LAB 1: INTRO TO VHDL

Objective: In this lab you will learn the basic skeleton of VHDL. You will also learn how to design one-bit and two-bit comparators and learn to simulate your design using test files. You will be also introduced to ModelSim.

Part I

Comparator

We will use the design tool Modelsim to design, simulate and verify our VHDL designs. Modelsim is widely used in industry.

VHDL is intended for describing and modeling a digital system at various levels and is an extremely complex language.

In this lab, we use a simple comparator to illustrate the skeleton of a VHDL program. The description uses only logical operators and represents a gate-level combinational circuit, which is composed of simple logic gates.

Input		Output
I0	I1	Eq
0	0	1
0	1	0
1	0	0
1	1	1

Truth table of a one-bit comparator

Eq = I0.I1 + I0'.I1'

Here is the VHDL code describing the truth table shown above.

One_bit_comparator.vhd

```
1     Library ieee;
2     Use ieee.std_logic_1164.all;

3     Entity equal is
4     Port (
5     I0, I1: in std_logic;
6     Eq    : out std_logic);
7     End equal;

8     Architecture arch of equal is
9     Signal p0, p1 : std_logic;
10    begin
```

```
11    Eq <= p0 or p1;
12    P0 <= (not I0) and (not I1);
13    P1<= i0 and i1;
14    End arch;
```

VHDL is case insensitive, which means that upper and lowercase letters can be used Interchangeably, and free formatting, which means that spaces and blank lines can be inserted freely.

Lines 1 and 2 tell the compiler (in our case Modelsim) which libraries to use. The libraries contain precompiled VHDL code. For example `ieee.std_logic_1164.all` contains the code for the 'or' function, and without it the VHDL compiler would generate and error on line 11.
Lines 3 to 7 are the entity declaration statements. The entity declaration outlines the input and output signals of the circuit. The mode term can be **in** or **out,** which indicates that the corresponding signals flow "into" or "out of" of the circuit. It can also be **inout,** for bidirectional signals.

Lines 8 to 14 are the architecture statements. The architecture body describes functions of the circuit. The architecture may include signals which we have used in line 9. We need the signals to store the value of first product and second product. You will understand signals more as you do more coding. The main description, encompassed between **begin** and **end,** contains three *concurrent statements.* Unlike a program in C language, in which the statements are executed sequentially, concurrent statements are like circuit parts that operate in parallel. The signal on the left-hand side of a statement can be considered as the output of that part, and the expression specifies the circuit function and corresponding input signals.

Test file

After the code is developed, it can be *simulated* in a host computer to verify the correctness of the circuit operation and can be *synthesized* to a physical device (DE2 board for example).We create a special program, known as a *test bench,* to imitate a physical lab bench. This can be done without a test bench as well but it is more professional and exact to use a test bench because you will have more control on the stimulation. Here's the test file for the one-bit comparator.

