**VHDL ENTITIES, ARCHITECTURES, AND PROCESS**

**VHDL models** consist of two major parts:
1) Entity declaration – defines the I/O of the model
2) Architectural body – describes the operation of the model

**Format of Entity declaration:**
```vhdl
entity entity_name is
    port(
        signal_name(s): mode signal_type;
    :
        signal_name(s): mode signal_type);
end entity entity_name;
```
signals of the same mode and signal_type can be grouped on 1 line

**MODE** describes the direction data is transferred through port
- in – data flows into the port
- out – data flows out of port **only**
- buffer – data flows out of port **as well as** internal feedback
  - note: can be used for any output regardless of feedback
- inout – bi-directional data flow into and out of port

**SIGNAL_TYPE** defines the data type for the signal(s)
- bit – single signals that can have logic values 0 and 1
- bit_vector – bus signals that can have logic values 0 and 1
- std_logic – same as bit but intended for standard simulation and synthesis (IEEE standard 1164)
- std_logic_vector – same as bit_vector but IEEE standard for simulation and synthesis

**Notes:**
- all vectors must have a range specified
- example for a 4 bit bus: `bit_vector (3 downto 0)`
  or `std_logic_vector (3 downto 0)`

**Values for std-logic:**
- **U** un-initialized (undefined logic value)
- **X** forced unknown logic value
- **0**
- **1**
- **Z** high impedance (tri-state)
- **W** weak unknown
- **L** weak 0
- **H** weak 1
- **-** don’t care value (for synthesis minimization)

**Comments in VHDL** are of the following format, the comment begins with the double dashes (no space between them) and continues to the end of the current line
-- this is a syntactically correct VHDL comment
Entity example:
a 4-bit full adder with Carry-in & Carry-out

entity ADDER is
  port (Cin: in bit;
        A, B: in bit_vector (3 downto 0);
        Sum: out bit_vector (3 downto 0);
        Cout: out bit);
end entity ADDER;

same entity declaration using std_logic & std_logic_vector:

library IEEE;
use IEEE.std_logic_1164.all;

entity ADDER is
  port (Cin: in std_logic;
        A, B: in std_logic_vector (3 downto 0);
        Sum: out std_logic_vector (3 downto 0);
        Cout: out std_logic);
end entity ADDER;

Format for Architecture body (in its simplest form):
architecture architecture_name of entity_name is
begin
  :
end architecture architecture_name;

Notes: entity and architecture in the end statement is optional.
The actual behavior of the VHDL model is described between the begin and end statements.
In order to give an example architectural body we must consider some constructs that allow us to describe model’s behavior. We first consider the process statement (very commonly used)

Format for process statement:
process_label: process (sensitivity_list)
begin
  :
end process process_label;

  note that the process_label is optional while the sensitivity list is required to implement the correct simulation behavior

Sensitivity list – list of signals that cause the process to execute
Within the process each statement is executed sequentially and only sequential statements can be used in a process (more on what are sequential statements later but for now just think of sequential execution of a typical program)
Now we need to look at a sequential statement construct in order for us to complete a process statement as well as an architecture body. One of the most commonly used is the **if-then-else** statement which we consider here.

**Format for if-then-else statement:**

if *condition* then
   *sequence of statements*

elsif *condition* then
   *sequence of statements*

else
   *sequence of statements*
end if;

-- active high level sensitive D-latch with active low reset
library IEEE;
use IEEE.std_logic_1164.all;
entity LAT is
   port(D, EN, RST: in std_logic;
        Q: out std_logic);
end entity LAT;
architecture LALA of LAT is
begin
   process (D,EN,RST)
   begin
      if (RST = '0') then
         Q <= '0'; -- here we reset the latch when RST=0
      elsif (EN = '1') then
         Q <= D; -- here we pass D to Q when EN=1
      end if;  -- note that no else implies storage state
   end process;
end architecture LALA;

A lot of code for a silly little latch? What about a whole register (say 4 bits)?

-- active high level sensitive D-latch register with active low reset
library IEEE;
use IEEE.std_logic_1164.all;
entity REG is
   port(EN, RST: in std_logic;
        D: in std_logic_vector (3 downto 0);
        Q: out std_logic_vector (3 downto 0));
end entity REG;
architecture LALA of REG is
begin
   process (D,EN,RST)
   begin
   end process;
end architecture LALA;
VHDL ENTITIES, ARCHITECTURES, AND PROCESS

if (RST = '0') then
    Q <= "0000"; -- reset the register when RST=0
elsif (EN = '1') then
    Q <= D;  -- here we pass D to Q when EN=1
end if;   -- no else implies storage state
end process;
end architecture LALA;

Notes: With very little change in VHDL code we have a whole register
In the previous example we use ‘0’ to pass a single bit value to Q but here we use “0000”
to pass a vector value (string) to Q. The assignment operator for signals is <=.
The order of the if-elsif-else sequence establishes the precedence of the operations by the
input signals (what take priority)
Passing data between bit_vectors is the same as between bits (but note that the size and
ordering was the same for both D and Q)
What if we have a bunch of registers that work the same but have different sizes? Can
one size fit all? Behold the power of VHDL! (via the generic)
The generic statement is like the port statement and is in the entity but it allows us to
specify (and change) the size of busses

Format for generic statement:
generic (identifier: type [:= default value];
        identifier: type [:= default value]);

Example: and N-bit register
-- active level sensitive D-latch based register with active low reset
library IEEE;
use IEEE.std_logic_1164.all;
entity REG is
    generic (N: integer := 4)
    port(EN, RST: in std_logic;
         D: in std_logic_vector (N-1 downto 0);
         Q: out std_logic_vector (N-1 downto 0));
end entity REG;
architecture BIGLALA of REG is
begin
    process (D,EN,RST)
    begin
        if (RST = '0') then
            Q <= (others => '0'); -- reset the register when RST=0
        elsif (EN = '1') then
            Q <= D;  -- here we pass D to Q when EN=1
        end if;   -- no else implies storage state
    end process;
end architecture BIGLALA;
Notes on the generic statement:
Here we are using a new type (the integer type) for our identifier N and the immediate assignment operator for an integer is :=
We have included the optional assignment of a default value (:= 4). For synthesis of an individual VHDL model specifying a default value is needed
But multiple calls to the same parameterized model using generics can specify any size for each instantiation of the model at the next higher level of hierarchy and the default value will be over-ridden

The moral to the BIGLALA story:
Whenever you can parameterize a VHDL model, you should do so!!!
It allows easier design verification of a smaller function (use small generic values but verify different generic values)
It promotes reuse of designs that have been verified and proven to work!!! This reduces design errors!!!
It also facilitates optimized synthesis for area and/or performance when you have taken the time to do such optimization!!!
Bottom line: it makes you a better designer!!!