



# ASIA PACIFIC UNIVERSITY TECHNOLOGY & INNOVATION

---

EE057-4-2 VLSI DESIGN  
GUIDED LAB REPORT

---

TITLE	VLSI LAB REPORT
NAME & STUDENT ID	PEDRO FABIAN OWONO ONDO MANGUE(TP063251)
INTAKE	APD3F2308CE
LECTURER	DR SOON KIAN LUN
DATE	11/12/2023

## Table of Contents

<b>LIST OF FIGURES .....</b>	<b>3</b>
<b>OBJECTIVES .....</b>	<b>4</b>
<b>MODULE CODES.....</b>	<b>5</b>
<b>TESTBENCH FILES.....</b>	<b>9</b>
<b>RESULTS .....</b>	<b>14</b>
<b>DISCUSSION .....</b>	<b>21</b>
<b>CONCLUSION .....</b>	<b>23</b>
<b>REFERENCES.....</b>	<b>24</b>

## LIST OF FIGURES

Figure 1: Multiplexer and Demultiplexer .....	5
Figure 2: Full Adder Dataflow Modelling and Full Subtractor .....	6
Figure 3: SR and JK Flip Flops.....	7
Figure 4: Binary2GrayConversion.....	8
Figure 5: Full Adder with Process Statement .....	8
Figure 6: Testbenches: Multiplexer and Demultiplexer .....	9
Figure 7: Testbenches: Full Adder Dataflow Modelling and Full Subtractor .....	10
Figure 8: Testbench for Full Adder Behavioral Model .....	11
Figure 9: Testbenches: SR and JK Flip Flop .....	12
Figure 10: Testbench for Binary to Gray Converter .....	13
Figure 11: Waveform for Multiplexer .....	14
Figure 12: Waveform for Demultiplexer .....	15
Figure 13: Waveform for Full Adder Dataflow Model .....	16
Figure 14: Waveform for Full Adder Behavioural Model.....	17
Figure 15: Waveform for Full Subtractor .....	18
Figure 16: Waveform for JK FF .....	18
Figure 17: Waveform for SR FF .....	19
Figure 18: Waveform for 4-bit Binary to Gray.....	21

## **TABLES:**

Table 1: Multiplexer

Table 2: Demultiplexer

Table 3: Full Adder

Table 4: Full Subtractor

Table 5: JK Flip Flop

Table 6: SR Flip Flop

Table 7: 4-bit Binary to Gray

## **INTRODUCTION**

The world nowadays is becoming more digitalize in such a way that the binary digits, 0s and 1s is getting so interesting because many operations can be achieved through the manipulations of those binary digits. Therefore, in this digital world we can analyse certain Hardware Description Languages such as VHDL which is the one that I implemented in this assignment. VHDL language is a very powerful language which is related to circuits and systems and through it we can simulate or perform many tasks for example logic operations, data processing, etc. Besides that, I had the opportunity to use the VHDL language to perform tasks such as Multiplexer, Demultiplexers, 1-bit Full Adders, Subtractors, JK and SR Flip Flops, and 4-bit Binary to Gray converters.

To explore this lab report assignment, it is important to have a solid understanding and a theoretical background foundation. Multiplexers and demultiplexers are important components in digital circuits. Multiplexers allow us to choose from multiple inputs and take it toward a single output signal while in demultiplexers we have a single input that is directed toward multiple output signals.

Full adders and subtractors are involved in arithmetic operations. We can perform some operations such as addition and subtraction using binary numbers.

JK and SR flip-flops are related to memory and sequential logic because they both can store and control the states in digital systems. Besides that, they both have the capacity for storage and synchronization.

Finally, the 4-bit Binary to Gray code converter where we simply convert binary numbers to Gray which is important in digital communication.

## **OBJECTIVES**

1. Demonstrate the HDL codes for an 8X1 Multiplexer and 1X8 Demultiplexer and verify its functionality.
2. Demonstrate the HDL code to describe the functions of a full Adder and subtractor.
3. Demonstrate the HDL codes for SR and JK flip flops and verify their functionality.
4. Demonstrate the HDL code for the 4-bit Binary to Gray code converter.