Test_One_bit_comparator.vhd

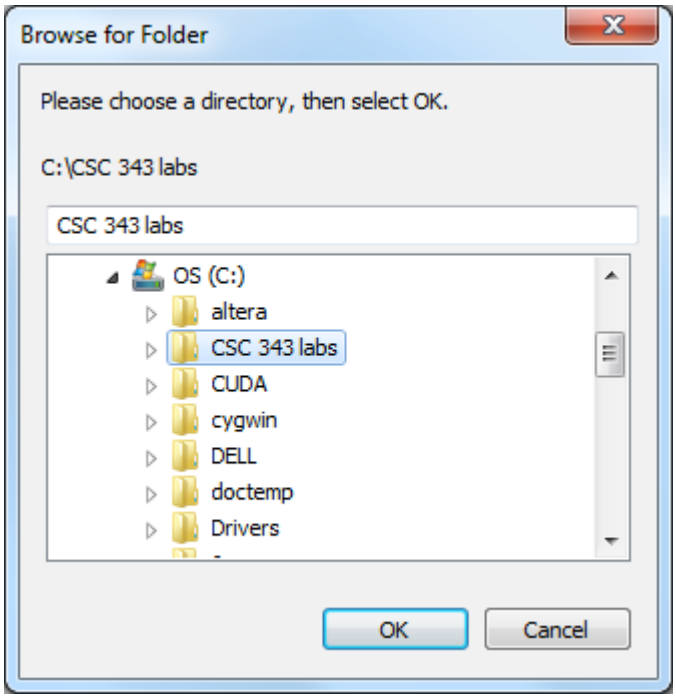
```
Library ieee;
Use ieee.std_logic_1164.all;

Entity test_equal is
End test_equal;

Architecture arch_test of test_equal is
```

```
component equal
Port (
I0, I1: in std_logic;
Eq    : out std_logic);
End component;

Signal p1, p0, pout : std_logic;
Signal error        : std_logic := '0';
begin
uut: equal port map(I0 => p0, I1 => p1, Eq => pout);
process
begin
p0 <=  '1';
p1 <= '0';
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  '1';
p1 <= '1';
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  '0';
p1 <= '1';
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  '0';
p1 <= '0';
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;

if (error = '0') then
         report "No errors detected. Simulation successful" severity
failure;
      else
         report "Error detected" severity failure;
end if;

end process;
End arch_test;
```

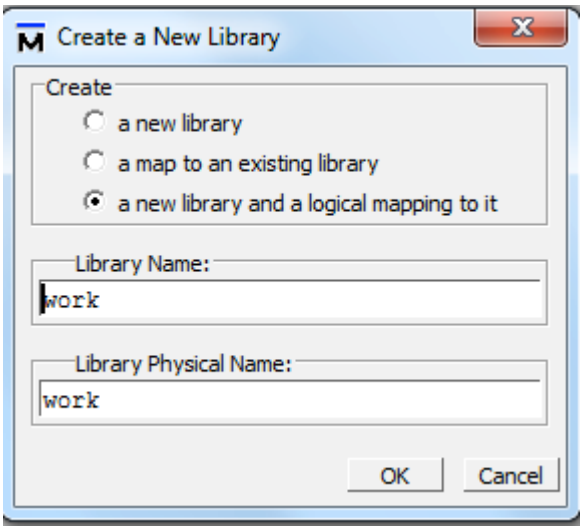
Read and try to understand the test file as much as you can. You will understand more as we go.

Implementing one-bit comparator in ModelSim

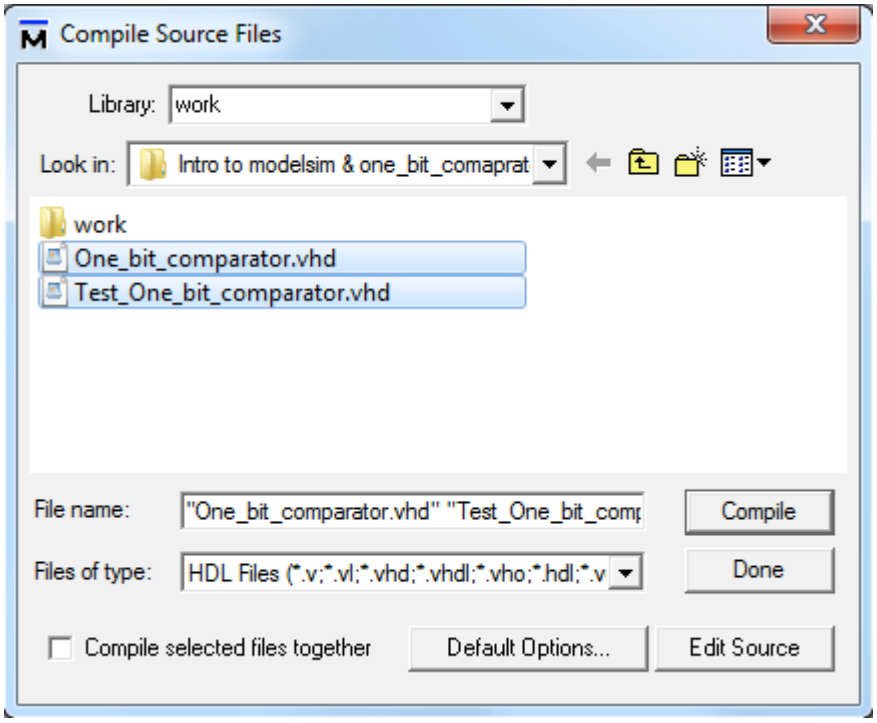
1) Type and save One_bit_comparator.vhd and Test_One_bit_comparator.vhd in any text editor you want.
2) Create a new directory and copy the above mentioned files into that directory.
3) Open Modelsim by typing Vsim in terminal.
4) Click on File->Change Directory You will see a screen similar to the one below. Select the directory you just created and press ok.



