stateval loop
    if current_state = state  then
         valid_state <= true;
    end if;
end loop loop1;
```

```
...
process
begin
    while error_flag /= '1'  and done /= '1' loop
        Clock <= not Clock;
        wait for CLK_PERIOD/2;
    end loop;
end process;
```

END PROCESS

```
reg: process(Rst,Clk)
        variable Qreg: std_ulogic_vector(0 to 7);
begin
        if Rst = '1' then  -- Async reset
            Qreg := "00000000";
        elsif rising_edge(Clk) then
            if Load = '1' then
                Qreg := Data;
            else
                Qreg := Qreg(1 to 7) & Qreg(0);
            end if;
        end if;
        Q <= Qreg;
end process;
```

END RECORD

```
type test_record is record
      CE: std_ulogic;  -- Clock enable
      Set: std_ulogic;
      Din: std_ulogic;
      CRC_Sum: std_ulogic_vector (15 downto 0);
end record;
```

END UNITS

```
type time isrange -2_147_483_647 to 2_147_483_647
    units
        fs;
        ps  = 1000 fs;
        ns  = 1000 ps;
        us  = 1000 ns;
        ms  = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr  = 60 min;
    end units;
```

Keyword: ENTITY

An **entity** declaration used to describe the interface of a design entity.

A design entity is an abstract model of a digital system. A design entity includes:

- an entity declaration (which specifies the name of the entity and its interface ports), and
- at least one architecture body (which models the internal workings of the digital system).

An **entity** declaration includes – in this order – the reserved word "**entity**", the entity's name, the reserved word "**is**", the following optional statements:

- the reserved word "**generic**" followed by a list of generics and their types,
- the reserved word "**port**" followed by a list of interface port names and their types,
- any declaration of entity items,
- the reserved word "**begin**" followed by appropriate entity declaration statements, and
- non-optionally, the reserved word "**end**" followed (optionally) by the entity's name.

The ports of an entity are visible within the architecture(s) of the entity, and may be referenced (have their values read, or have values assigned to them, depending on their mode) as signals within the architecture(s).

Declarations made within an **entity** statement are visible within the corresponding architecture(s).

Example

```
entity Mux is
generic(RISE, FALL: time := 0 ns);
port(A,B: in std_ulogic;
        Sel: in  std_ulogic;
        Y: out std_ulogic);
end Mux;
```

Keyword: EXIT

The **exit** keyword is a sequential statement used in a loop to cause execution to jump out of the loop.

An exit statement can only be used in a loop and can include an optional **when** condition. An exit statement includes – in this order – the reserved word "**exit**", an optional loop identifier (if no identifier is given, the exit statement is applied to the loop in which the exit statement occurs), and, optionally, the reserved word "**when**" followed by the condition under which the exit action is to be executed.

Example

```
for idx in vectors'range loop
  apply_vector(vec(idx));
  wait for PERIOD;
  if done = '1' then
    exit;
  end if;
end loop;
```

Keyword: FILE

The **file** keyword declares a file.

A file declaration includes – in this order – the reserved word "**file**", the name of the file (as used by the program), the subtype indicator (which must define a file subtype), the reserved word "**is**", on optional mode indicator (which must be either "**in**" or "**out**"), and the file's external name (which must be a string expression and is surrounded by quote marks). If no mode is specified, the default is "**in**".

Example

```
file vector_file: text is in "VECTOR.DAT";
```

Keyword: FOR

The **for** keyword is a statement used to identify:

- a block specification in a **block** configuration,
- a component specification in a **component** configuration,
- a parameter specification in a **generate** statement,
- a parameter specification in a **loop** statement, or

- a time expression in a **wait** statement.

Example

```
configuration build1 of testfib is
    for stimulus
        for DUT: fib use entity work.fib(behavior)
            port map(Clk=>Clk,Clr=>Clr,Load=>Load,
                     Data_in=>Data_in,S=>S);
        end for;
    end for;
end configuration build1;
```

Keyword: FUNCTION

A **function** statement defines a group of sequential statements that return a single value.

A function specification includes – in this order – the reserved word "**function**", the function's name, a parameter list (which can only include **constants** and **signal** objects, and must all be of mode **in**), the reserved word "**return**", and the type of the value to be returned by the function.

Example