## MODULE CODES

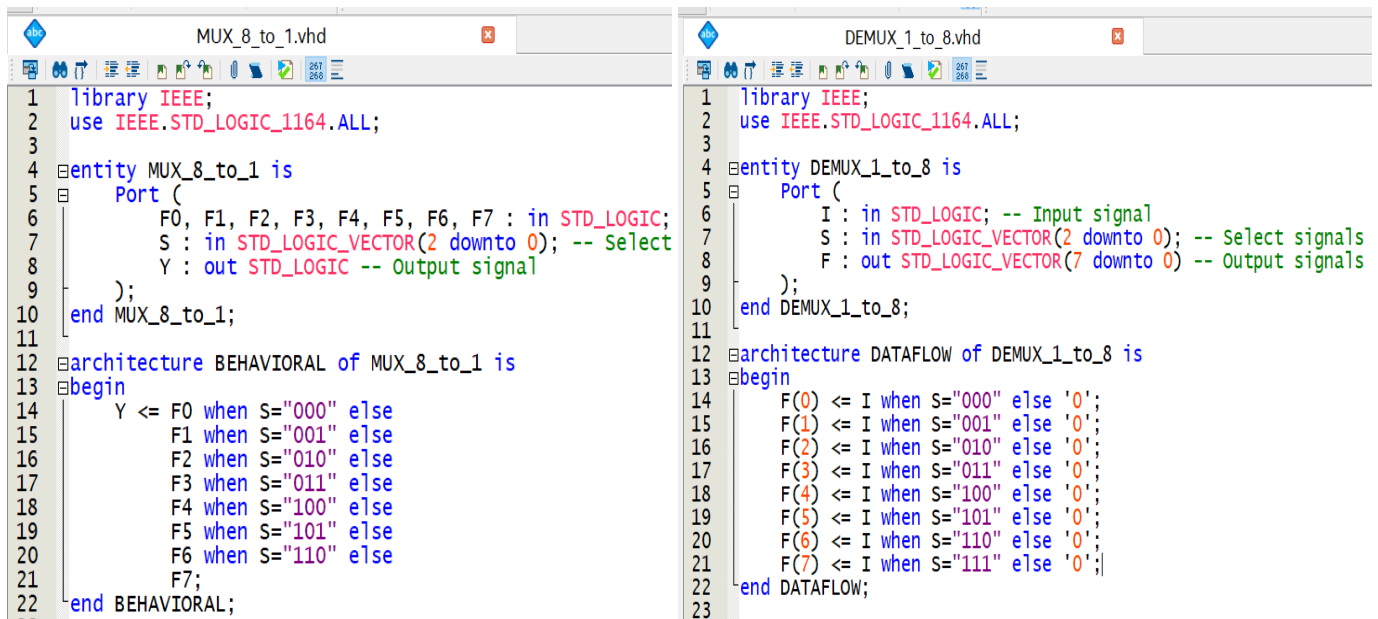


Figure 1: Multiplexer and Demultiplexer

### Module: Multiplexer 8-to-1

My VHDL entity name is MUX\_8\_to\_1.

- Eight input signals have been defined (F0 to F7).
- Three select signals are defined in the code (S from 0 to 2).
- One output signal named Y.
- I added a conditional assignment to direct the selected input to the output based on the value of S.
- I finally implemented a suitable testbench file where I used various combinations of input signals and select lines to prove that the output is working smoothly.

### Module: Demultiplexer 1-to-8

My VHDL entity name is DEMUX\_8\_to\_1.

- One input signal has been defined (I).
- Three select signals are defined in the code (S from 0 to 2).
- Eight-bit output signal named F.
- I created a suitable signal assignment that distributes the input signal to the correct output line based on S.

