Computer systems engineering Department.

Advanced Digital Systems Design (ENCS3310).

Project Report.

Student Name: Mahmoud Qaisi

Student ID: 1190831

Instructor: Abdallatif Abuissa.

Section: 2

Date: 26/12/2021

- **Abstract:**

In this Project an 8-bit comparator will be implemented using VHDL structural description. The implementation will be divided into two stages as each stage will implement the comparator in a different method the first stage is implementing the comparator using a ripple adder. The second stage is implementing it using a magnitude comparator.

Contents:

• Figures:

• Tables:

• Introduction:

There are two approaches to compare two numbers in binary. The first one is to subtracts the second number from the first and analyze the result. The second is to compare the magnitude and sign of each number.

• The Gates:

The first step is to prepare the library of gates that will help build this project structurally.

1. The inverter:

The inverter is a logic gate that takes one input (x) and produces one output (y) where y = x' with a determined delay of 5 ns.

x	y
0	1
1	0

Table 1: NOT truth table

2. And Gate:

The and gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 and x2 … and xn with a determined delay of 7 ns.

X1	X2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: And2 Truth Table

3. Or Gate:

The or gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 or x2 … or xn with a determined delay of 7 ns.

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3: OR2 Truth Table

4. Nand Gate:

The Nand gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 nand x2 … nand xn with a determined delay of 5 ns. And it produces exactly the opposite outputs to The And gate.

X1	X2	y
0	0	1
0	1	1
1	0	1
1	1	0

Table 4: NAND truth table.

5. Nor Gate:

The Nand gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 nor x2 … nor xn with a determined delay of 5 ns. And it produces exactly the opposite outputs of The Or gate.

X1	X2	y
0	0	1
0	1	1
1	0	1
1	1	0

Table 5: NOR truth table.

6. Xnor Gate:

The Xnor gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 xnor x2 … xnor xn with a determined delay of 9 ns. It's also known as the even function where it produces 1 when the number of inputs equal to 1 is even.

X1	X2	y
0	0	1
0	1	0
1	0	0
1	1	1

Table 6: XNOR truth table.

7. Xor Gate:

The Xor gate is a logic gate that takes n inputs (x1, x2…, xn) and produces one output (y) where y = x1 xor x2 … xor xn with a determined delay of 12 ns. It's also known as the odd function where it produces 1 when the number of inputs equal to 1 is odd.

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 7: XOR: truth table.

• Components:

The are the components needed to design the stages of our project. They will be designed with the gates that were built earlier.

a) The half adder:

Which is a combinational digital circuit that adds two inputs together and produces a sum and a carry out.

X1	X2	sum	C-out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 8:Half Adder truth table.

After solving the K-map for the truth table above, It is found that sum = x1 xor x2 and C-out = x1 and x2.

b) The full adder:

Which is a combinational digital circuit that adds two inputs together along with a carry in bit to produces a sum and a carry out.

X1	X2	C-in	sum	C-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 9: Full Adder truth table.

The full adder can be designed using two half adders with an Or Gate between the carry out of each half adder. We can derive this from the results of the k map from the table above.

Sum = x1 xor x2 xor C-in. We can group the x1 and x2 together, thus we get the sum of the first half adder. And then C-in is entered to the second half adder along with the sum of the first half adder into the second half adder.

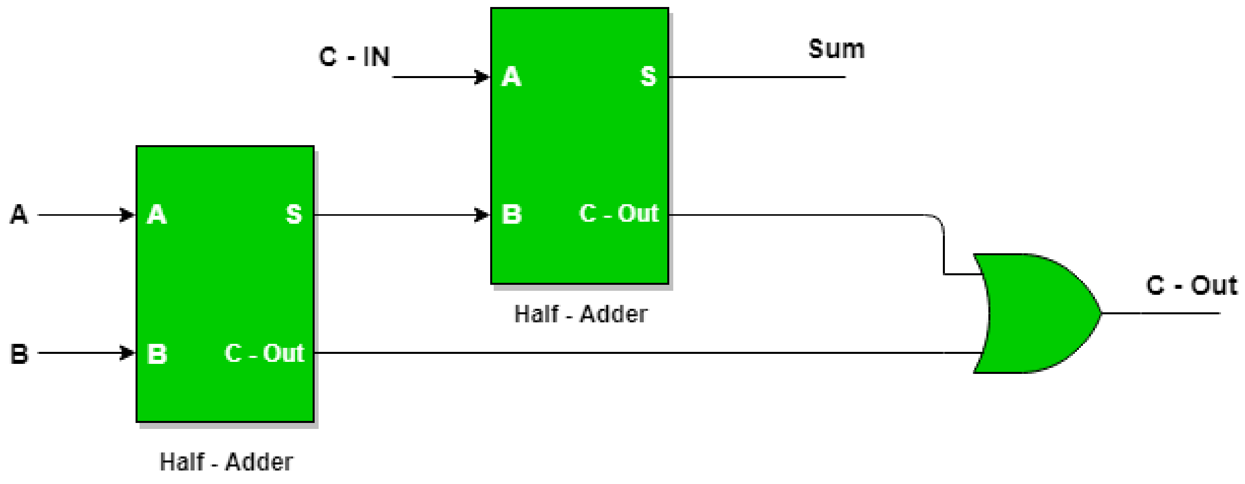


Figure 1: FULL adder design

c) 8-Bits Adder:

