Bitwise Operations (Equal and Less)

Name: Jonathan J. Rodriguez

CSC343 – Computer Organization Lab, Spring 2020

Instructor: Izidor Gertner

Table of Contents

Objective

The objective of this laboratory exercise is to do bitwise operations on two registers created previously to create a bitwise comparator. Once the comparator receives the information from the registers, it will then check if they are equal. If so, it will send the third register a logical '1'. Else, it will send the third register a logical '0'. This will be tested using 1, 4, 8, and 32 bit registers.

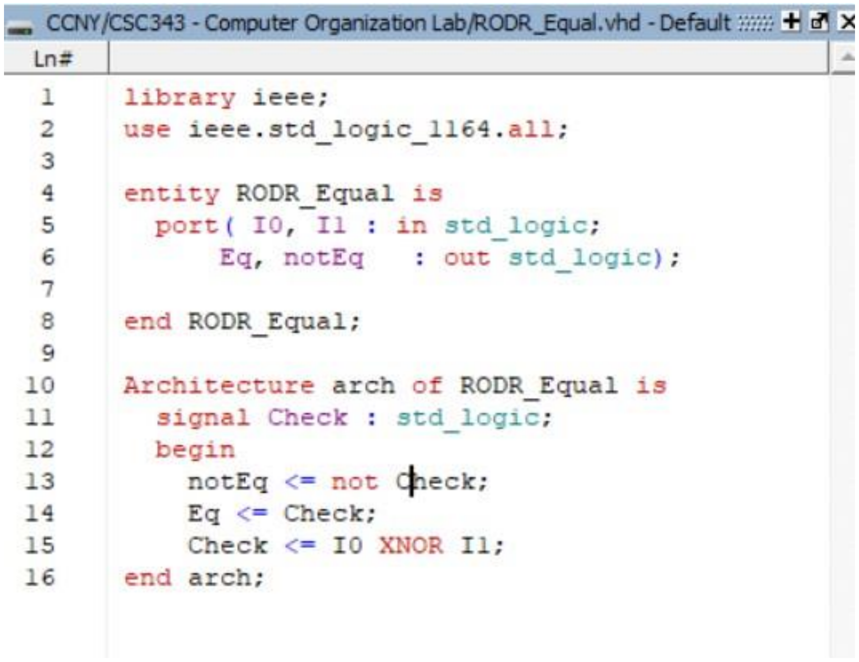
Board Specifications & Software Used

1. Altera DE2 Board (Quartus II 13.0 sp1)
2. Altera DE1-SOC Board (Quartus Prime 16.0)
3. Model Sim

Design Specifications & Functionality

1 Bit Equal Comparator

Code



CCNY/CSC343 - Computer Organization Lab/RODR_Equal.vhd - Default

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

entity RODR_Equal is
  port( I0, I1 : in std_logic;
        Eq, notEq   : out std_logic);

end RODR_Equal;

Architecture arch of RODR_Equal is
  signal Check : std_logic;
  begin
    notEq <= not Check;
    Eq <= Check;
    Check <= I0 XNOR I1;
end arch;
```

Figure 1: VHDL Code for 1-bit equal comparator

For this code, we have a 1-bit equal comparator. This will allow us to check if two 1-bit registers are equal or not. Ports **I0** and **I1** will be used as those registers. **Eq** will show if the two registers are equal or not, and **notEq** is the opposite of **Eq** and will light up on the board if the two registers are not equal.

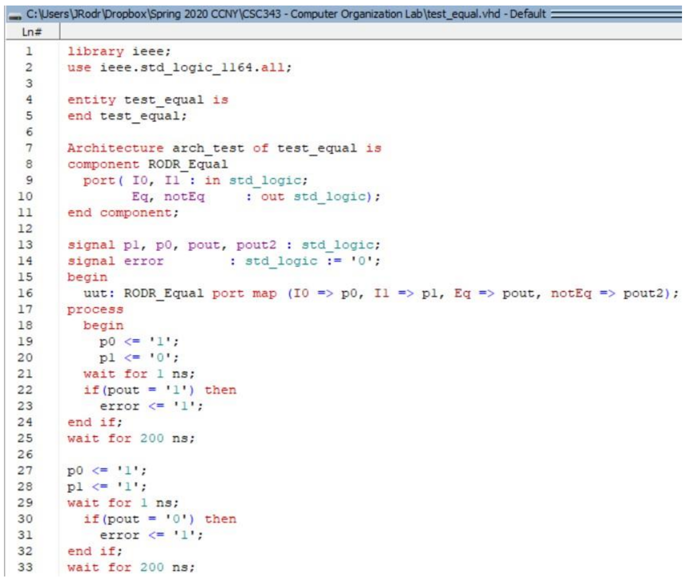
Inside the architecture, we have a signal called **Check** that will take in both registers through an XNOR gate and see if they are equal or not. That information will then get fed into both **Eq** and **notEq**, with the only difference that **Check** will be negated for **notEq**.

I0	I1	Output
0	0	1
0	1	0
1	0	0
1	1	1

Table 1: Truth Table for XNOR gate

Based on this truth table, we see that tan XNOR gate is the opposite of an XOR gate. An XNOR gate only outputs a logical '1' whenever both I0 and I1 are the same logical bit. If not, then the output of the gate is a logical '0'. This simplifies the code in the tutorial as we wouldn't need to compare all buts by using the simplest POS.

ModelSim Testbench VHDL Code



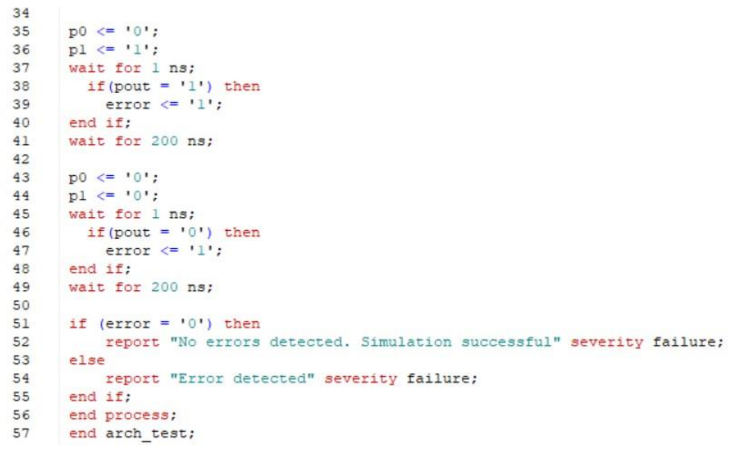
C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_equal.vhd - Default

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

entity test_equal is
end test_equal;

Architecture arch_test of test_equal is
component RODR_Equal
  port( I0, I1 : in std_logic;
        Eq, notEq     : out std_logic);
end component;

signal p1, p0, pout, pout2 : std_logic;
signal error        : std_logic := '0';
begin
  uut: RODR_Equal port map (I0 => p0, I1 => p1, Eq => pout, notEq => pout2);
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;
```

Figure 2: Testbench VHDL code for 1-bit equal comparator

Now we will use modelsim to create a testbench to check if our VHDL code works. We create a component that calls RODR_Equal for the 1-bit equal comparator. We then create signals **p0** and **p1** that will be our inputs into the component, **pout** that will be for the LED to show that the two registers are equal, and **pout2** that will light up if the two registers are not equal. Also, we have created an **error** signal that will provide for an output in the terminal on Modelsim to ensure that there are no logical errors in the running of the program.

Then, we create a process in which will change the inputs of the comparator. We call the command **wait for 1 ns** to allow for ModelSim to complete its computation and then check if any errors have occurred when inserting the numbers that will go into the registers. All, error calculation will be represented at the very end of the code. We also **wait for 200 ns** after the error signal has completed its operation to allow for a change in inputs as hold times need to be obeyed for register usage.