```
function to_unsigned (a: std_ulogic_vector)
           return integer is
        alias av: std_ulogic_vector (1 to a'length) is a;
        variable ret,d: integer;
begin
        d := 1;
        ret := 0;
        for i in a'length downto 1 loop
            if (av(i) = '1') then
                ret := ret + d;
            end if;
            d := d * 2;
        end loop;
        return ret;
end to_unsigned;
```

Keyword: GENERATE

The **generate** keyword is used to do one of the following:

- replicate a set of concurrent statements (a **for**-generation), or
- selectively execute a set of concurrent statements if a specified condition is met (an **if**-generation).

A generate statement used to replicate a set of concurrent statements includes – in this order – the following:

- a label for the generate, followed by the reserved word "**for**", followed by a parameter specification for the "**for**",
- the reserved word "**generate**",
- the concurrent statements to be replicated,
- the reserved words "**end generate**".

A generate statement used to selectively execute a set of concurrent statements includes – in this order – the following:

- a label for the generate, followed by the reserved word "**if**", followed by the condition for the "**if**",
- the reserved word "**generate**",
- the concurrent statements to be selectively executed if the test condition is true,
- the reserved words "**end generate**".

Example

```
G: for I in 0 to (WIDTH - 2) generate
    -- This generate statement creates the first
    -- XOR gate in the series...
    G0: if I = 0 generate
        X0: xor2 port map(A => D(0), B => D(1), Y => p(0));
    end generate G0;
    -- This generate statement creates the middle
    -- XOR gates in the series...
    G1: if I > 0 and I < (WIDTH - 2) generate
        X0: xor2 port map(A => p(i-1), B => D(i+1), Y => p(i));
    end generate G1;
    -- This generate statement creates the last
    -- XOR gate in the series...
    G2: if I = (WIDTH - 2) generate
        X0: xor2 port map(A => p(i-1), B => D(i+1), Y => ODD);
    end generate G2;
end generate G;
```

Keyword: GENERIC

The **generic** keyword used in a component or configuration to define constants whose values may be controlled by the environment.

A generic statement includes – in this order – the reserved word "**generic**", followed by a list of declarations for the generics being defined.

Example

```
generic(RISE, FALL: time := 0 ns);
```

Keyword: GROUP

The **group** keyword is used to define a group template or specific group. Groups may be used to give a name to a collection of named entities.

Group Template Declaration

A group template declaration includes – in this order – the reserved word "**group**" followed by a group name, the reserved word "**is**", and a list of classes enclosed in parentheses.

Example

```
group signal_pair is (signal1, signal2);  -- group of two signals
```

Group Declaration

A group declaration includes – in this order – the reserved word "**group**" followed by a group name, the character "**:**", a group template name, and a list of named entities enclosed in parentheses.

Example

```
group G1: signal_pair(Clk1,Clk2);
```

Keyword: GUARDED

The **guarded** keyword is used to limit the execution of a signal statement within a block when the block includes a guard statement.

Example

```
use ieee.std_logic_1164.all;
entity latch is
    port( D, LE: in std_logic;
          Q, QBar: out std_logic);
end latch;
architecture mylatch of latch is
begin
    L1: block (LE = '1')
    begin
        Q <= guarded D after 5 ns;
        QBar <= guarded not(D) after 7 ns;
    end block L1;
end mylatch;
```

Keyword: IF

The **if** keyword is a sequential statement used for describing conditional logic.

Example

```
if A > B then
    Compare <= GT;
elsif A < B then
    Compare <= LT;
else
    Compare <= EQ;
end if;
```

Notes

The condition expression of an **if** statement must be a Boolean logic expression (meaning that it must evaluate to a True or False value).

If statements are sequential and may only be used in processes, procedures or functions.

Keyword: IMPURE

The **impure** keyword is used to declare a function that may return a different value given the same actual parameters, due to side effects.

Impure functions have access to a broader class of values than pure functions, and can modify objects that are outside their own scope.

Example

```
impure function HoldCheck (Clk, Data) return Boolean;
```

Keyword: IN

The **in** keyword can be used in two different ways depending on the context:

- One of five possible modes for an interface port (the other four are **inout**, **out**, **buffer**, and **linkage**); the **in** mode indicates a port which can be used only for input; and
- An optional word in a **constant** declaration.

Example

```
component COUNT4EN
    port ( CLK,RESET,EN : in  std_logic;
           COUNT : out std_logic_vector(3 downto 0)
    );
end component;
```

Keyword: INERTIAL

The **inertial** keyword is used to specify that a delay is inertial. In the absence of an **inertial** or **transport** keyword, the delay is assumed to be inertial.

Example

```
Qout <= A and B inertial after 12 ns;
```

Keyword: INOUT

The **inout** keyword specifies one of five possible modes for an interface port. (The other four are **in**, **out**, **buffer**, and **linkage**.) The **inout** mode indicates a port which can be used for both input and output.

Example

```
procedure jkff (signal Rst, Clk: in std_logic;
                signal J, K: in std_logic;
                signal Q,Qbar: inout std_logic) is
begin
    if Rst = '1' then
        Q <= '0';
    elsif Clk = '1' and Clk'event then
        if J = '1' and K = '1' then
            Q <= Qbar;
        elsif J = '1' and K = '0' then
            Q <= '1';
        elsif J = '0' and K = '1' then
            Q <= '0';
        end if;
    end if;
    Qbar <= not Q;
end jkff;
```

Keyword: IS

The **is** keyword is used as part of the syntax when declaring, for example, an architecture, case statement, configuration, entity, function, package, package body, procedure, subtype or type.

Example