We need this adder to implement the first stage where the two compared numbers will be subtracted from each other. In order to design this adder, a full adder will be used to add the Carry in with the first bit and another to add the first bit with the second bit and so on until the last bit. The Carry out of each full adder will represent the Carry in of the adder next to it. An the carry out of the last full adder will represent the main Carry out which is an output from the circle.

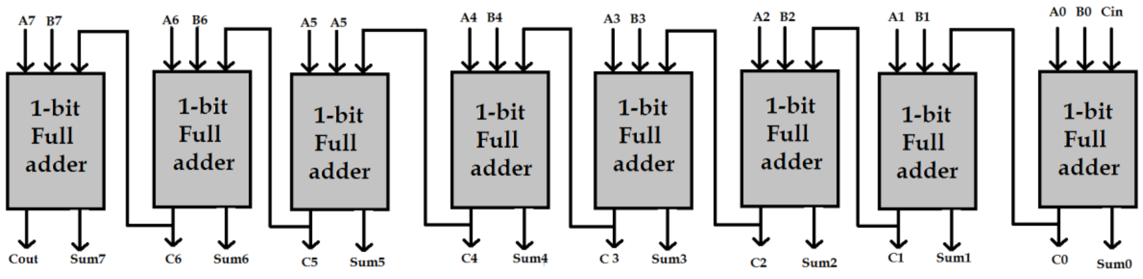


Figure 2:8-bit Adder design.

d) 8-Bit Subtractor:

In order to subtract two numbers in binary, the 2’s complement of the second number should be found. 2’s complement can be obtained be inverting all the bits of the number and then adding one. This can be achieved by entering each bit into an Xor gate with one port fixed on 1. This produces the complement of the number. And the 1 is added to the 2’s complement via the carry in of the 8-Bit Adder.

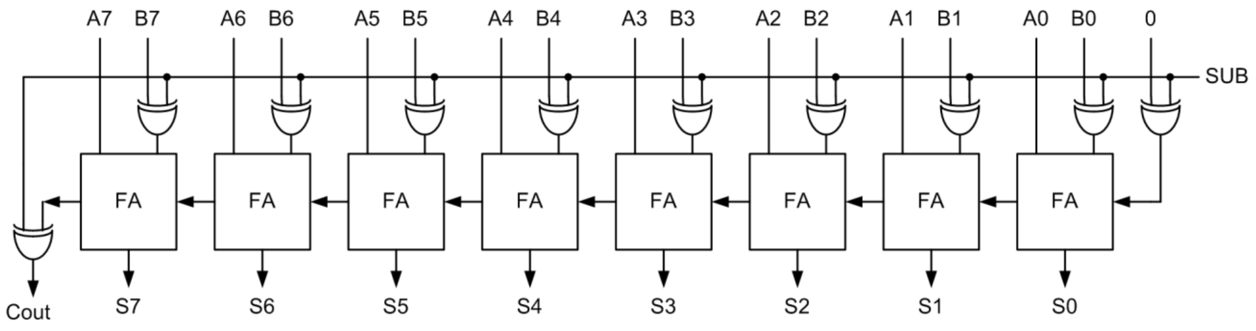


Figure 3: 8-Bit Subtractor

e) 8x3 Priority Encoder:

The Encoder is a combinational Digital Circuit that has 2^n inputs and n outputs. It produces the output according to the input with value one that has the highest priority. So if bit number n is ‘1’ and bit number n+2 is also one then the output will be n+2.

f) Bit Comparator:

It is a combinational circuit with 2 inputs x and y with 1 bit each, and three outputs b(bigger), e(equal) and s (smaller). It compares x to y. if x is bigger then b is 1 and the rest is 0. But if x is the sign bit the results would differ. That’s why an s bit was added to indicate if this bit is the sign bit or not.

s	x	y	b	e	s
0	0	0	0	1	0
0	0	1	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	1	1
1	1	1	0	1	0

Figure 4:Bit Comparator Truth Table

g) Mux8x1:

It is a combinational Circuit with 2^n inputs, one output and n selection lines. The output produced will be equal to the value of the line pointed to by the selection lines.