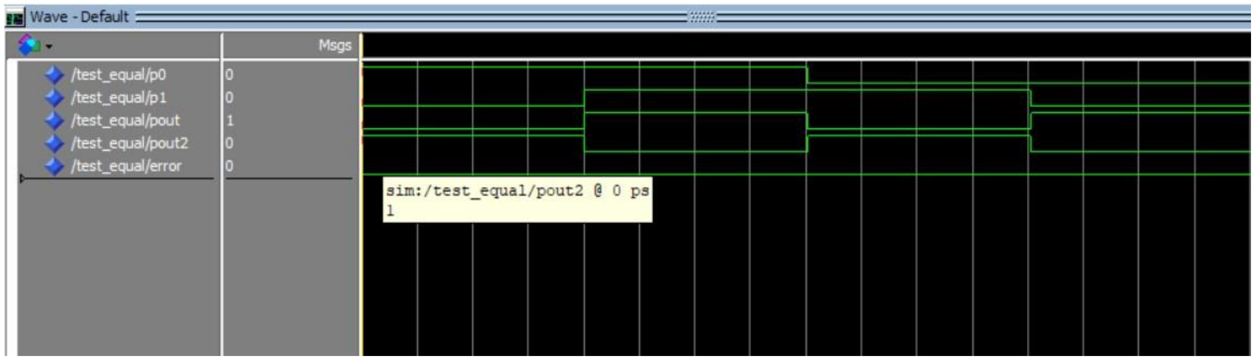


```

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;
```

Figure 3: Continuation of Testbench code for 1-bit comparator

This image is very similar to the previous one, except that now we have error handling before the process has been completed. If there has been no error in the computation of the comparator, then we will display "No errors detected. Simulation successful" on the terminal window. If there is an error, "Error detected" will be displayed in the error window.

ModelSim Waveform



```
/test_equal/p0      0
/test_equal/p1      0
/test_equal/pout    1
/test_equal/pout2   0
/test_equal/error   0
```

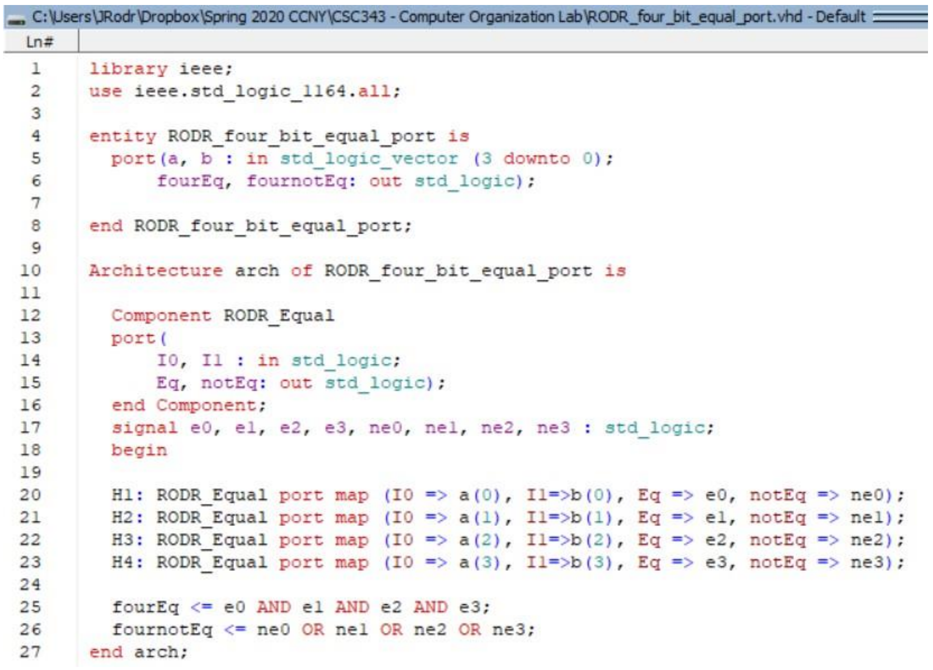
```
sim:/test_equal/pout2 @ 0 ps
1
```

Figure 4: Waveform of Testbench code for 1-bit comparator

Recall that **p0** and **p1** are the names for the two 1-bit registers, **pout** is the Equal flag that displays a '1' if both registers are equal, **pout2** is the Not Equal flag, and error that displays a '1' if a computation is made incorrectly.

In its first iteration, we see that **p0** is '1' and **p1** is **'**0'. Since they are not the same, it will then give a logical '0' to **Eq** and '1' to **notEq**. No errors were made. In its second iteration, we see that both **p0** and **p1** are logical '1'. **Eq** will be then set to '1' and **notEq** will be set to '0' without any errors. In the third iteration, we see that **p0** is '0' and **p1** is '1'. Thus, **Eq** will be set to '0' and **notEq** will be set to '1' without any errors. Lastly, **p0** and **p1** will both be set to '1' and are equal, so the **Eq** flag will be set to '1' with **notEq** being set to '0' without any errors.

4-Bit Equal Comparator


C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_four_bit_equal_port.vhd - Default

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

entity RODR_four_bit_equal_port is
  port(a, b : in std_logic_vector (3 downto 0);
      fourEq, fournotEq: out std_logic);

end RODR_four_bit_equal_port;

Architecture arch of RODR_four_bit_equal_port is

  Component RODR_Equal
  port(
      I0, I1 : in std_logic;
      Eq, notEq: out std_logic);
  end Component;
  signal e0, e1, e2, e3, ne0, ne1, ne2, ne3 : std_logic;
  begin

  H1: RODR_Equal port map (I0 => a(0), I1=>b(0), Eq => e0, notEq => ne0);
  H2: RODR_Equal port map (I0 => a(1), I1=>b(1), Eq => e1, notEq => ne1);
  H3: RODR_Equal port map (I0 => a(2), I1=>b(2), Eq => e2, notEq => ne2);
  H4: RODR_Equal port map (I0 => a(3), I1=>b(3), Eq => e3, notEq => ne3);