```
architecture arch2 of my_design is
    signal Bus1, Bus2: std_logic_vector(7 downto 0);
begin
    . . .
end declare;
```

Keyword: LABEL

The **label** keyword is used to specify a label name in an **attribute** statement.

Example

```
attribute CHIP_PIN_LC of u0 : label is "LAB2";
attribute CHIP_PIN_LC of u2 : label is "LAB7";
```

Keyword: LIBRARY

The **library** keyword identifies a library. The library statement is a context clause used to identify libraries from which design units can be referenced.

A library statement includes – in this order – the reserved word "**library**" followed by a list of library logical names.

Example

```
library std_logic_1164;   -- Use the IEEE 1164 standard library
```

Notes

Using a **library** clause makes a named library visible to the working environment. However, to use a design unit from within that library, a "**use**" statement must also be included specifying the design unit to be used.

All design units automatically include the following implicit **library** clause:

```
library STD, WORK;
```

Keyword: LINKAGE

The **linkage** keyword specifies one of five possible modes for an interface port. (The other four are **in**, **out**, **inout**, and **buffer**.)

The **linkage** mode indicates a port which can be used for both input and output, and it can only correspond to a signal.

Keyword: LITERAL

The **literal** keyword is used in group template declarations.

Keyword: LOOP

The l**oop** keyword executes a series of sequential statements multiple times.

A loop statement can include either:

- a "**while**" iteration scheme,
- a "**for**" iteration scheme, or
- no iteration scheme.

A loop statement using a "**while**" iteration scheme includes – in this order – the following:

- an optional loop label,
- the reserved word "**while**", followed by the condition which controls whether the series of sequential statements within the loop is executed, followed by the reserved word "**loop**",
- the series of sequential statements to be executed if the test condition evaluates to be True,
- the reserved words "**end loop**", followed by an optional loop label (which, if used, must be the same as the loop label declared above.

Example1

```
while (I < DBUS'length) loop
...
I := I + 1;
end loop;
```

A loop statement using a "**for**" iteration scheme includes – in this order – the following:

- an optional loop label,
- the reserved word "**for**", followed by a parameter specification for the "**for**", followed by the reserved word "**loop**",
- the series of sequential statements to be executed for the instances defined in the parameter specification,
- the reserved words "**end loop**", followed by an optional loop label, which, if used, must be the same as the loop label declared above.

Example2

```
for I in 0 to DBUS'length - 1 loop
...
end loop;
```

A loop statement with no iteration scheme includes – in this order – the following:

- an optional loop label,
- the reserved word "**loop**",

- the series of sequential statements to be executed,
- the reserved words "**end loop**", followed by an optional loop label, which, if used, must be the same as the loop label declared above.

A loop statement with no iteration scheme continues to execute until some action causes execution to cease. This could be done using an "**exit**" statement, a "**next**" statement, or a "**return**" statement within the loop.

Example3

```
loop
exit when I = DBUS'length;
I := I + 1;
end loop;
```

Keyword: MAP

The **map** keyword is used in conjunction with the **port** and **generic** keywords to declare a port map or generic map respectively.

Port Map

A **port map** statement is used to associate signals of ports within a block to ports defined outside the block.

For example, suppose a given entity includes an architecture, and the architecture includes a block. A **port map** statement could be used to set the value of an entity port (which was defined by a "**port**" statement in the entity declaration), equal to the value of a block port (which was defined by a "**port**" statement in the block).

A **port map** statement includes – in this order – the reserved words **port** and **map** followed by an association list (e.g., "`LOCAL_PORT => GLOBAL_PORT`"). The association list may use positional or named association, as shown in the following examples. Ports may be left unconnected through the use of the **open** keyword.

Example

```
U1: And2 port map (IN1, IN2, OUT1);
U1: And2 port map (A => IN1, B => IN2, Y => OUT1);
A18: AddBlk port map (A => A1, B => A1, S => Sum, Cout =>open);
```

Generic Map

A **generic map** statement is used to associate values of constants within a block to constants defined outside the block.

For example, suppose a given entity includes an architecture, and the architecture includes a block. A generic map statement could be used to set the value of an entity constant (which was defined by a "**generic**" statement in the entity declaration), equal to the value of a block constant (which was defined by a "**generic**" statement in the block).

A generic map statement includes – in this order – the reserved words **generic** and **map** followed by an association list (e.g., "LOCAL => GLOBAL").

Example