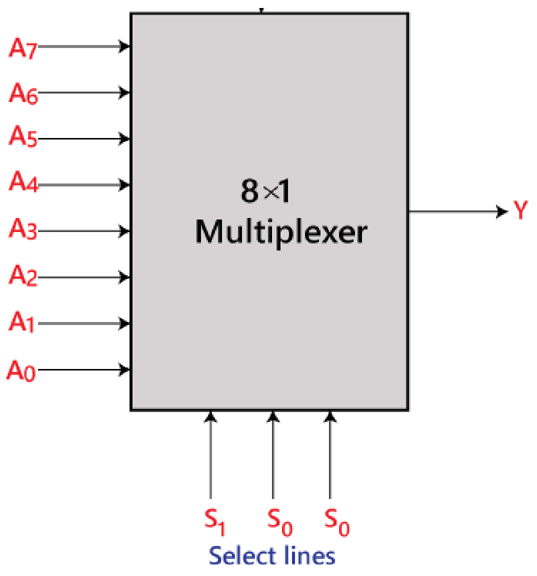


Figure 5:MUX 8x1

h) D-flip flop:

It is a digital circuit with one input and one output along with a clock input that defines when the value of the input will be transferred to the output. It is usually used to delay the output of a circuit.

• Design Approach:

After building all the necessary components for the required project all that is left is to assemble the pieces together.

• Stage 1:

The task of Stage 1 is to design a comparator using a ripple adder. After entering the values of x and y (the values to be compared), an output will be produced which is the difference between them. After the difference is produced, it will be checked if all of its bits are equal to zero using a nor gate with 8 inputs. If all the bits are zero then the output of e will be 1. To check if the number is negative, we check the last bit of the difference if it is 1 that means it's negative and the number is smaller then s is set to 1. There is one case where this doesn't apply which is when there is an overflow, which is when the resulting output is positive, the first number is negative and the second number is positive. To solve this particular case, The first bit of each number is compared to check if this case applies then s will be set to one. b is found by checking if e and s are both zero by using a nand gate.

This method to compare two numbers has two major issues. The first is the huge delay between the input and producing a stable output is major. And it is due to using the ripple adder which creates a delay equal to the delay of each full adder used multiplied by the number of full adders used. It can be solved by using the lookahead adder which will decrease the delay. The second issue is the case of overflow while subtracting. Which can be solved by handling each individual case where it occurs but is might be hard to cover them all.

• Stage 2:

The second method to compare the two numbers is to compare the signs and magnitude of each number. The approach taken here is to use a bit wise xor between the two numbers. The resulting number will be entered into a priority encoder to define what bit will be the determining bit in comparing these two numbers. The Encoder will output the number of the bit and an output v that will define if it the sign bit or not. Then this number we got will be entered into the selection lines of two muxes, each one contains one of the two numbers. The output of these muxes will be the two defining bits we determined from the encoder. Then both of these bits and v that we got earlier will be entered into our special one-bit-comparator. This comparator will produce the desired output of b,e,s.

Example:

X = '00101101', Y = '00011011'.

Bitwise Xor = '00110110'

Priority Encoder output = '101' sign-bit checker '0'.

Mux1 output = X(5) = '1' Mux1 output = Y(5) = '0'

One-Bit-Comp output = (B = 1, E = 0, S = 0).

This method reduces the delay from stage 1 and doesn't have the overflow case. So it can be generally better and faster method to compare two number in binary.

• Simulation:

In order to check the results of the designed circuits the following components were designed:

1) Test Generator:

It is a circuit that will generate all the possible values that can go into x and y. It will also generate the theoretical values that the implemented circuit is supposed to produce.

2) Test Analyzer:

This circuit will compare the results generated from the test generator with the results from the implemented circuited.

3) Simulation Results:

- Stage 1:

After testing multiple values on the circuit, it was found that the maximum circuit latency is about 155 ns. So, the clock cycle was set to 160 ns to eliminate all possible errors.

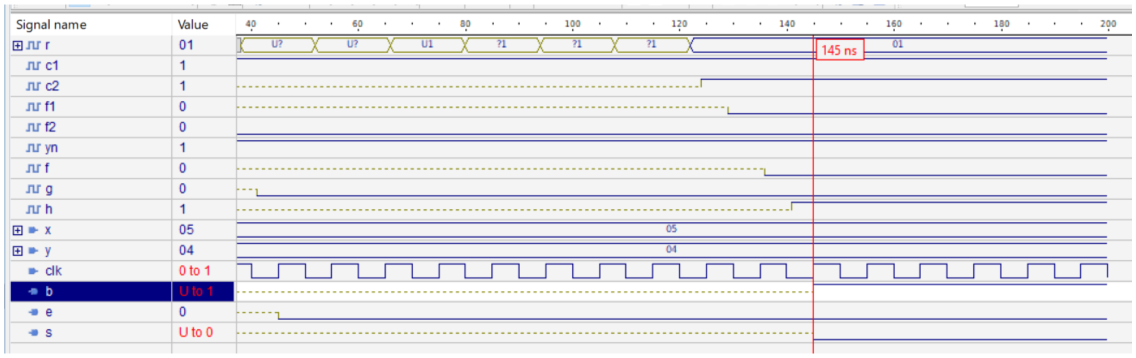


Figure 6: Stage 1 Simulation try (1).