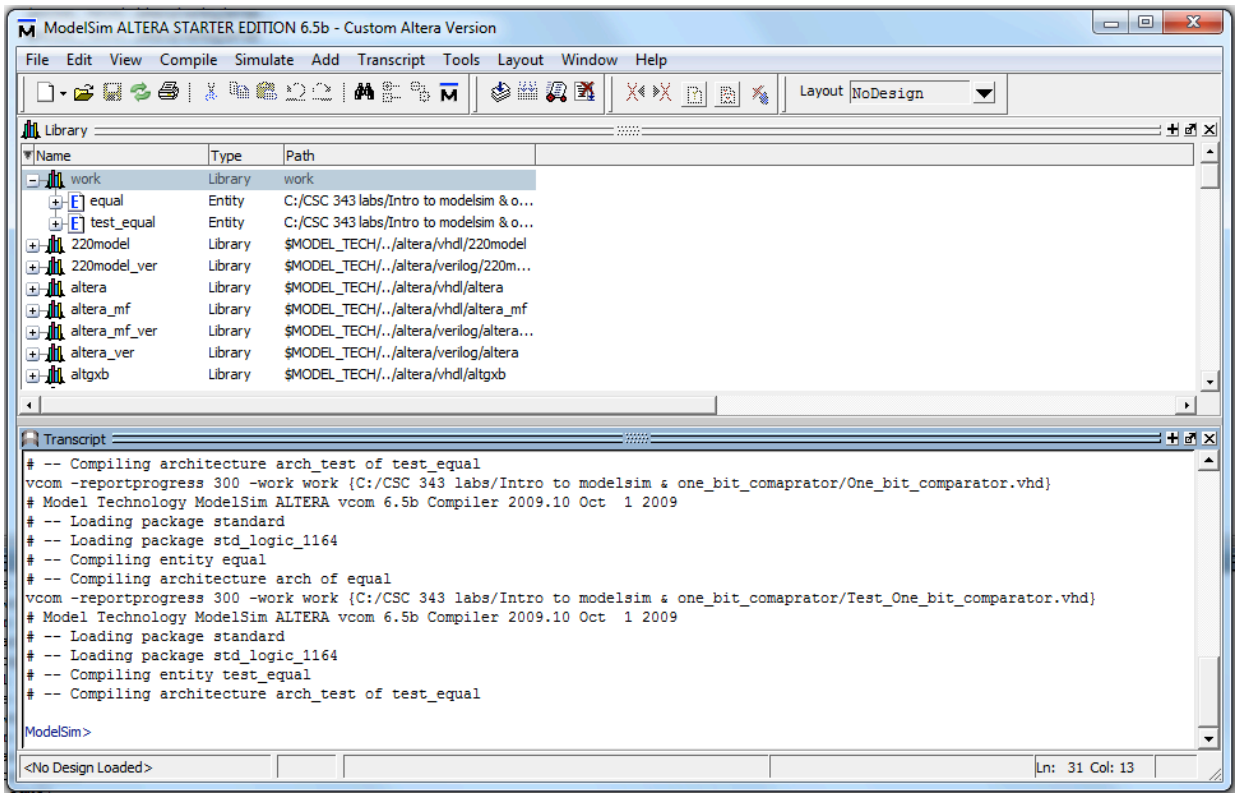
5) Click on File->New Library. A screen will appear similar to the figure below. Name the library work.