```
U1: And2
    generic map (RISE_TIME => 2 ns, FALL_TIME => 2 ns);
    port map (A => IN1, B => IN2, Y => OUT1);
```

Keyword: MOD

The **mod** keyword is a modulus operator that can be applied to integer types. The result of the expression "`A mod B`" is an integer type and is defined to be the value such that:

- the sign of (`A mod B`) is the same as the sign of `B`, and
- `abs (A mod B) < abs (B)`, and
- `(A mod B) = (A * (B - N))` for some integer `N`.

Example

```
begin
  for i in 0 to (bits-1) loop
    if ((tmp mod 2) = 1) then
      out_vec(i) := '1';
    end if;
    tmp := tmp/2;
  end loop;
  return out_vec;
end int_2_v;
```

Keyword: NAND

Nand is a logical “not and” operator which can be used in an expression. It produces the opposite of the logical negation of the “**and**” operator.

The expression “`A nand B`” returns True when

- `A` is false, or
- `B` is false, or
- both `A` and `B` are false.

Example

```
begin
    Y <= (A nand B) and Sel;
    Y <= (A nor B) and not Sel;
end;
```

Keyword: NEW

The **new** keyword is used to create an object of a specified type and return an access value that refers to the created object.

A new statement includes – in this order – the allocator (which, when evaluated, refers to the created object), followed by the reserved symbol “:=“, followed by the reserved word “**new**”, followed by the type of the object being created, and optionally followed by the reserved “**new**” and an expression for the initial value of the object being created.

Example

```
count:= new natural;
```

Keyword: NEXT

Next is a statement allowed within a loop that causes the current iteration of the loop to be terminated and cycles the loop to the beginning of its next iteration.

A next statement includes – in this order – the reserved word “**next**”, an optional loop label (which must be the same as the label of the loop in which the next statement occurs), and, optionally, the reserved word “**when**” followed by a condition which, when True, causes the **next** statement to be executed.

If a “**when**” clause is not included, a “**next**” statement is executed as soon as it is encountered.

Example

```
L1 : for i in 0 to 9 loop
  L2 : for j in opcodes loop
         for k in 4 downto 2 loop -- loop label is optional
           if k = i next L2;      -- go to next L2 loop
         end loop;
       exit L1 when j = crash; -- exit loop L1
  end loop;
end loop;
```

Keyword: NOR

Nor is a logical “not or” operator which can be used in an expression. It produces the logical negative of the “**or**” operator. The expression “`A nor B`” returns True only when both `A` and `B` are false.

Example

```
begin
    Y <= (A nand B) and Sel;
    Y <= (A nor B) and not Sel;
end;
```

Keyword: NOT

Not is a logical “not” operator which can be used in an expression. The expression “`not A`” returns True if `A` is false and returns False if `A` is true.

Example

```
begin
    Y <= not (A and B) and Sel;
    Y <= not (A or B) and not Sel;
end arch4;
```

Keyword: NULL

Null is a statement that performs no action.

The **null** statement can be used in situations where it is necessary to explicitly specify that no action is needed. For example, a **null** statement may be useful in a **case** statement where all alternatives must be specified but where no action may be required for some alternatives.

Example

```
D1 <= '0'; -- Default values...
Strobe <= '0';
Rdy <= '0';
case current_state is
    when S0 =>
      D1 <= '1';
    when S1 =>
      Strobe <= '1';
    when S2 =>
      Rdy <= '1';
    when others =>
      null;
end case;
```

Keyword: OF

The **of** keyword is used as part of the syntax when declaring, for example, an architecture, array, attribute or configuration.

Example

```
architecture arch1 of my_design is
    signal Q: std_logic;
begin
    . . .
end arch1;
```

Keyword: ON

The **on** keyword is used as part of a wait statement to temporarily suspend a process until an event occurs which affects one or more specified signals. The process will resume when any or all of the listed signals change.

Example

```
Example: process is
begin
   sum <= a xor b after time_period;
   carry <= a and b after time_period;
   wait on a, b;
end process Example;
```

Keyword: OPEN

The **open** keyword is used in an association list (within a component instantiation statement) to indicate a port that is not connected to any signal.

Example

```
U2: count8 port map (C => Clk1, Rst => Clr, L => Load, D => Data,
                     Q => , Cin => open);
```

Keyword: OR

Or is a logical "or" operator that can be used in an expression. The expression "`A or B`" returns True if

- `A` is true, or
- `B` is true, or
- both `A` and `B` are true.

Example

```
begin
    Y <= not (A and B) and Sel;
    Y <= not (A or B) and not Sel;
end;
```

Keyword: OTHERS

The **others** keyword is used to specify all remaining elements in:

- an element association (in an aggregate),
- an attribute specification,
- a configuration specification,
- a disconnection specification,
- case statement, or
- a selected assignment

Example