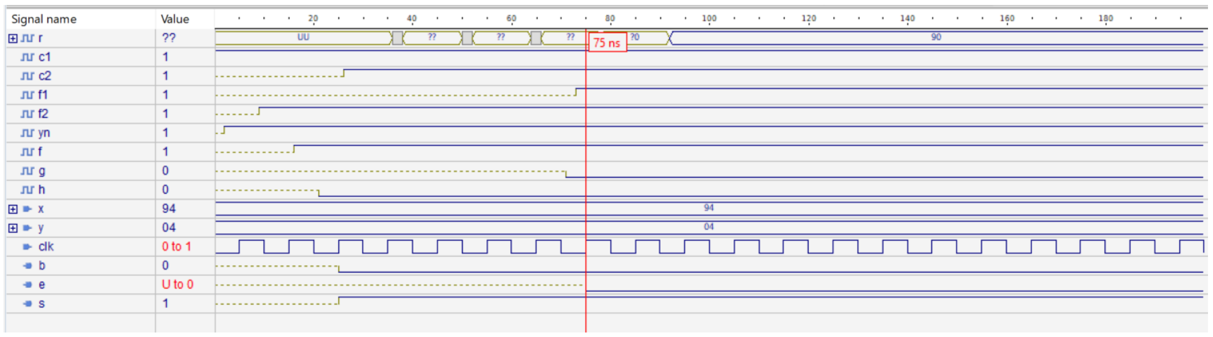


Figure 7:Stage 1 Simulation try (2).

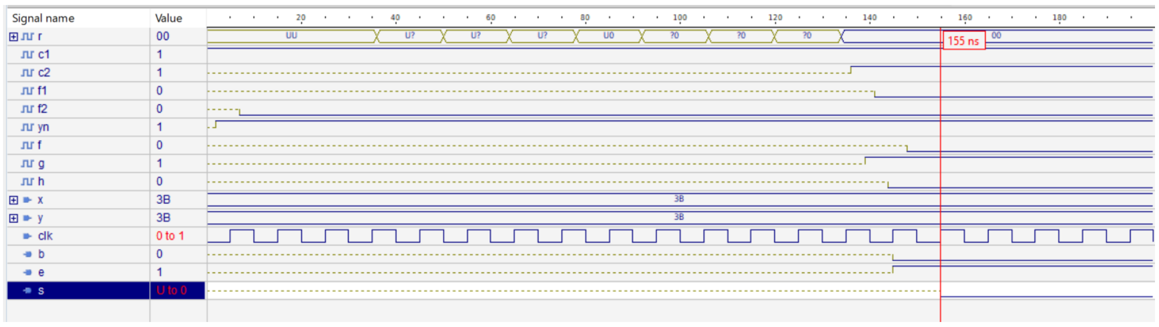
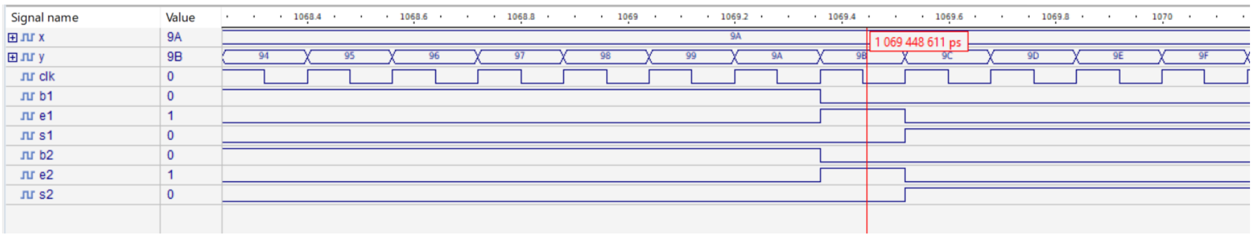


Figure 8: Stage 1 Simulation try (3).



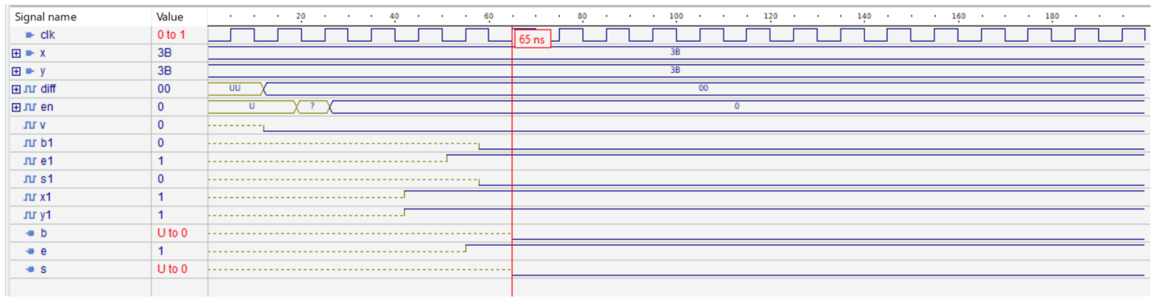
Signal name	Value
x	9A
y	9B
clk	0
b1	0
e1	1
s1	0
b2	0
e2	1
s2	0

Figure 9: Stage 1 Simulation Results.