  fourEq <= e0 AND e1 AND e2 AND e3;
  fournotEq <= ne0 OR ne1 OR ne2 OR ne3;
end arch;
```

Figure 5: VHDL Code for 4-bit Equal Comparator

With the 4-bit equal comparator, it will inherit the same techniques created by the 1-bit comparator, except now each bit from each registers will be inserted into four different comparator checks in one VHDL file. Our ports **a** and **b** are our inputs from the 4-bit registers and **fourEq** and **fournotEq** will be our outputs and will be flags checking if both registers are equal or not.

Inside the architecture, we use the same component as was created previously called **RODR_Equal** to be used to compare each bit. Now, we have created signals **e0 through e3** to be inserted into the **Eq** output for each port map that has been created. Signals **ne0 through ne3** will be designated outputs from **notEq** for each port map. The creation of 4 port maps is needed as there are 4 bits being inserted into this comparator.

At the very end, we use a 4-to-1 AND gate for signals **e0 through e3** to ensure that all bits are the same that come out from each 1-bit comparator and insert them into **fourEq**. The exact opposite happens for **fournotEq**, using a 4-to-1 OR gate to check if one of the four ports end up having two different bits inserted in them, will this flag be activated.

I0	I1	Output
0	0	0
0	1	0
1	0	0

1	1	1

Table 2: Truth Table for AND gate

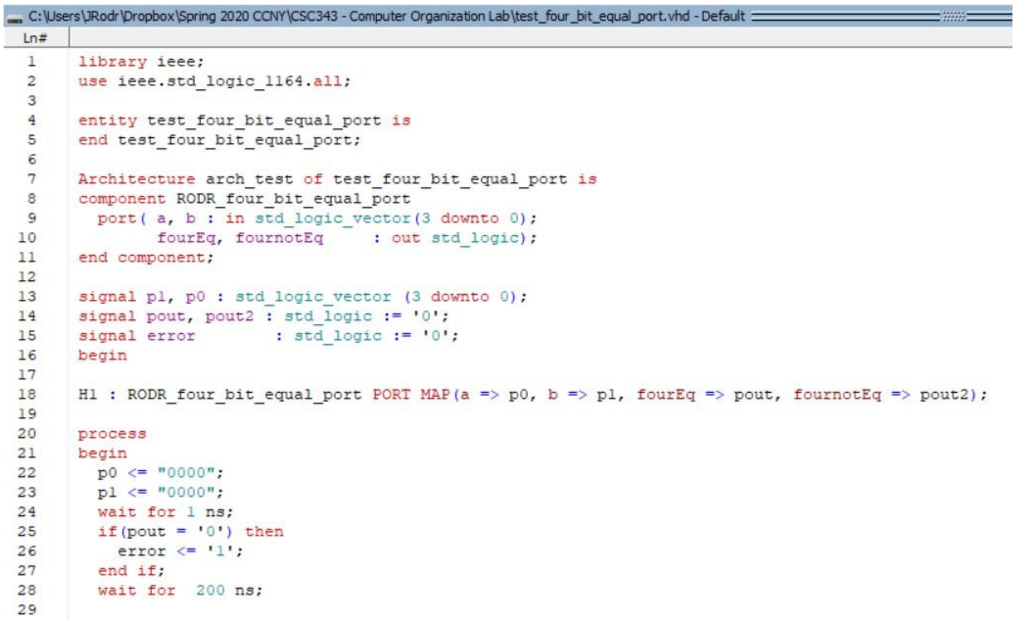
Now this may not seem like the answer we need as we also need to check for if **I0** and **I1** need to be the same with logical ‘0’. However, recall in *Table 1* for the 1-bit comparator that it is using an XNOR gate to change its output into a 1 whenever the bits are the same. So for the 4-bit comparator, all we need to do is check if all the **fourEq**’s for each bit are ‘1’ and if so, turn on its LED and disable **fournotEq**. Otherwise, both outputs will be its opposite.

I0	I1	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table 3: Truth table for OR gate

We need this for the output of **fournotEq** as it will allow for the flag to be turned on whenever just one of the 4 bits are different from both registers.

ModelSim Testbench VHDL Code

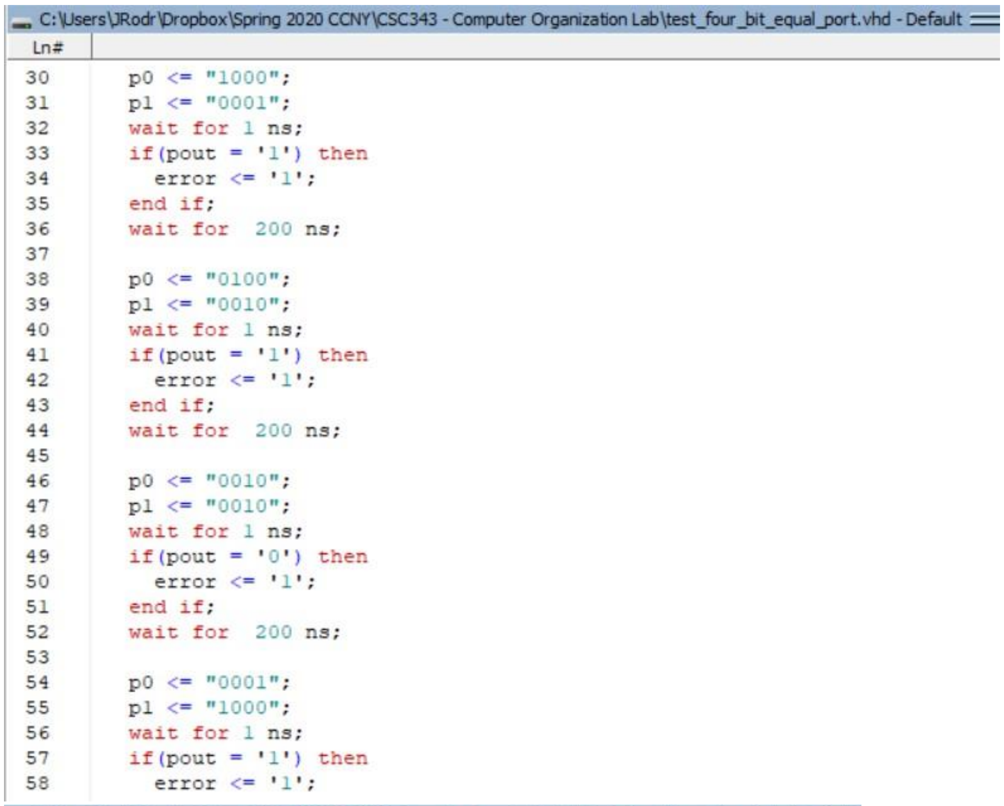


C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_four_bit_equal_port.vhd - Default

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

entity test_four_bit_equal_port is
end test_four_bit_equal_port;

Architecture arch_test of test_four_bit_equal_port is
component RODR_four_bit_equal_port
  port( a, b : in std_logic_vector(3 downto 0);
        fourEq, fournotEq     : out std_logic);
end component;

signal p1, p0 : std_logic_vector (3 downto 0);
signal pout, pout2 : std_logic := '0';
signal error        : std_logic := '0';
begin

H1 : RODR_four_bit_equal_port PORT MAP(a => p0, b => p1, fourEq => pout, fournotEq => pout2);

process
begin
  p0 <= "0000";
  p1 <= "0000";
  wait for 1 ns;
  if(pout = '0') then
    error <= '1';
  end if;
  wait for  200 ns;

```

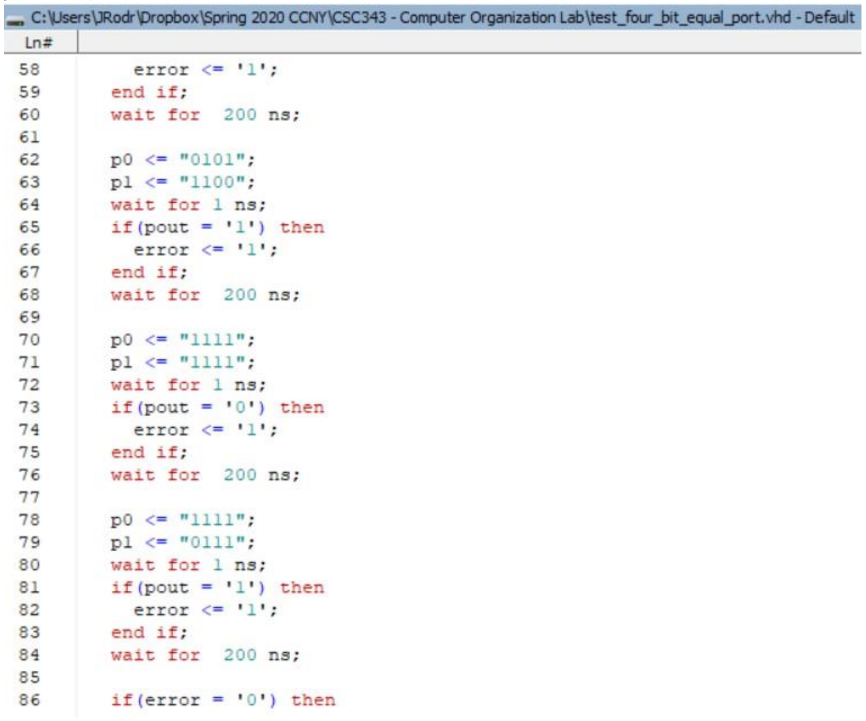
Figure 6: Part 1 of Testbench code of 4-bit equal comparator

This testbench will be very similar to that of the 1-bit equal comparator, with the only difference being that the component will be that of **RODR_four_bit_equal_port**. Other than that, the names of the in and out ports are different but function the same as **RODR_Equal**. Another key difference in this file is that both signals **p0** and **p1** are of a vector size 4. We place those into the four-bit port map as **a** and **b** inputs. Also, note that there will be taking in 4 bit sizes instead of 1. This is because the file that contains the logic for the 4-bit equal comparator has in itself 4 1-bit comparators that will check each bit.