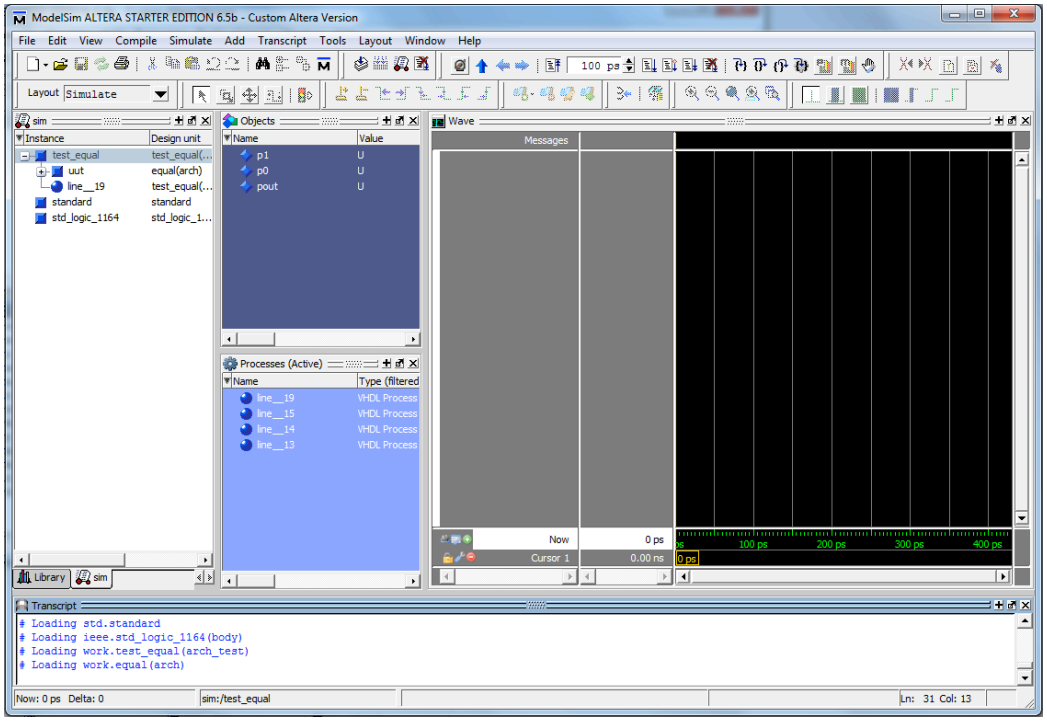
6) Click Compile->compile. Select the two VHD files that you created and compile them and then press done. See the figure below.