Notice how the results in Theoretically generated values (b1, e1, s1) are equal to the values generated by the implemented circuit (b2, e2, s2). However, the both sets of values are always late by one clock cycle. Other than that, the results are accurate.

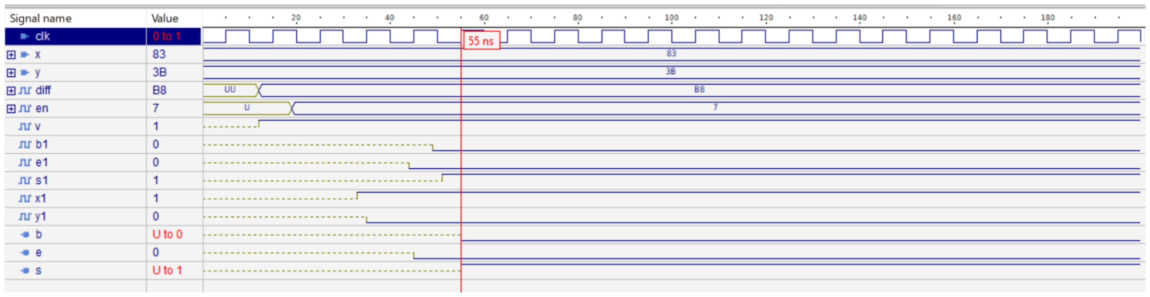
- Stage 2:

Stage two has a smaller latency compared to Stage 1. So, after several test values, the maximum latency was approximately 65ns. The clock cycle of the system was set to 70ns to elmenate all possible errors.



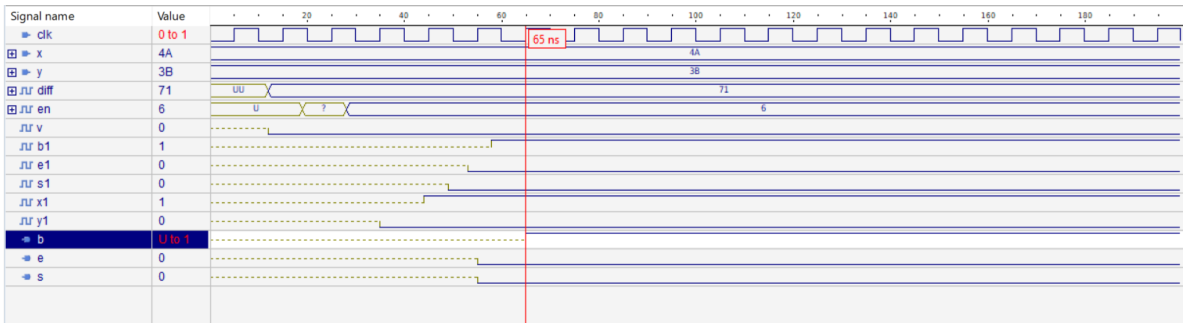
Signal name	Value
clk	0 to 1
x	3B
y	3B
diff	00
en	0
v	0
b1	0
e1	1
s1	0
x1	1
y1	1
b	U to 0
e	1
s	U to 0

Figure 10: Stage 2 Simulation try (1).



Signal name	Value
clk	0 to 1
x	83
y	3B
diff	B8
en	7
v	1
b1	0
e1	0
s1	1
x1	1
y1	0
b	U to 0
e	0
s	U to 1

Figure 11: Stage 2 Simulation try (2).



Signal name	Value
clk	0 to 1
x	4A
y	3B
diff	71
en	6
v	0
b1	1
e1	0
s1	0
x1	1
y1	0
b	U to 1
e	0
s	0

Figure 12: Stage 2 Simulation try (3).

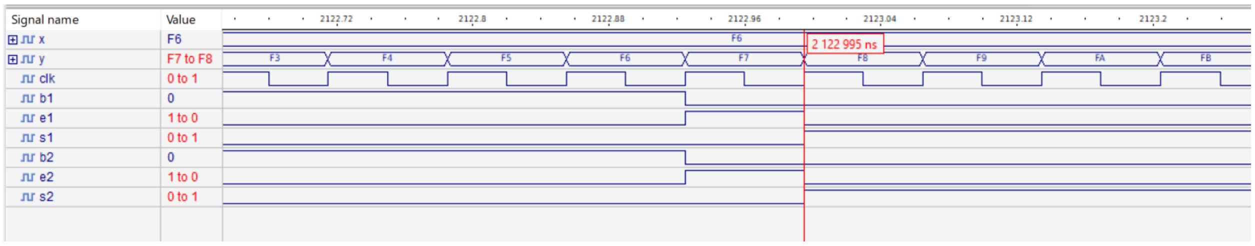


Figure 13: Stage 2 Simulation Results.