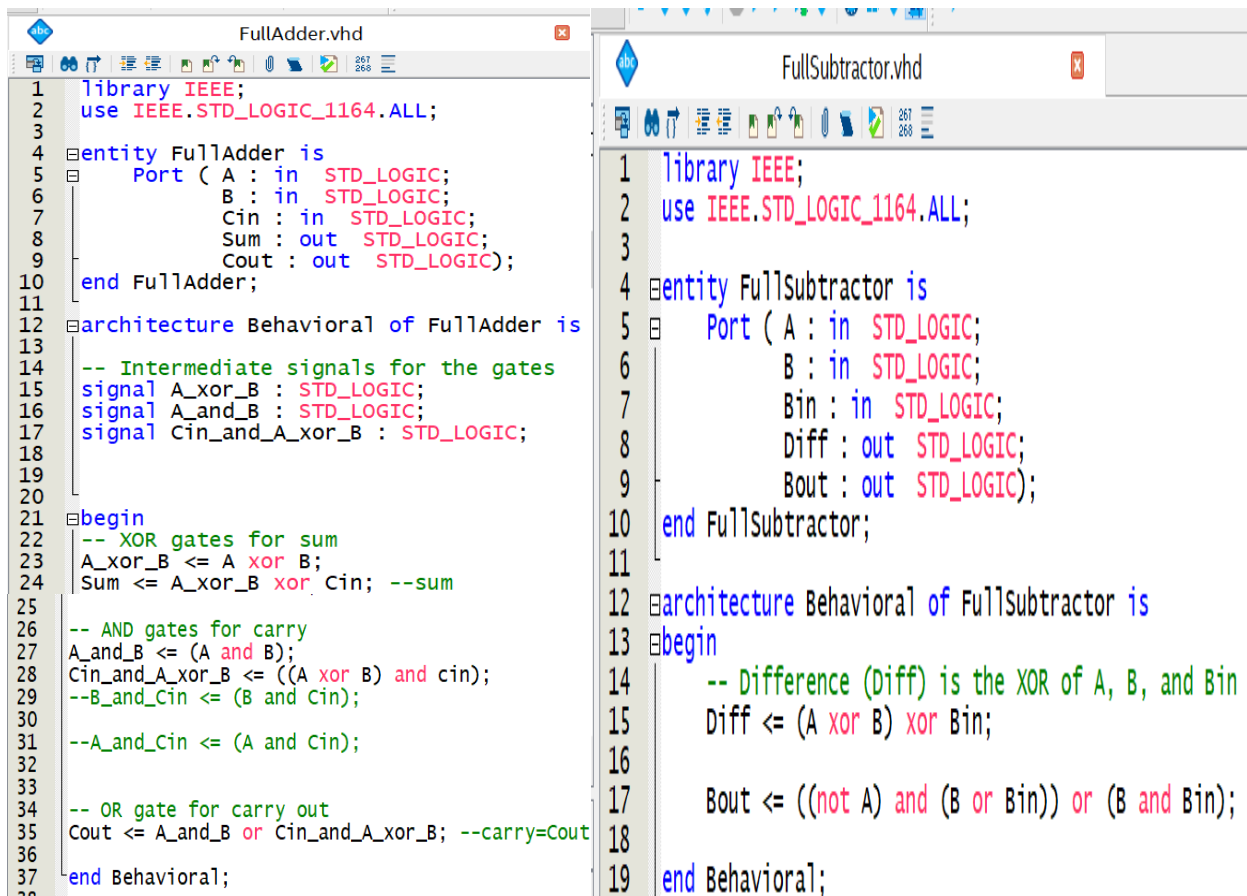


Figure 2: Full Adder Dataflow Modelling and Full Subtractor

Module: Full Adder

My VHDL entity name is Full Adder.

- Three inputs have been implemented in the module file which are A, B, Cin(Carry-in).
- Two outputs have been used which are Sum and Cout(Carry-out).
- XOR logic has been used for sum.
- AND-OR has been used for carry-out.

Module: Full Subtractor

My VHDL entity name is Full Subtractor.