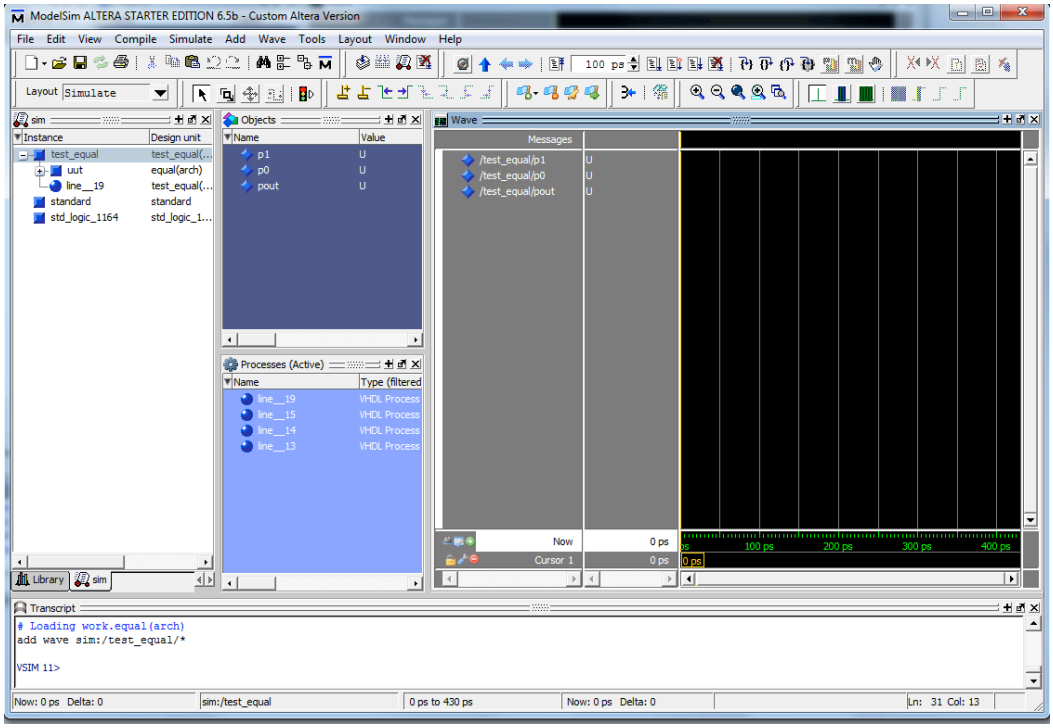
7) On the left side of the screen you will see a set of libraries in the workspace window. Click on the plus sign to the left of the work library that you just created. You should see the files that u have just compiled. See figure below.



8) To simulate double-click on Test_One_bit_comparator.vhd under the work library. This will give you the screen shown below.



9) Right-click "Test_One_bit_comparator" in the instance window. Click Add-> Add to wave->all items in the region. Your screen should look similar to the picture below.