Just like phase 1, the results are from both the generator and the implemented circuit are delayed by 1 full clock cycle. Other than that, all results are similar and there were no errors reported from the test analyzer. The results are accurate.

• References:

- Storr, W. (2021, January 27). Priority Encoder and Digital Encoder Tutorial. Basic Electronics Tutorials. https://www.electronics-tutorials.ws/combination/comb_4.html

- Hussaini, U. (2020, September 2). Comparator – Designing 1-bit, 2-bit and 4-bit comparators using logic gates. Technobyte. https://technobyte.org/2-bit-4-bit-comparator/

- Zwolinski, M. (2000). Digital System Design and VHDL (1st ed.). Prentice Hall.

- Handouts and slides provide by the instructor.

• Appendix:

This appendix will contain all the code for this project.

• Gates:

```
--The inverter--

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity not1 is

	port(x: in std_logic;y:out std_logic);

end;


architecture simple of not1 is

begin

	y<= not x after 2ns;

end;

--And Gates (with different number of ports)--

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity and2 is

	port(x0,x1: in std_logic;y:out std_logic);

end;


architecture simple of and2 is

begin

	y<= x0 and x1 after 7ns;

end;


LIBRARY ieee;

USE ieee.std_logic_1164.ALL;
```

```
entity and3 is
	port(x0,x1,x2: in std_logic;y:out std_logic);
end;

architecture simple of and3 is
begin
	y<= x0 and x1 and x2 after 7ns;
end;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity and4 is
	port(x0,x1,x2,x3: in std_logic;y:out std_logic);
end;

architecture simple of and4 is
begin
	y<= x0 and x1 and x2 and x3 after 7ns;
end;

--or gates--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity or2 is
	port(x0,x1: in std_logic;y:out std_logic);
end;
```

```
architecture simple of or2 is
begin
	y<= x0 or x1 after 7ns;
end;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity or4 is
	port(x0,x1,x2,x3: in std_logic;y:out std_logic);
end;

architecture simple of or4 is
begin
	y<= x0 or x1 or x2 or x3 after 7ns;
end;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity or8 is
	port(x: in std_logic_vector(7 downto 0);y:out std_logic);
end;

architecture simple of or8 is
begin
	y<= x(0) or x(1) or x(2) or x(3) or x(4) or x(5) or x(6) or x(7) after 7ns;
end;

--nand gate--
```

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity nand2 is

        port(x0,x1: in std_logic;y:out std_logic);

end;


architecture simple of nand2 is

begin

        y<= x0 nand x1 after 5ns;

end;


--nor gates--


LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity nor2 is

        port(x0,x1: in std_logic;y:out std_logic);

end;


architecture simple of nor2 is

begin

        y<= x0 nor x1 after 5ns;

end;


LIBRARY ieee;

USE ieee.std_logic_1164.ALL;
```

```
entity nor8 is
        port(x: in std_logic_vector(7 downto 0);y:out std_logic);
end;

architecture simple of nor8 is
signal n:std_logic;
begin
        n<= x(0) or x(1) or x(2) or x(3) or x(4) or x(5) or x(6) or x(7);
        y<= not n after 5ns;
end;

--xnor gates--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity xnor2 is
        port(x0,x1: in std_logic;y:out std_logic);
end;

architecture simple of xnor2 is
begin
        y<= x0 xnor x1 after 9ns;
end;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity xor2 is
        port(x0,x1: in std_logic;y:out std_logic);
```

```
end;

architecture simple of xor2 is
begin
            y<= x0 xor x1 after 12ns;
end;

--xor gate--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity xor3 is
            port(x0,x1,x2: in std_logic;y:out std_logic);
end;

architecture simple of xor3 is
begin
            y<= x0 xor x1 xor x2 after 12ns;
end;
```

- Components:

```
-- Half Adder--

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity halfAdder is

        port(x,y: in std_logic; s,c:out std_logic);

end;


architecture simple of halfAdder is

begin

        g1: entity work.xor2(simple) port map(x,y,s);

        g2: entity work.and2(simple) port map(x,y,c);

end;

-- Full Adder--

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity fullAdder is

        port(x,y,z: in std_logic; s,co:out std_logic);

end;


architecture simple of fullAdder is

signal a,b,c,d: std_logic;

begin

        g1: entity work.halfAdder(simple) port map(x,y,a,b);

        g2: entity work.halfAdder(simple) port map(a,z,s,d);

        g3: entity work.or2(simple) port map(b,d,co);

end;


