VHDL Overview

condensed from:
Accolade Reference Guide

http://www.acc-eda.com/vhdlref

Notes: / 3055-05 / pdf / lec_vhdl_notes.pdf

Entity and Architecture Declarations

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

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


Entity and Architecture Declarations - 2

Entity declaration

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

Architecture declaration

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 to define the actual functional description.

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

Packages

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.

Package Example

```
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;
```


Configuration

The final type of design unit available in VHDL is called a configuration declaration. You can think of a configuration declaration as being a parts list for your design. A configuration declaration 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.

```
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;
```

Conditional Signal Assignment

A conditional signal assignment is a special form of signal assignment, similar to the if-then-else statements found in software 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;
```

7

Selected Assignment

The conditional expression:

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

versus the **selected assignment** :

```
with std_logic_vector'(A,B,C) select
    Q2 <= "01" when "100",
          "01" when "101",        # note commas at end of line, not else
          "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.

8

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.

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;
```

9

Component Instantiations

Component instantiations are statements that reference lower-level components in your design, in essence creating unique copies (or instances).

```
library ieee;
use ieee.std_logic_1164.all;                # note: "half-adder" and "full_adder" defined in library

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 adder4s 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));   # this is "positional mapping"
      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 adder4s ;
```


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.)

We could modify the previous 4-bit adder example to use **named association** as follows:

```
architecture adder4s 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 adder4s ;
```


Generic Mapping

If the lower-level entity being referenced includes generics (described in more detail in Chapter 8, Partitioning Your Design), you can specify a **generic map** in addition to the port map **to pass actual generic parameters** to the lower-level entity:

```
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 adder4s ;
```

Process Statements

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.

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

It is like a program that executes 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.

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 may be either combinational, sequential (registered), or a combination of the two. They 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 `**event** signal attribute determines 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          # note missing "e" in "elsif"
      if Load = '1' then
         Q <= Data_in;
      else
         Q <= Q(1 to 7) & Q(0);
      end if;
   end if;
end process;
```


Sequential Statements in Subprograms

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.

For examples, see http://www.acc-eda.com/vhdlref/

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.

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.

Other Features

These slides have only highlighted some of the VHDL features. More features are explained in the Accolade tutorial:

http://www.acc-eda.com/vhdlref/.