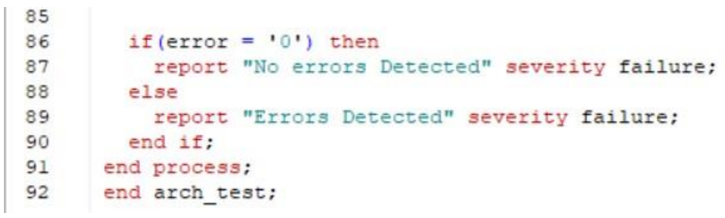
C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_four_bit_equal_port.vhd - Default

```
    p0 <= "1000";
    p1 <= "0001";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "0100";
    p1 <= "0010";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "0010";
    p1 <= "0010";
    wait for 1 ns;
    if(pout = '0') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "0001";
    p1 <= "1000";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
```



C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_four_bit_equal_port.vhd - Default

```
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "0101";
    p1 <= "1100";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "1111";
    p1 <= "1111";
    wait for 1 ns;
    if(pout = '0') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "1111";
    p1 <= "0111";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    if(error = '0') then
```



```

    if(error = '0') then
      report "No errors Detected" severity failure;
    else
      report "Errors Detected" severity failure;
    end if;
  end process;
  end arch_test;
```

Figure 7: Rest of Testbench Code for 4-bit Comparator

The rest of this code remains the same as the testbench for the 1-bit comparator with only a difference in the extension of **p0** and **p1** into being 4 bits long.

ModelSim Waveform

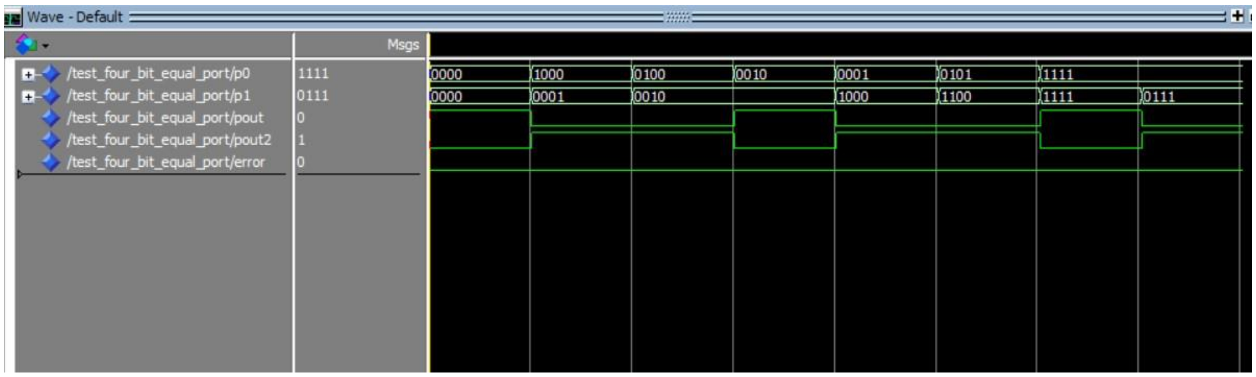
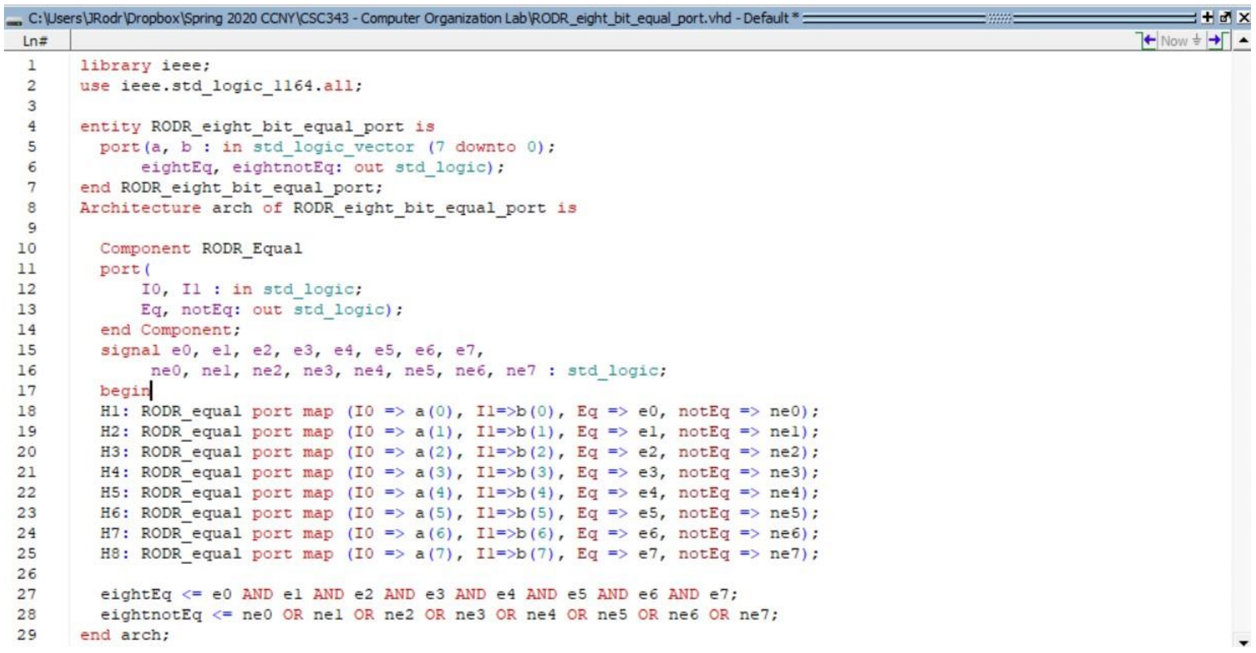

Wave - Default
Msgs
/test_four_bit_equal_port/p0
/test_four_bit_equal_port/p1
/test_four_bit_equal_port/pout
/test_four_bit_equal_port/pout2
/test_four_bit_equal_port/error
1111
0111
0
1
0
0000
1000
0100
0010
0001
0101
1111
0000
0001
0010
1000
1100
1111
0111

Figure 8: Waveform for 4-bit comparator

In this waveform, we use ModelSim to use our testbench to create these waveform models. As we can see here, all the numbers that are the same will have **pout** being high while if not, **pout** will be ‘0’. Also, not that **pout2** will always be the opposite of **pout** as they are **notEq** and **Eq**, respectively in their port maps as shown in *figure 6*. There were no errors being generated from this testbench, meaning that all calculations are correct.