--8-Bit Adder/Subtractor--
```

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_signed.ALL;


entity Adder8bits is

            port (a,b: in std_logic_vector(7 downto 0);cin:in std_logic;r: out std_logic_vector(7 downto 0);
cout: out std_logic);

end Adder8bits;


architecture simple of Adder8bits is

signal ripple,b_neg :std_logic_vector(7 downto 0);

begin

                        ripple(0)<=cin;

            gen1: FOR i IN 0 TO 6 GENERATE

                        BEGIN

                                    g1: entity work.xor2(simple) port map(cin,b(i),b_neg(i));

                                    g2:         entity work.fullAdder(simple) port
map(a(i),b_neg(i),ripple(i),r(i),ripple(i+1));

                        END GENERATE;

                        g3: entity work.xor2(simple) port map(cin,b(7),b_neg(7));

                        g4:         entity work.fullAdder(simple) port map(a(7),b_neg(7),ripple(7),r(7),cout);

end simple;

--Priority Encoder 8x3--

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;


entity Encoder8x3 is

            port (x: in std_logic_vector(7 downto 0); y: out std_logic_vector(2 downto 0);v: out std_logic);

end Encoder8x3;

-- x(7-0)--a,b,c,d,e,f,g,h
```

```
architecture simple of Encoder8x3 is

signal reg: std_logic_vector(9 downto 0);


-- reg9 -- 6'

-- reg8 -- 5'

-- reg7 -- 4'

-- reg6 -- 3'

-- reg5 -- 2'

-- reg4 -- 5'4'2

-- reg3 -- 5'4'3

-- reg2 -- 6'4'2'1

-- reg1 -- 6'4'3

-- reg0 -- 6'5


begin

        g1: entity work.or4(simple) port map(x(7),x(6),x(5),x(4),y(2));

        g2: entity work.not1(simple) port map(x(6),reg(9));

        g3: entity work.not1(simple) port map(x(5),reg(8));

        g4: entity work.not1(simple) port map(x(4),reg(7));

        g5: entity work.not1(simple) port map(x(3),reg(6));

        g6: entity work.not1(simple) port map(x(2),reg(5));

        g7: entity work.and3(simple) port map(reg(8),reg(7),x(2),reg(4));

        g8: entity work.and3(simple) port map(reg(8),reg(7),x(3),reg(3));

        g9: entity work.or4(simple) port map(reg(4),reg(3),x(6),x(7),y(1));

        g10: entity work.and4(simple) port map(reg(9),reg(7),reg(5),x(1),reg(2));

        g11: entity work.and3(simple) port map(reg(9),reg(7),x(3),reg(1));

        g12: entity work.and2(simple) port map(reg(9),x(5),reg(0));

        g13: entity work.or4(simple) port map(reg(2),reg(1),reg(0),x(7),y(0));

        v <= x(7);
```

```
end simple;


--Mux8x1--


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;


entity Mux8x1 is
        port (x: in std_logic_vector(7 downto 0); s: in std_logic_vector(2 downto 0);y: out std_logic);
end Mux8x1;


architecture simple of Mux8x1 is
signal s1: std_logic_vector(2 downto 0);
signal x1: std_logic_vector(7 downto 0);
begin
g0: entity work.not1(simple) port map(s(0),s1(0));
g1: entity work.not1(simple) port map(s(1),s1(1));
g2: entity work.not1(simple) port map(s(2),s1(2));
g3: entity work.and4(simple) port map(x(0),s1(0),s1(1),s1(2),x1(0));
g4: entity work.and4(simple) port map(x(1),s(0),s1(1),s1(2),x1(1));
g5: entity work.and4(simple) port map(x(2),s1(0),s(1),s1(2),x1(2));
g6: entity work.and4(simple) port map(x(3),s(0),s(1),s1(2),x1(3));
g7: entity work.and4(simple) port map(x(4),s1(0),s1(1),s(2),x1(4));
g8: entity work.and4(simple) port map(x(5),s(0),s1(1),s(2),x1(5));
g9: entity work.and4(simple) port map(x(6),s1(0),s(1),s(2),x1(6));
g10: entity work.and4(simple) port map(x(7),s(0),s(1),s(2),x1(7));
g11: entity work.or8(simple) port map(x1,y);


end simple;
```

```
--One Bit/ Sign Comparator--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity oneBitComp is
        port (x,y,sn: in std_logic;b,e,s: out std_logic);
end oneBitComp;

architecture simple of oneBitComp is
signal xn,yn,snn,c1,c2,c3,c4: std_logic;
begin
        g1: entity work.not1(simple) port map(x,xn);
        g2: entity work.not1(simple) port map(y,yn);
        g3: entity work.not1(simple) port map(sn,snn);
        g4: entity work.and3(simple) port map(snn,x,yn,c1);
        g5: entity work.and3(simple) port map(sn,xn,y,c2);
        g6: entity work.or2(simple) port map(c1,c2,b);
        g7: entity work.and3(simple) port map(snn,xn,y,c3);
        g8: entity work.and3(simple) port map(sn,x,yn,c4);
        g9: entity work.or2(simple) port map(c3,c4,s);
        g10: entity work.xnor2(simple) port map(x,y,e);

end simple;
-- D flip flop--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity DFF is
```

```
        port (D,clk:in std_logic; Q: out std_logic);
end DFF;

ARCHITECTURE simple OF DFF IS
BEGIN
        PROCESS (clk)
                BEGIN
                        IF ( rising_edge(clk) ) THEN
                                q <= d;
                        END IF;
        END PROCESS;
END ARCHITECTURE simple;

--8 bit register---

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity DFF8 is
        port (D:in std_logic_vector(7 downto 0); clk:in std_logic; Q: out std_logic_vector(7 downto 0));
end DFF8;

ARCHITECTURE simple OF DFF8 IS
BEGIN
        gen1: FOR i IN 0 TO 7 GENERATE
                BEGIN
                        g1: entity work.DFF(simple) port map(D(i),clk,Q(i));
                END GENERATE;
END ARCHITECTURE simple;
```