```
when others => null;
...
constant ZERO: std_ulogic_vector (A'left to A'right) := (others=>0);
```

Keyword: OUT

The **out** keyword specifies one of five possible modes for an interface port. (The other four are **in**, **inout**, **buffer**, and **linkage**.)

The **out** mode indicates a port which can be used only for output.

Example

```
component COUNT4EN
    port ( CLK,RESET,EN : in  std_logic;
           COUNT : out std_logic_vector(3 downto 0)
    );
```

```
end component;
```

Keyword: PACKAGE

The **package** keyword specifies a set of declarations which can include the following items: aliases, attributes, components, constants, files, functions, types, and subtypes. A package declaration can also include attribute specifications, disconnection specifications, and use clauses.

By grouping common declarations in a package declaration, the package declaration can be imported and used in other design units.

Example

```
package conversions is
    function to_unsigned (a: std_ulogic_vector) return
        integer;
    function to_vector (size: integer; num: integer) return
        std_ulogic_vector;
end conversions;
```

Keyword: PORT

The **port** keyword is used in a configuration to define the input and output ports of an entity. A port statement includes – in this order – the reserved word "**port**", followed by a list of declarations for the port signals being defined.

Example

```
entity Mux is
   port(A,B: in std_ulogic;
        Sel: in  std_ulogic;
        Y: out std_ulogic);
end Mux;
```

Keyword: POSTPONED

The **postponed** keyword is used to declare a process as a postponed process.

Postponed processes do not execute until the final simulation cycle at the currently modeled time.

Example

```
P1: postponed process (D,Snd,Int)
begin
      -- Statements are postponed to end of simulation cycle
end postponed process;
```

Keyword: PROCEDURE

A **procedure** is a group of sequential statements that are to be executed when the procedure is called.

A procedure does not have a return value, but instead can return any number of values (or no values) via its parameter list. Parameters of a procedure must have a mode associated with them (e.g. **in**, **out**, **inout**). Values are returned by using mode **out** or mode **inout**.

A procedure specification includes – in this order – the reserved word "**procedure**", the procedure name, and a list of the procedure's parameters (which may be constants, signals, or variables, each of whose modes may be **in**, **out**, or **inout**).

Example

```
procedure dff (signal Clk,Rst,D; in std_ulogic;
               signal Q: out std_ulogic) is
  begin
      if Rst <= '1' then
        Q <= '0';
```

```
      elsif rising_edge(Clk) then
        Q <= D;
      end if;
end procedure;
```

Keyword: PROCESS

The **process** keyword defines a sequential process intended to model all or part of a design entity.

A process statement includes – in this order – an optional sensitivity list, a declarations section, a "**begin**" statement, the sequential statements describing the operation of the process, and an "**end**" statement.

The sensitivity list identifies signals to which the process is sensitive. Whenever an event occurs on an item in the sensitivity list, the sequential instructions in the process are executed. If no sensitivity list is provided, the process executes until suspended by a **wait** statement.

In addition to **signal** and **variable** assignments, the sequential statements in the body of the process can include the following: **assertion**, **case**, **exit**, **if**, **loop**, **next**, **null**, **procedure**, **return**, and **wait**.

Example

```
reg: process(Rst,Clk)
        variable Qreg: std_ulogic_vector(0 to 7);
begin
        if Rst = '1' then   -- Async reset
            Qreg := "00000000";
        elsif rising_edge(Clk) then
            if Load = '1' then
                Qreg := Data;
            else
                Qreg := Qreg(1 to 7) & Qreg(0);
            end if;
        end if;
        Q <= Qreg;
end process;
```

Keyword: PURE

The **pure** keyword is used to declare a pure function. Pure functions always return the same value for a given set of input actual parameters, and have no side effects.

Pure is assumed if there is no **pure** or **impure** keyword.

Example

```
pure function HoldCheck (Clk, Data) return Boolean;
```

Keyword: RANGE

The **range** keyword is used to define a range constraint for a scalar type.

A range statement includes – in this order – the reserved word "**range**", the name of the range, and, optionally, two simple expressions for the outer bounds of the range separated by either the reserved word "**to**" (the ascending direction indicator) or the reserved word "**downto**" (the descending direction indicator).

Example

```
variable Q: integer range 0 to 15;
```

Keyword: RECORD

The **record** keyword is used to declare a record type and its corresponding element types.

A record statement includes – in this order – the following:

- the reserved word "**record**",
- an element declaration which includes – in this order – one or more identifiers which share a common subtype, followed by identification of that subtype,
- optionally, additional element declarations of the form specified above, and
- the reserved words "**end record**".

An element declaration that includes more than one identifier (for example, "`COUNT, SUM, TOTAL: INTEGER`") is equivalent to a series of single element declarations.

Example