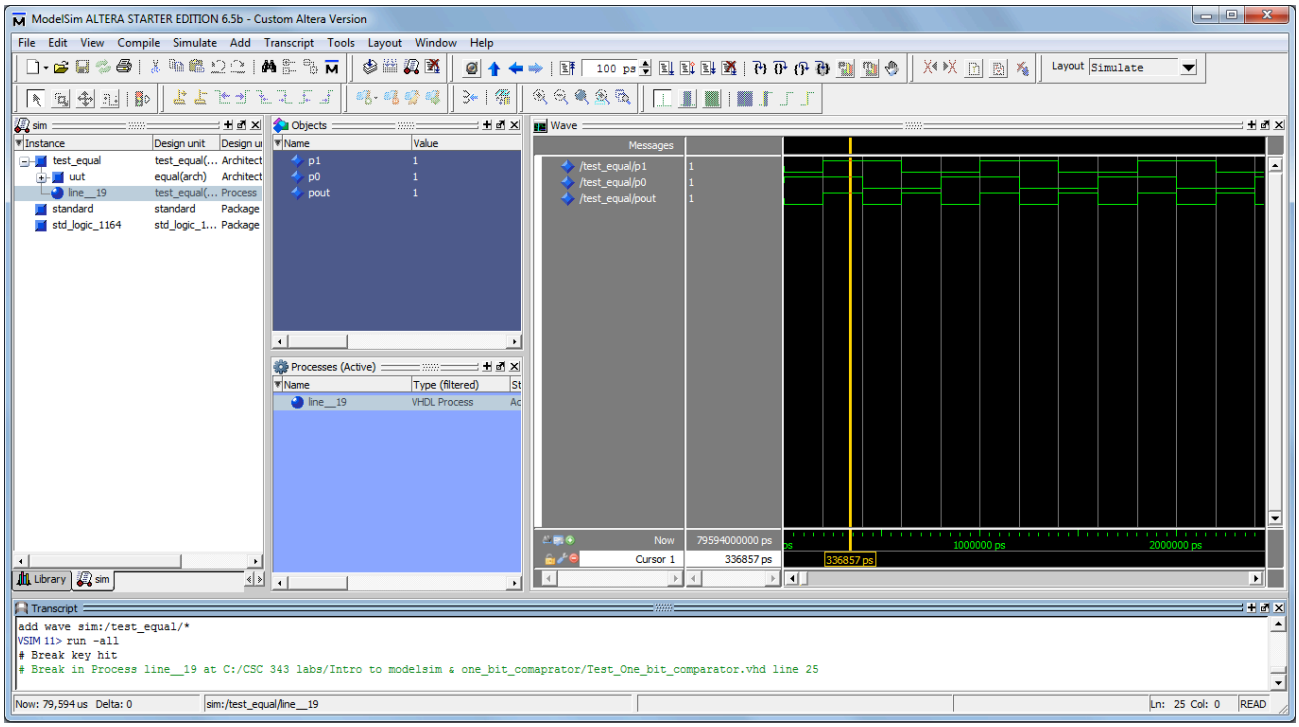


10) To run the simulation, click on the "Run All" icon. Normally a simulation will run until the "Break" icon is pressed, however in our test file the simulation is forced to terminate after a short period of time.

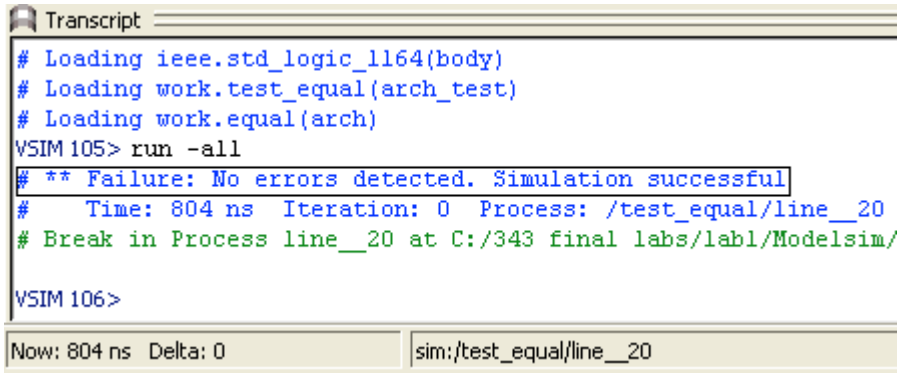

Run All


Break

11) After simulating your design you will see a waveform in the wave window (you may have to close the ModelSim text editor to see it). You can zoom in or zoom out to observe your results. The values you have gotten may not be exactly like the picture below. Analyze that your results are correct based on the truth table. Use the scrollbar to scroll through the values.



The test code also produces a report in the transcript window on the bottom of the screen. If no errors are found it report that simulation was successful, else is report that an error was detected. You can use the error signal in the waveform to see at what exact point the error was found. Here is how a successful simulation would look like.



Part II

Two-bit comparator

Now that we have written our first piece of VHDL code we want to make it a bit more complicated. Now we want to make a two-bit comparator. The program will take two two-bit inputs and compare them and check if their equal or not. This can be done in two ways. You will implement the two-bit comparator in both ways and at the end you will understand their differences.

If you draw out the truth table for the two-bit comparator you will be able to derive the logical statement given below:

aeqb = (a1'.b1').(a0'.b0') + (a1'.b1').(a0.b0) + (a1.b1).(a0'b0') + (a1.b1).(a0.b0)

Fill out the missing parts in the code below using the logical expression above.

two_bit_comparator.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity two_bit_equal is
Port (
a, b: in std_logic_vector(1 downto 0);
aeqb    : out std_logic);
End two_bit_equal;