- Stage 1 and 2:

```
--Main Entity--


LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_signed.ALL;


entity Comparator is

        port (x,y:in std_logic_vector(7 downto 0);clk:in std_logic;b,e,s:out std_logic);

end Comparator;


-- Stage 1 Archtucture --


architecture Stage1 of Comparator is

signal r: std_logic_vector(7 downto 0);

signal c1: std_logic := '1';

signal c2: std_logic;

signal f1,f2,yn,f,g,h: std_logic;

begin

        g1: entity work.Adder8bits(simple) port map(x,y,c1,r,c2);

        g2: entity work.and2(simple) port map(r(7),'1',f1);

        g3: entity work.not1(simple) port map(y(7),yn);

        g4: entity work.and2(simple) port map(x(7),yn,f2);

        g5: entity work.or2(simple) port map(f1,f2,f);

        g6: entity work.nor8(simple) port map(r,g);

        g7: entity work.nor2(simple) port map(f,g,h);

        reg0: entity work.DFF port map(h,clk,b);

        reg1: entity work.DFF port map(g,clk,e);

        reg2: entity work.DFF port map(f,clk,s);

end Stage1;
```

- ## Simulation Components:

```
--Test Generator--


LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_signed.ALL;

USE ieee.numeric_std.ALL;


entity TestGenerator is

        port(clk: in std_logic; x,y: out std_logic_vector(7 downto 0);b,e,s: out std_logic);

end TestGenerator;


architecture simple of TestGenerator is

signal x1,y1: std_logic_vector(7 downto 0);

signal b1,e1,s1: std_logic;

begin

        process

        begin

                WAIT UNTIL RISING_EDGE(CLK);

                FOR i IN -128 TO 127 LOOP

                        FOR j IN -128 TO 127 LOOP

                        x1 <= std_logic_vector( to_signed(i,8));

                        y1 <= std_logic_vector( to_signed(j,8));

                        WAIT UNTIL RISING_EDGE(CLK);

                        END LOOP;

                END LOOP;

                wait;

        end process;

        x<=x1;

        y<=y1;

        b1 <= '1' when x1 > y1 else '0';
```

```
		e1<= '1' when x1 = y1 else '0';
		s1<= '1' when x1 < y1 else '0';
		reg0: entity work.DFF port map(b1,clk,b);
		reg1: entity work.DFF port map(e1,clk,e);
		reg2: entity work.DFF port map(s1,clk,s);
end simple;

--Test Analyzer--

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity Analyzer is
		port (b1,e1,s1,b2,e2,s2,clk:in std_logic);
end Analyzer;

architecture simple of Analyzer is
begin
		 process
		begin
			assert b1 = b2
			report "Practical Resuls differ from theoraticly obtained observastions"
			severity ERROR;
			assert e1 = e2
			report "Practical Resuls differ from theoraticly obtained observastions"
			severity ERROR;
			assert s1 = s2
			report "Practical Resuls differ from theoraticly obtained observastions"
			severity ERROR;
			WAIT UNTIL rising_edge(CLK);
```

```
        end process;
end simple;
--Test Bench-- (just runs the Stages, The generator and the Analyzer)


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;


entity testb is
end;


architecture bb of testb is
signal x,y:        std_logic_vector(7 downto 0);
signal clk: std_logic := '0';
signal b1,e1,s1,b2,e2,s2:std_logic;
begin


        g0: entity work.TestGenerator(simple) port map (clk,x,y,b1,e1,s1);
        g1:     entity work.Comparator(Stage2) port map(x,y,clk,b2,e2,s2);
        g2: entity work.Analyzer(simple) port map (b1,e1,s1,b2,e2,s2,clk);


        clk <= not clk after 35ns;
end;
```