8-bit Equal Comparator

Code

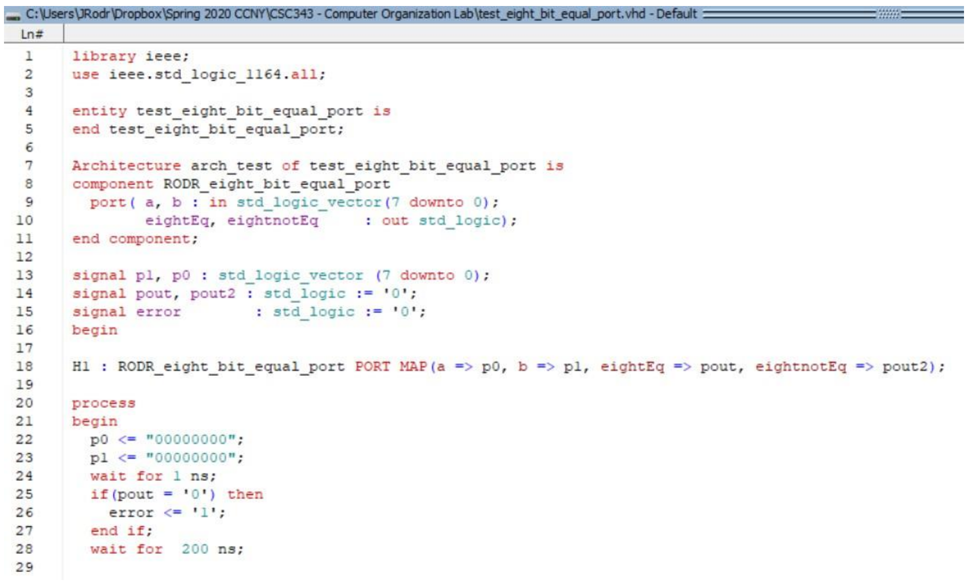

C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_eight_bit_equal_port.vhd - Default *
Ln#
Now

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

entity RODR_eight_bit_equal_port is
  port(a, b : in std_logic_vector (7 downto 0);
      eightEq, eightnotEq: out std_logic);
end RODR_eight_bit_equal_port;
Architecture arch of RODR_eight_bit_equal_port is

  Component RODR_Equal
  port(
      I0, I1 : in std_logic;
      Eq, notEq: out std_logic);
  end Component;
  signal e0, e1, e2, e3, e4, e5, e6, e7,
       ne0, ne1, ne2, ne3, ne4, ne5, ne6, ne7 : std_logic;
  begin
  H1: RODR_equal port map (I0 => a(0), I1=>b(0), Eq => e0, notEq => ne0);
  H2: RODR_equal port map (I0 => a(1), I1=>b(1), Eq => e1, notEq => ne1);
  H3: RODR_equal port map (I0 => a(2), I1=>b(2), Eq => e2, notEq => ne2);
  H4: RODR_equal port map (I0 => a(3), I1=>b(3), Eq => e3, notEq => ne3);
  H5: RODR_equal port map (I0 => a(4), I1=>b(4), Eq => e4, notEq => ne4);
  H6: RODR_equal port map (I0 => a(5), I1=>b(5), Eq => e5, notEq => ne5);
  H7: RODR_equal port map (I0 => a(6), I1=>b(6), Eq => e6, notEq => ne6);
  H8: RODR_equal port map (I0 => a(7), I1=>b(7), Eq => e7, notEq => ne7);

  eightEq <= e0 AND e1 AND e2 AND e3 AND e4 AND e5 AND e6 AND e7;
  eightnotEq <= ne0 OR ne1 OR ne2 OR ne3 OR ne4 OR ne5 OR ne6 OR ne7;
end arch;
```

Figure 9: VHDL Code for 8-Bit Equal Comparator

In this section, we have the code for the 8-bit equal comparator. We use ports **a** and **b** for taking in 8-bit vector inputs from each incoming register. Ports **eightEq** and **eightnotEq** will be used as flags for determining whether the two registers are equal or not, respectively. Inside the architecture, we will use the **RODR_Equal** component and create 8 instances of it (one for every bit sequence up to 8 bits). Each port map will take inputs as every bit from the least significant bit from the most significant and output to **Eq** for signals **e0 through e7** and **notEq** for **ne0 through ne7**. Those outputs would then go towards **eightEq** and **eightnotEq**. Their methodology is the same as the 4-bit equal comparator, except that they have been extended to have 8-to-1 AND/OR gates to accommodate for the 8 bits they are taking in.

Testbench VHDL Code



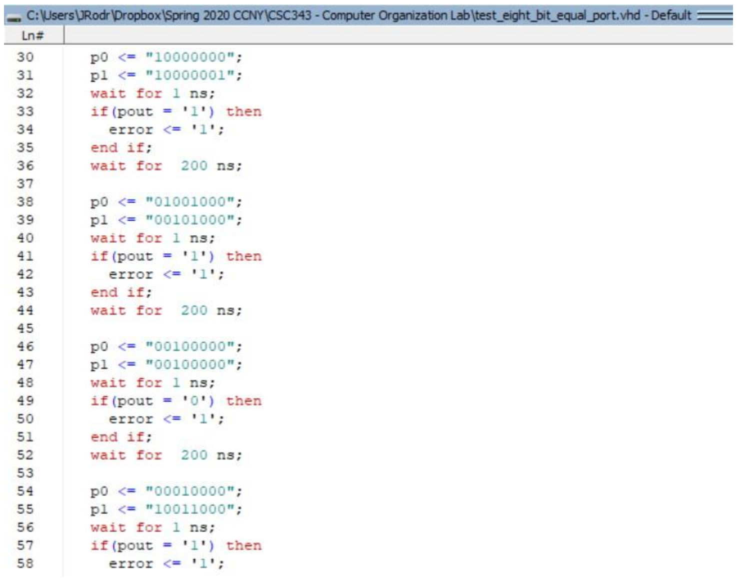
C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_eight_bit_equal_port.vhd - Default

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

entity test_eight_bit_equal_port is
end test_eight_bit_equal_port;

Architecture arch_test of test_eight_bit_equal_port is
component RODR_eight_bit_equal_port
  port( a, b : in std_logic_vector(7 downto 0);
        eightEq, eightnotEq     : out std_logic);
end component;

signal p1, p0 : std_logic_vector (7 downto 0);
signal pout, pout2 : std_logic := '0';
signal error        : std_logic := '0';
begin

H1 : RODR_eight_bit_equal_port PORT MAP(a => p0, b => p1, eightEq => pout, eightnotEq => pout2);

process
begin
  p0 <= "00000000";
  p1 <= "00000000";
  wait for 1 ns;
  if(pout = '0') then
    error <= '1';
  end if;
  wait for  200 ns;

```

Figure 10: First part of testbench code for 8-bit Equal Comparator

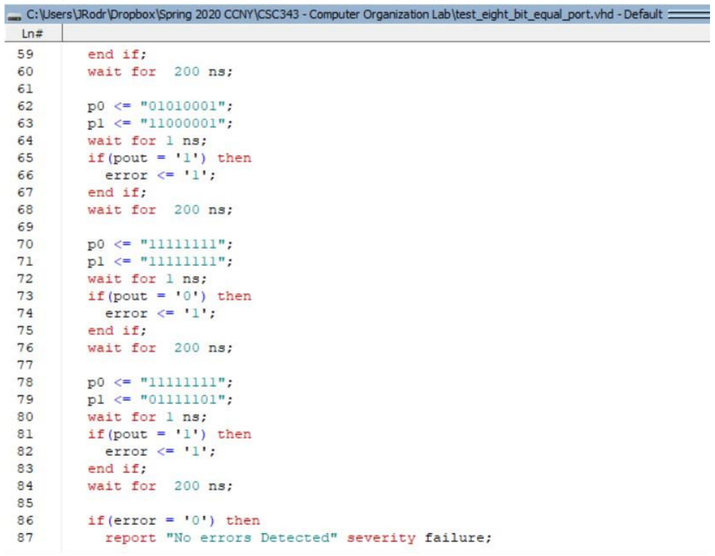
This test bench is the exact same as the previous 4-bit equal comparator, with some minor changes. From *figure 10*, we can see that both **p1** and **p0** now has been vectorized to accommodate for 8 bits. This will allow us to only use one port map that will have a check for every bit using **RODR_Equal**. The outputs of the system will be **pout** (for eightEq) and **pout2** (for eightnotEq).