```
type test_record is record
      CE: std_ulogic;  -- Clock enable
      Set: std_ulogic;
      Din: std_ulogic;
      CRC_Sum: std_ulogic_vector (15 downto 0);
end record;
type test_array is array(positive range <>) of test_record;
```

Keyword: REGISTER

Register is one of two kinds of signals used in a signal declaration (the other is **bus**).

A register signal represents a hardware storage register and defaults to its last driven value when all of the signal's drivers are turned off.

Example

```
signal storage_state : resolve_state state_type register := state_one;
```

Keyword: REJECT

The **reject** keyword is used to specify the minimum pulse width to propagate as a result of an **after** clause.

If no reject time is specified, the specified delay time is assumed for the reject time.

Example

```
Q <= Data reject 2 ns after 7 ns;   -- Delay is 7 ns, reject time is 2 ns
```

Keyword: REM

The **rem** operator is a remainder operator that can be applied to integer types. The result of the expression "`A rem B`" is an integer type and is defined to be the value such that:

- the sign of (`A rem B`) is the same as the sign of `A`, and
- `abs (A rem B) < abs (B)`, and
- `(A rem B) = (A - (A / B) * B)`.

Example

```
begin
  for i in 0 to (bits-1) loop
    if ((tmp rem 2) = 1) then
      out_vec(i) := '1';
    end if;
    tmp := tmp/2;
  end loop;
  return out_vec;
end int_2_v;
```

Keyword: REPORT

The **report** keyword is an option that can be defined as part of an assert statement. It allows a user-defined message to be displayed if the given condition of the statement is false.

Example1

```
assert (S = S_expected)
report "S does not match the expected value!"
severity Error;
```

The **report** keyword can also be used within a loop for debugging purposes. A message will be reported to the screen at each iteration of the loop.

Example2

```
architecture example of loop_stmt is
begin
  process (a)
    variable b: integer;
  begin
    b := 1;
    while b < 7 loop
       report "Loop number = " & integer'image(b);
       b := b + 1;
    end loop;
  end process;
end example;
```

Keyword: RETURN

Return is a sequential statement used at the end of a subprogram (a function or procedure) to terminate the subprogram and return control to the calling object.

When used in a procedure, the reserved word "**return**" appears alone.

When used in a function, the reserved word "**return**" must be followed by an expression which defines the result to be returned by the function. The expression's type must be the same type as specified by the return statement in the function's specification.

A return statement must be the last statement executed in a function.

Example

```
function rising_edge (signal s: std_logic) return boolean is
begin
     return (s'event and (To_X01(s) = '1') and
         (To_X01(s'last_value) = '0'));
end rising_edge;
```

Keyword: ROL

Rol is the rotate left operator. Each bit in the left operand is shifted left by the number of bits specified in the right operand. Bits in the left-most positions of the operand are shifted to the right-most bits of the operand.

Example

```
Sreg <= Sreg rol 2;
```

Keyword: ROR

Ror is the rotate right operator. Each bit in the left operand is shifted right by the number of bits specified in the right operand. Bits in the right-most positions of the operand are rotated to the left-most bits of the operand.

Example

```
Sreg <= Sreg ror 2;
```

Keyword: SELECT

Select is a concurrent signal assignment statement that selects and assigns a value to a target signal from among a list of alternatives, based on the value of a given expression.

A select statement includes – in this order – the following:

- the reserved word "**with**", followed by the expression to be evaluated, followed by the reserved word "**select**",
- the target signal, followed by the reserved symbol "**<=**", followed by:

  (a) the first value which could be assigned to the target signal, followed by the reserved word "**when**", followed by a choice which, if the expression evaluates to be that choice, will cause the first value to be assigned to the target signal, and

  (b) second and subsequent values which could be assigned to the target signal, each followed by the reserved word "**when**", and each followed by a choice which, if the expression evaluates to be that choice, will cause the value to be assigned to the target signal.

Since the select statement chooses one and only one alternative for execution at a given time, all possible values for the expression must be covered in "**when**" statements. An "**others**" clause may be used to cover values not explicitly named.

Example

```
architecture concurrent of mux is
begin
    with Sel select
        Y <= A when "00",
        B when "01",
        C when "10",
        'X' when others;
end concurrent;
```

Keyword: SEVERITY

The **severity** keyword is an option that can be defined as part of an assert statement. It allows the user to choose a severity level if the given condition of the statement is false.

The four possible severity levels are: Note, Warning, Error, and Failure. The value of severity is typically used to control the actions of a simulation in the event the given condition is false. For example, a severity level of Failure may be used to stop the simulation.

Example

```
if (S /= S_expected) then
   err_cnt := err_cnt + 1;
   assert false
      report "Vector failure!" & lf &
      "Expected S to be  " & stdvec_to_str(S_expected) & lf &
      "but its value was " & stdvec_to_str(S) & lf
      severity note;
end if;
```

Keyword: SHARED

The shared keyword is used as part of a variable declaration to allow the variable to be accessed by multiple processes.

Example