Architecture arch of two_bit_equal is
Signal p0, p1,p2,p3 : std_logic;
begin
aeqb <= --Enter you code here;
P0 <= --Enter you code here;
P1<= --Enter you code here.;
P2<=--Enter you code here;
P3 <=--Enter you code here;
End arch;
```

The only difference in this code is that inputs a and b are declared as "STD_LOGIC_VECTOR(1 downto 0)" which means its a vector of size two.
Now to test our design we need a test file. Again the test file will be given to you but try to understand how it works.

Test_two_bit_equal.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity Test_two_bit_equal is
End Test_two_bit_equal;

Architecture arch_test of test_two_bit_equal is

component  two_bit_equal
Port (
a, b: in std_logic_vector(1 downto 0);
aeqb    : out std_logic);
End component;

Signal p1, p0 : std_logic_vector(1 downto 0);
Signal pout   : std_logic;
Signal error        : std_logic := '0';
begin
uut: two_bit_equal port map(a => p0, b => p1, aeqb => pout);
process
begin
p0 <=  "00";
p1 <= "00";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "01";
p1 <= "00";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "01";
p1 <= "11";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11";
p1 <= "00";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11";
p1 <= "11";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
```

```
end if;
wait for 200 ns;
p0 <=  "10";
p1 <= "11";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "10";
p1 <= "10";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11";
p1 <= "01";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;

if (error = '0') then
         report "No errors detected. Simulation successful" severity
failure;
      else
          report "Error detected" severity failure;
end if;

end process;
End arch_test;
```

Part III

As you can see writing the two-bit comparator using logical expressions could get a bit tiresome and confusing especially as we increase the number of bits. There is an easier way to do this using "PORT MAPS". What port map does is that it uses other components that we or others have created and connects them together to make something bigger. For example to make a two-bit comparator we will connect 2 one-bit comparators making life much easier and simpler.
Below is the code for the two-bit comparator with the use of port maps.

two_bit_equal_port.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity two_bit_equal_port is
Port (
a, b: in std_logic_vector(1 downto 0);
aeqb    : out std_logic);
End two_bit_equal_port;

Architecture arch of two_bit_equal_port is

-- component declaration...we are telling the compiler which
components we want to use from the library.

component equal
Port (
I0, I1: in std_logic;
Eq    : out std_logic);
End component;

signal e0,e1: std_logic;

begin

--instantiates two one-bit comparators

H1: equal
port map(i0=>a(0), i1=>b(0), eq=>e0);
H2: equal
port map(i0=>a(1), i1=>b(1), eq=>e1);

-- a and b are equal if individual bits are equal.

aeqb <= e0 and e1;

end arch;
```

Observer the syntax of using port maps because it is very important as we continue with more sophisticated projects. Port maps act as wires connecting different components or black boxes together. H1 and H2 used in the code are just names for the components; you can name them anything you want.

To test this design use the test file given above but change the name of the component to two_bit_equal_port since you are simulating this file instead. Simulate your code and analyze your results.

Part IV

Using the two-bit comparator example, write the VHDL code for an eight-bit comparator. Your code should look very similar to the code from part III of this lab with the only difference that instead of using two instances of the "equal" component, you will use eight instances of the one-bit comparator. Use the following test code to verify your results.

Test_eight_bit_equal_port_map.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity Test_eight_bit_equal_port_map is
End Test_eight_bit_equal_port_map;

Architecture arch_test of Test_eight_bit_equal_port_map is

component eight_bit_equal_port_map
      Port (
      a, b: in std_logic_vector(7 downto 0);
      aeqb    : out std_logic);
End component;

Signal p1, p0 : std_logic_vector(7 downto 0);
Signal pout   : std_logic;
Signal error        : std_logic := '0';
begin
uut: eight_bit_equal_port_map port map(a => p0, b => p1, aeqb => pout);
process
begin
p0 <=  "00000000";
p1 <= "00000000";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "01010101";
p1 <= "00010101";
```

```
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "01100101";
p1 <= "11111001";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11110011";
p1 <= "00010100";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11001100";
p1 <= "11001100";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "10010001";
p1 <= "11100111";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "10111001";
p1 <= "10111001";
wait for 1 ns;
if (pout = '0') then
  error <= '1';
end if;
wait for 200 ns;
p0 <=  "11010011";
p1 <= "01101001";
wait for 1 ns;
if (pout = '1') then
  error <= '1';
end if;
wait for 200 ns;

if (error = '0') then
         report "No errors detected. Simulation successful" severity
failure;
      else
          report "Error detected" severity failure;
end if;

end process;
End arch_test;
```