We then begin a process that is very similar to the previous comparator, except that **p0** and **p1** have now been extended to accommodate for eight bits.



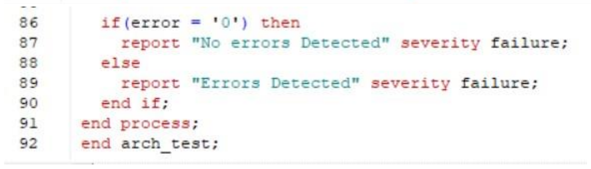
C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_eight_bit_equal_port.vhd - Default

```
    p0 <= "10000000";
    p1 <= "10000001";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "01001000";
    p1 <= "00101000";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "00100000";
    p1 <= "00100000";
    wait for 1 ns;
    if(pout = '0') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "00010000";
    p1 <= "10011000";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
```



C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_eight_bit_equal_port.vhd - Default

```
    end if;
    wait for  200 ns;

    p0 <= "01010001";
    p1 <= "11000001";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "11111111";
    p1 <= "11111111";
    wait for 1 ns;
    if(pout = '0') then
      error <= '1';
    end if;
    wait for  200 ns;

    p0 <= "11111111";
    p1 <= "01111101";
    wait for 1 ns;
    if(pout = '1') then
      error <= '1';
    end if;
    wait for  200 ns;

    if(error = '0') then
      report "No errors Detected" severity failure;
```



```
    if(error = '0') then
      report "No errors Detected" severity failure;
```

```
    else
      report "Errors Detected" severity failure;
    end if;
  end process;
end arch_test;
```

Figure 11: Rest of testbench code for 8 bit Equal Comparator

The rest of this code is the same as every previous testbench with the only changes being the bit sequences inserted as a testbench to ensure that the comparators work correctly.

ModelSim Waveform

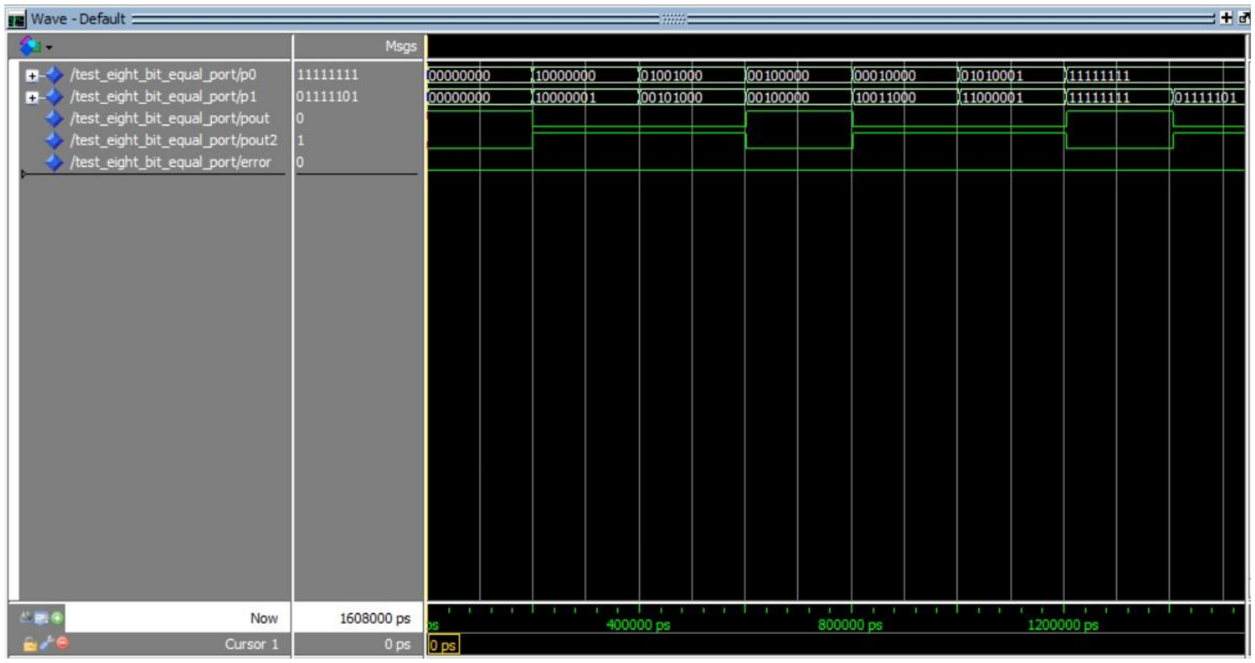


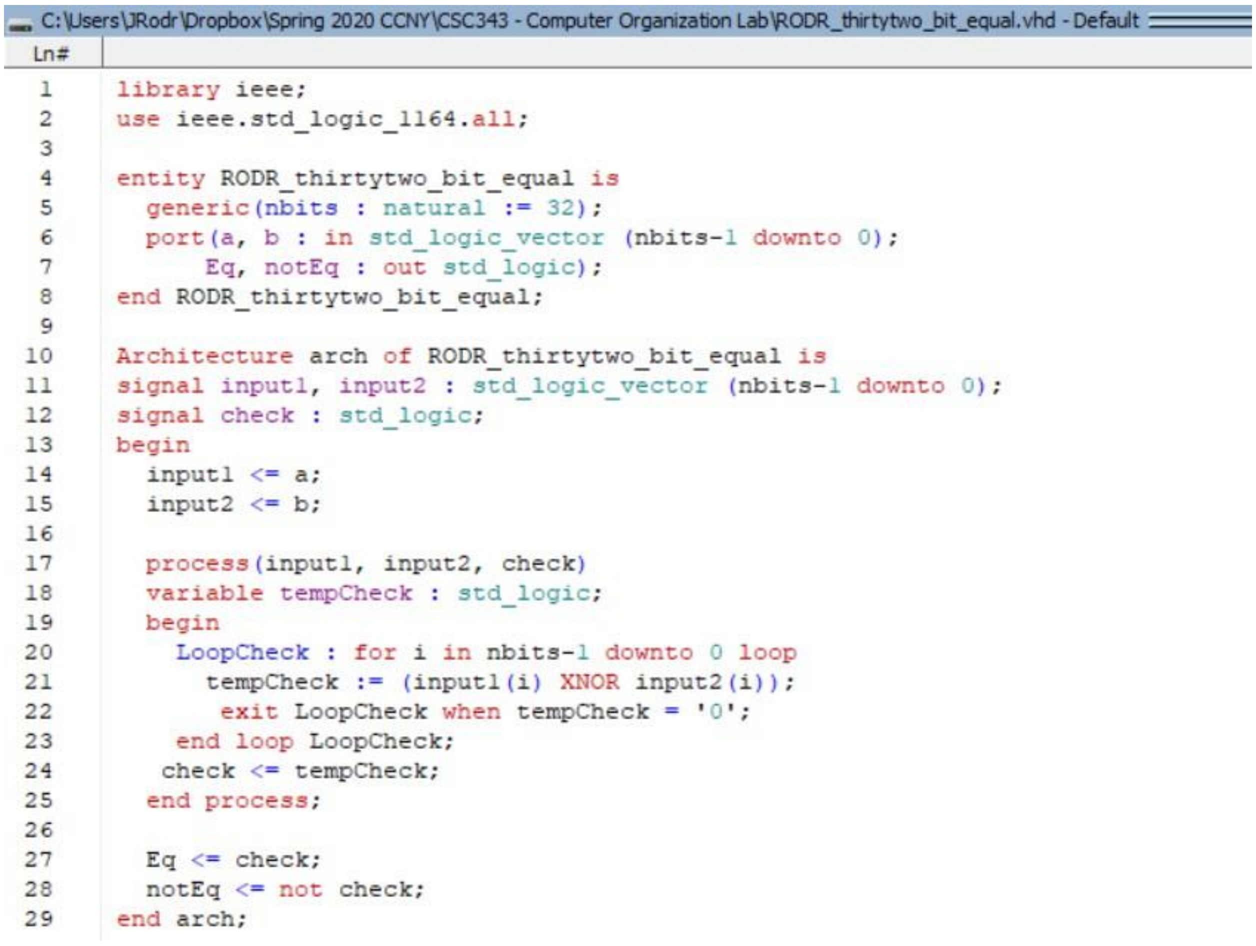
Figure 12: Waveform for 8-bit Comparator

Based on this waveform, we see the waveforms created by the testbench. Note that **p0** and **p1** are the vector inputs from the registers, **pout** is the Eq flag, **pout2** is the notEq flag, and error checks for any errors in the computation made by the testbench itself.