```
architecture example of test1 is
shared variable base_time : natural := 0;
...
begin
...
```

```
end architecture example;
```

Notes

Shared variables can only be declared in specific areas of VHDL code, namely:

- as part of an entity declaration
- in the body of an architecture
- in a block statement
- in a generate statement
- in a package declaration.

Keyword: SIGNAL

Signal declares a signal of a specified type.

A signal declaration includes – in this order – the reserved word "**signal**", the name of the signal, the subtype of the signal, an optional indication of the signal's kind (which must be either "**register**" or "**bus**"), and optionally, an expression specifying the initial value of the signal.

Example

```
architecture behavior of fsm is
    signal current_state: state;
    signal DataBuf: std_logic_vector(15 downto 0);
begin
    ...
end behavior;
```

Notes

Signals declared within an entity are visible in the corresponding architecture(s).

A signal cannot be declared within a process, procedure or function.

Keyword: SLA

The **sla** keyword is the shift left arithmetic operator.

Example

```
Addr <= Addr sla 8;
```

Keyword: SLL

The **sll** keyword is the shift left logical operator.

Example

```
Addr <= Addr sll 8;
```

Keyword: SRA

The **sra** keyword is the shift right arithmetic operator.

Example

```
Addr <= Addr sra 8;
```

Keyword: SRL

The **srl** keyword is the shift right logical operator.

Example

```
Addr <= Addr srl 8;
```

Keyword: SUBTYPE

The **subtype** keyword declares a subtype (a type with a constraint that is based on an existing parent type).

A subtype declaration includes – in this order – the reserved word "**subtype**", the subtype's identifier, the reserved word "**is**", an optional resolution function, the base type of the subtype, and an optional constraint. If no constraint is included, the subtype is the same as the specified base type.

Example

```
subtype short is integer range 0 to 255;
...
subtype X01Z is std_ulogic range 'X' to 'Z';
```

Keyword: THEN

The **then** keyword is part of the syntax of an **if** statement.

Example

```
if A > B then
    Compare <= GT;
elsif A < B then
    Compare <= LT;
else
    Compare <= EQ;
end if;
```

Keyword: TO

The **to** keyword is used to indicate an ascending range in a **range** statement or other statement which includes a range (for example, a variable statement). (The reserved word "**downto**" is used to indicate a descending range.)

Example

```
signal A0,A1: std_ulogic_vector(0 to 15);
```

Keyword: TRANSPORT

The **transport** keyword is used to specify non-inertial delay in a signal assignment statement.

Example

```
Waveform <= transport '1' after 10 ns;
```

Keyword: TYPE

The **type** keyword declares a type.

There are two kinds of type declarations: a full type declaration and an incomplete type declaration.

A full type declaration includes – in this order – the reserved word "**type**", the type identifier, the reserved word "**is**", and the type definition. A type definition can be an access type, a composite type, a file type, or a scalar type.

An incomplete type declaration includes only the reserved word "**type**" followed by the type's identifier. If an incomplete type declaration exists, a full type declaration with the same identifier must also exist. The full type declaration must occur after the incomplete type declaration and within the same declarations section as the incomplete type declaration.

Example

```
type StateMachine is (RESET, IDLE, READ, WRITE,
        ERROR);
...
type RAD16 is range 0 to 15;
...
type test_record is record
    CE: std_ulogic;  -- Clock enable
```

```
    Set: std_ulogic;
    Din: std_ulogic;
    CRC_Sum: std_ulogic_vector (15 downto 0);
end record;
```

Notes

The two type declarations define two different types, even if the definitions are the same and they differ only by their respective identifiers.

Keyword: UNAFFECTED

The **unaffected** keyword is used to indicate in a conditional or selected signal assignment when the signal is not to be given a new value.

Example

```
Mux <= A when Sel = "00" else
       B when Sel = "01" else
       C when Sel = "10" else
       unaffected;
```

Keyword: UNITS

The **units** keyword is used in a type declaration to declare physical types.

A units statement includes – in this order – the following:

- the reserved word "**units**",
- the base unit,
- optionally, one or more secondary units, and
- the reserved words "**end units**".

Example

```
type time is range -2_147_483_647 to 2_147_483_647
    units
        fs;
        ps  = 1000 fs;
        ns  = 1000 ps;
        us  = 1000 ns;
        ms  = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr  = 60 min;
    end units;
```

Keyword: UNTIL

The **until** keyword is used as part of a wait statement to temporarily suspend a process until a specified condition is met.

Example

```
process
begin
    wait until Clk = '1' and Clk'event;
    M_out <= data_in;
    wait until Clk = '1' and Clk'event;
    M_out <= not data_in;
end process;
```

Keyword: USE

The **use** statement identifies items in other design units so those items can be referenced.

A use clause includes – in this order – the reserved word "**use**", followed by a list of design units (or design unit items) to be referenced.

A use clause makes the referenced design units visible to the working environment. If a design unit (or design unit item) belongs to a library different from the current library, a **library** statement must be included before the use statement. The library statement must specify the library holding the referenced design unit.

Example