The waveforms are correct as for every bit sequence has been checked from the least significant to the most significant and properly puts Eq into the right state. As notEq is the opposite of Eq, it will always be correct as it is corrected via a NOT gate. Also shown in this waveform is that there are no errors in computation.

32-Bit Equal Comparator

Code

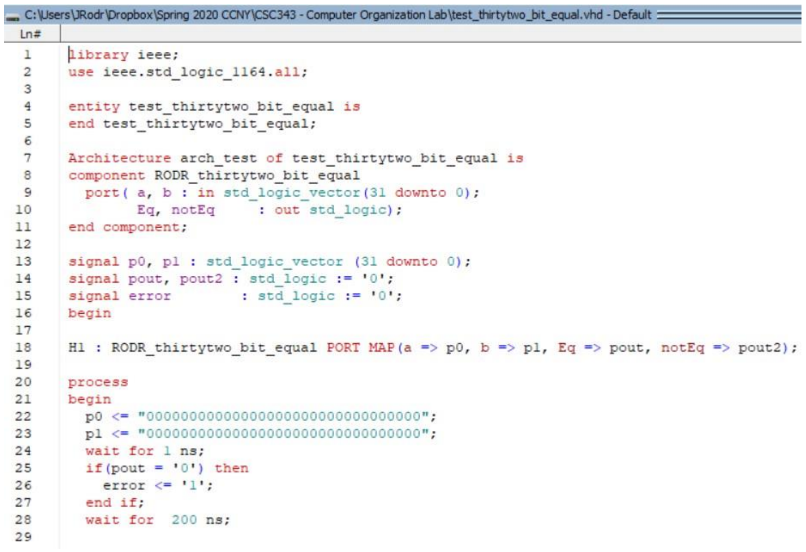

C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\RODR_thirtytwo_bit_equal.vhd - Default

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

entity RODR_thirtytwo_bit_equal is
  generic(nbits : natural := 32);
  port(a, b : in std_logic_vector (nbits-1 downto 0);
      Eq, notEq : out std_logic);
end RODR_thirtytwo_bit_equal;

Architecture arch of RODR_thirtytwo_bit_equal is
signal input1, input2 : std_logic_vector (nbits-1 downto 0);
signal check : std_logic;
begin
  input1 <= a;
  input2 <= b;

  process(input1, input2, check)
  variable tempCheck : std_logic;
  begin
    LoopCheck : for i in nbits-1 downto 0 loop
      tempCheck := (input1(i) XNOR input2(i));
       exit LoopCheck when tempCheck = '0';
    end loop LoopCheck;
   check <= tempCheck;
  end process;

  Eq <= check;
  notEq <= not check;
end arch;
```

Figure 13: Code for 32-Bit Equal Comparator

This code is different from the others that I have created. Instead of always using the RODR_Equal component to check through each bit, we will do so directly. The ports defined are the same as the previous components. This VHDL code was initially created to accommodate for n-bit comparison. We will store the inputs of **a** and **b** into signals **input1** and **input2**, respectively and use a **check** to see whether both registers are equal or not. In the process defined below, we have created a for loop to iterate through each bit and use an XNOR gate to check if both bits are the same or not. The loop will only terminate when the temporary variable called **tempCheck** is ‘0’. If it never is ‘0’, then we will just take the value ‘1’ and set it to **Eq**, meaning that both registers have the exact same bit sequence. If the loop terminates before it completes, then **tempCheck** is ‘0’ and thus places that into **Eq**, meaning that both registers are not equal. **notEq** will always be the opposite of **Eq**, so they would be ‘0’ when **Eq** is ‘1’, and ‘1’ when **Eq** is ‘0’.

VHDL Testbench Code

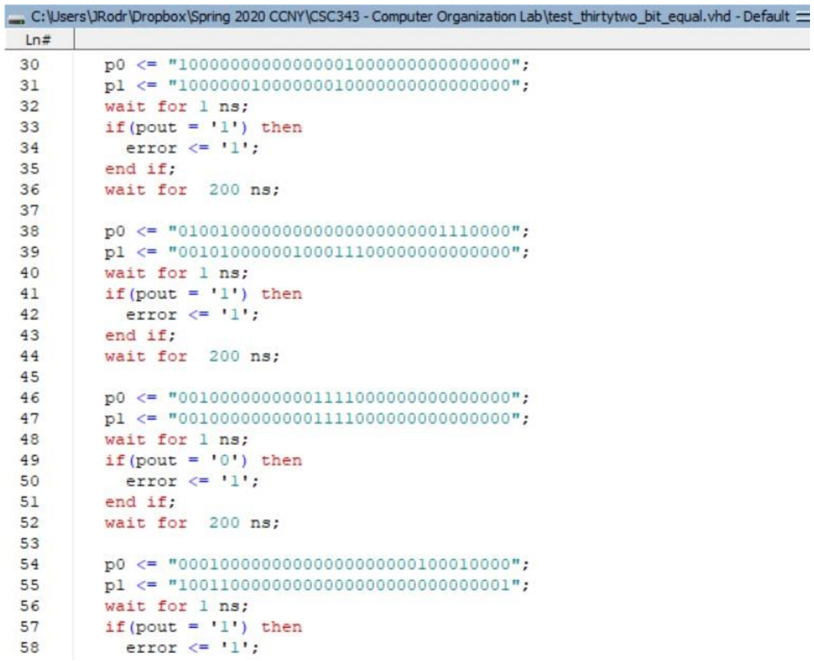


C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_thirtytwo_bit_equal.vhd - Default

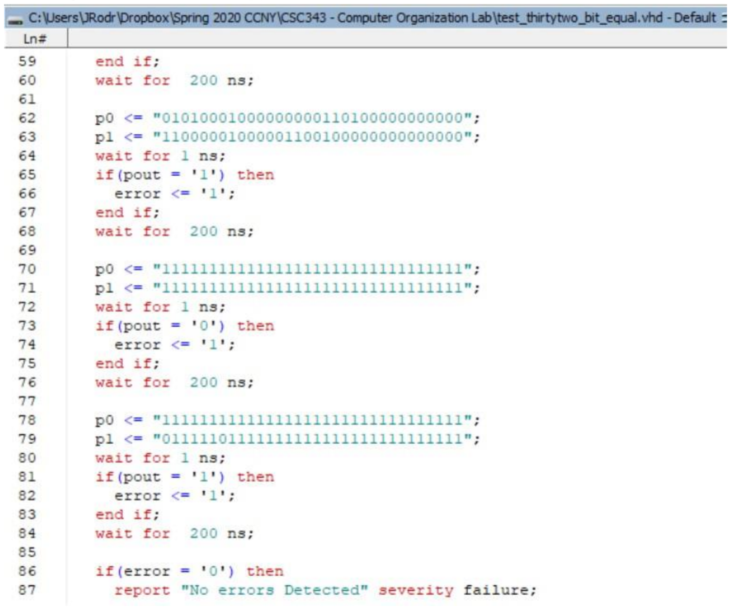
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity test_thirtytwo_bit_equal is
end test_thirtytwo_bit_equal;

Architecture arch_test of test_thirtytwo_bit_equal is
component RODR_thirtytwo_bit_equal
  port( a, b : in std_logic_vector(31 downto 0);
        Eq, notEq     : out std_logic);
end component;

signal p0, p1 : std_logic_vector (31 downto 0);
signal pout, pout2 : std_logic := '0';
signal error        : std_logic := '0';
begin

H1 : RODR_thirtytwo_bit_equal PORT MAP(a => p0, b => p1, Eq => pout, notEq => pout2);

process
begin
  p0 <= "00000000000000000000000000000000";
  p1 <= "00000000000000000000000000000000";
  wait for 1 ns;
  if(pout = '0') then
    error <= '1';
  end if;
  wait for  200 ns;

```

Figure 14: First part of testbench code for 32-Bit Equal Comparator.

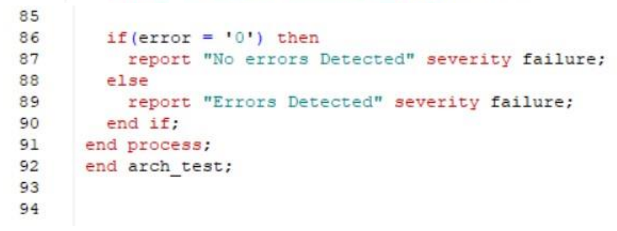
Again, the code for the testbench is very similar to the previous equal comparators. The only major difference is that now we are testing for 32-bits and now **p0** and **p1** will hold 32-bit logic. We are also no longer using **RODR_Equal** internally in our 32-bit equal comparator as it was initially created to work for n-bits.


C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_thirtytwo_bit_equal.vhd - Default

```
  p0 <= "10000000000000001000000000000000";
  p1 <= "10000001000000010000000000000000";
  wait for 1 ns;
  if(pout = '1') then
    error <= '1';
  end if;
  wait for  200 ns;

  p0 <= "01001000000000000000000001110000";
  p1 <= "00101000000100011100000000000000";
  wait for 1 ns;
  if(pout = '1') then
    error <= '1';
  end if;
  wait for  200 ns;

  p0 <= "00100000000001111000000000000000";
  p1 <= "00100000000001111000000000000000";
  wait for 1 ns;
  if(pout = '0') then
    error <= '1';
  end if;
  wait for  200 ns;

  p0 <= "00010000000000000000000100010000";
  p1 <= "10011000000000000000000000000001";
  wait for 1 ns;
  if(pout = '1') then
    error <= '1';
```


C:\Users\JRodr\Dropbox\Spring 2020 CCNY\CSC343 - Computer Organization Lab\test_thirtytwo_bit_equal.vhd - Default

```
  end if;
  wait for  200 ns;

  p0 <= "01010001000000011010000000000000";
  p1 <= "11000001000001100100000000000000";
  wait for 1 ns;
  if(pout = '1') then
    error <= '1';
  end if;
  wait for  200 ns;

  p0 <= "11111111111111111111111111111111";
  p1 <= "11111111111111111111111111111111";
  wait for 1 ns;
  if(pout = '0') then
    error <= '1';
  end if;
  wait for  200 ns;

  p0 <= "11111111111111111111111111111111";
  p1 <= "01111101111111111111111111111111";
  wait for 1 ns;
  if(pout = '1') then
    error <= '1';
  end if;
  wait for  200 ns;

  if(error = '0') then
    report "No errors Detected" severity failure;
```



```

  if(error = '0') then
    report "No errors Detected" severity failure;
  else
    report "Errors Detected" severity failure;
  end if;
end process;
end arch_test;


```

Figure 15: Rest of testbench code for 32-bit equal comparator

As we can see here, the code structure has not changed much from the previous comparators except for the bit extension to both **p0** and **p1** that will accommodate for 32 bits. The error detection system remains the same by analyzing if **pout** provides the correct answer or not.

ModelSim Waveform

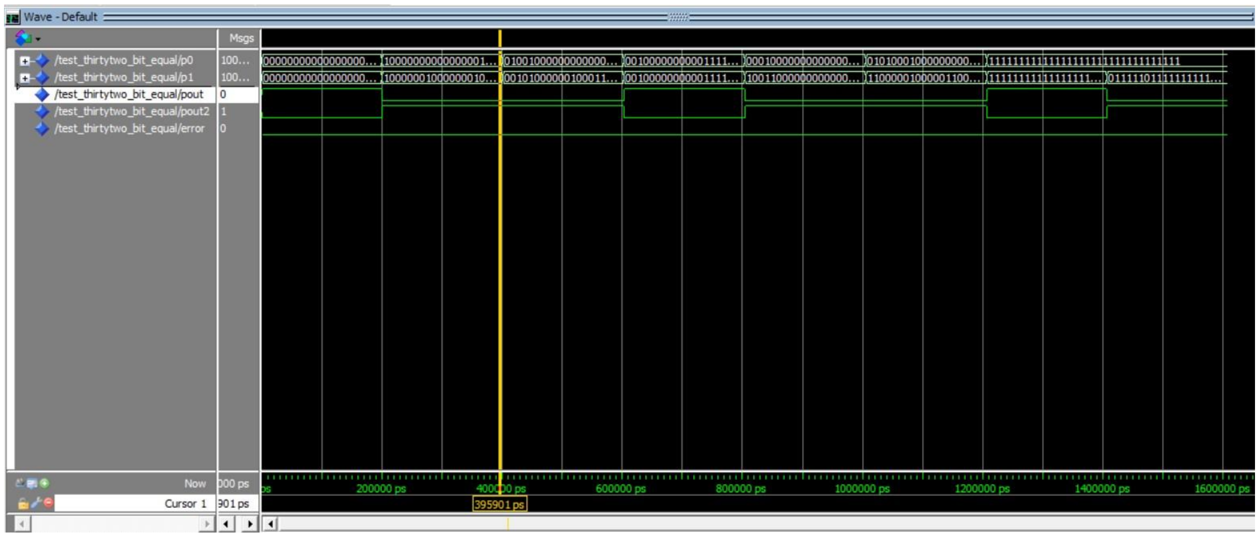

Wave - Default
Msgs
/test_thirtytwo_bit_equal/p0
/test_thirtytwo_bit_equal/p1
/test_thirtytwo_bit_equal/pout
/test_thirtytwo_bit_equal/pout2
/test_thirtytwo_bit_equal/error
Now
Cursor 1
395901 ps

Figure 16: Waveform for 32-bit Equal Comparator

In this waveform, we use ModelSim to use our testbench to create these waveform models. As we can see here, all the numbers that are the same will have **pout** being high while if not, **pout** will be ‘0’. Also, not that **pout2** will always be the opposite of **pout** as they are **notEq** and **Eq**. There were no errors being generated from this testbench, meaning that all calculations are correct.

Conclusion

In this laboratory exercise, I have learned how to create a bitwise comparator that will check if two incoming registers are equal or not. My methodology was to XNOR every bit from each register and have them check if they’re the same instead of checking every single combination using POS and then using an AND gate to ensure that they’re all the same. Instead, I do them bit by bit and then as an output, it will be a logical ‘1’ if the XNOR’s all result in ‘1’. However, the notEq output will be the opposite and will activate whenever one of the XNOR gates equal ‘0’ and thus change both Eq and notEq (Eq having logical ‘0’ and notEq having a logical ‘1’). Also, we could have used a simple vector math and simply checked if they were equal using relational operators. This kind of comparator can be used whenever there is a need to detect copies of the same data in different registers and potentially change the state of one of them later on for a different use.