```
use mylib.mypackage.dff;
...
use mylib.mypackage.all;
...
use mylib.all;
...
use work.all;
```

All design units automatically include the following two implicit clauses:

```
library STD, WORK;
use STD.STANDARD.all;
```

Keyword: VARIABLE

The **variable** keyword declares a variable of a specified type.

A variable declaration includes – in this order – the reserved word "**variable**", the variable's name, the variable's subtype, and, optionally, an expression specifying the initial value of the variable.

Example

```
process(Rst,Clk)
        variable Q: integer range 0 to 15;
begin
    if Rst = '1' then   -- Asynchronous reset
        Q := 0;
    elsif rising_edge(Clk) then
        if Load = '1' then
            Q := to_unsigned(Data);  -- Convert vector to
-- integer
        elsif Q = 15 then
            Q := 0;
        else
            Q := Q + 1;
        end if;
    end if;
    Count <= to_vector(4,Q); -- Convert integer to
-- vector
end process;
```

Notes

A variable can only be declared within a process, procedure or function. Also, a variable cannot be of a file type.

Variables declared within a process have their values preserved during subsequent executions of the process.

Variables declared within a function or procedure have their values initialized each time the function or procedure is called.

Keyword: WAIT

The **wait** statement is used to temporarily suspend a process until:

- a specified time has passed ("**wait for**", followed by a time expression), or
- a specified condition is met ("**wait until**", followed by a Boolean expression), or
- an event occurs which affects one or more signals ("**wait on**", followed by a sensitivity list which specifies signals on each of which an event must occur before processing continues).

Example

```
CLOCK: process
    variable c: std_ulogic := '0';
    constant PERIOD: time := 50 ns;
begin
    wait for PERIOD / 2;
    c := not c;
    clk <= c;
end process;
```

Notes

When a **wait** statement is used within a process, the process must not include a sensitivity list.

Keyword: WHEN

The **when** keyword is used to specify a condition during which an **exit** or **next** statement will be executed.

Example1

```
L1 : for i in 0 to 9 loop
  L2 : for j in opcodes loop
         for k in 4 downto 2 loop-- loop label is optional
           if k = i next L2;       -- go to next L2 loop
         end loop;
       exit L1 when j = crash; -- exit loop L1
  end loop;
end loop;
```

It is also used to specify a choice (or choices) within a **case** statement.

Example2

```
case current_state is
     when IDLE =>
       if start_key = '1' then
       current_state <= READ0;
       end if;
     when READ0 =>
       current_state <= READ1;
     when READ1 =>
       current_state <= READX;
     when READX =>
       current_state <= WRITE0;
     when WRITE0 =>
```

```
        current_state <= WRITEX;
      when WRITEX =>
        current_state <= IDLE;
end case;
```

Keyword: WHILE

The **while** keyword is used to specify a condition during which a loop will be executed.

Example

```
process
begin
    while error_flag /= '1'  and done /= '1' loop
        Clock <= not Clock;
        wait for CLK_PERIOD/2;
    end loop;
end process;
```

Keyword: WITH

The **with** keyword is used in the syntax of a selected signal assignment.

Example

```
architecture concurrent of mux is
begin
    with Sel select
        Y <= A when "00",
        B when "01",
        C when "10",
        'X' when others;
end concurrent;
```

Keyword: XNOR

Xnor is the logical “both or neither” (equality) operator which can be used in an expression.

The expression “`A xnor B`” returns True only when

- `A` is true and `B` is true, or
- `A` is false and `B` is false.

Example

```
architecture example of test is
begin
    Y <= a xnor b;
end example;
```

Keyword: XOR

Xor is the logical “one or the other but not both” (inequality) operator which can be used in an expression. The expression “`A xor B`” returns True only when

- `A` is true and `B` is false, or
- `A` is false and `B` is true.

Example

```
entity fulladder is
```

```
    port (X: in bit;
          Y: in bit;
          Cin: in bit;
          Cout: out bit;
          Sum: out bit);
end fulladder;
architecture concurrent of fulladder is
begin
    Sum <= X xor Y xor Cin;
    Cout <= (X and Y) or (X and Cin) or (Y and Cin);
end concurrent;
```

Revision History

Date	Version No.	Revision
29-Nov-2004	1.0	New product release
20-May-2005	1.1	Updated for SP4
20-Sep-2005	1.2	Correction to Use statement in Partitioning Features\Packages section to use the same name as that defined for the package (my_types).
04-Mar-2008	2.0	Updated for Altium Designer Summer 08
21-Aug-2011	-	Updated template.