- Three inputs have been implemented in the module file which are A, Bin (borrow-in).
- Two outputs have been used which are Diff and Bout (borrow-out).
- XOR logic has been used for difference and borrow out.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity SR_FF is
5   Port (
6     S : in STD_LOGIC;
7     R : in STD_LOGIC;
8     Q : out STD_LOGIC
9   );
10 end SR_FF;
11
12 architecture Behavioral of SR_FF is
13 begin
14   process(S, R)
15   begin
16     if S = '0' and R = '0' then
17       Q <= '0'; -- Reset to '0' v
18     elsif S = '0' and R = '1' then
19       Q <= '0'; -- Reset to '0' v
20     elsif S = '1' and R = '0' then
21       Q <= '1'; -- Set to '1' whe
22     elsif S = '1' and R = '1' then
23       Q <= 'X'; -- Undefined stat
24     end if;
25   end process;
26 end Behavioral;

```

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity JK_FF is
5   Port (
6     J : in STD_LOGIC;
7     K : in STD_LOGIC;
8     clk : in STD_LOGIC; -- Typically,
9     --JK flip-flops are edge-triggered
10    Q : out STD_LOGIC;
11    Qn : out STD_LOGIC -- Qn is
12    --the complement of Q
13  );
14 end JK_FF;
15
16 architecture Behavioral of JK_FF is
17   signal Q_int : STD_LOGIC := '0'; -- Internal
18   -- signal
19   -- to hold the state of Q
20 begin
21   process(clk)
22   begin
23     if rising_edge(clk) then
24       case STD_LOGIC_VECTOR'(J & K) is
25         when "00" => -- No change
26           Q_int <= Q_int;
27         when "01" => -- Reset
28           Q_int <= '0';
29         when "10" => -- Set
30           Q_int <= '1';
31         when others => -- Toggle
32           Q_int <= not Q_int;
33       end case;
34     end if;
35
36     Q <= Q_int;
37     Qn <= not Q_int;
38   end process;
39 end Behavioral;

```

Figure 3: SR and JK Flip Flops

## Module: SR Flip Flop

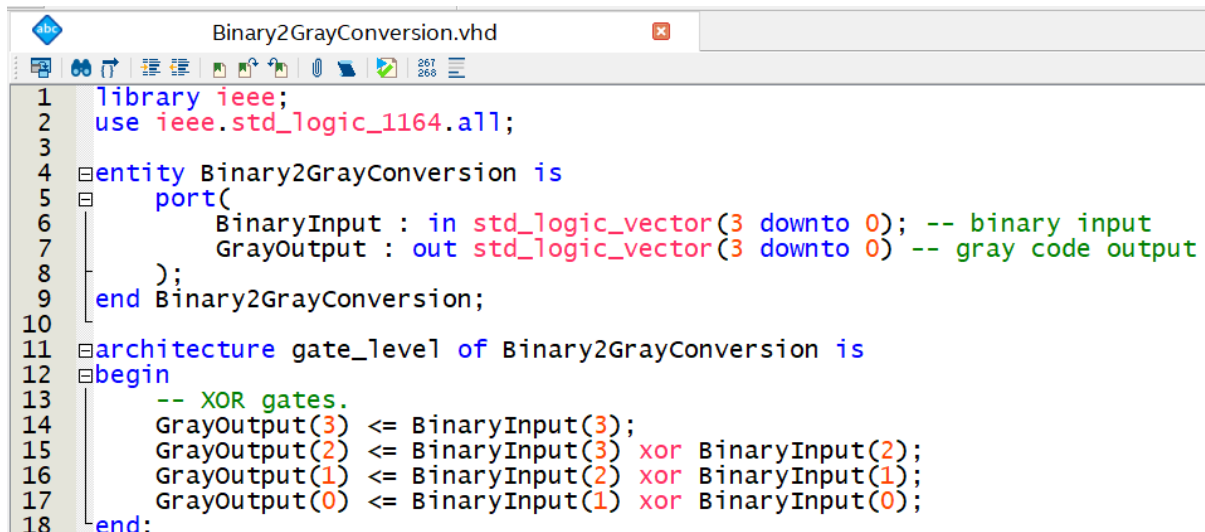
My VHDL entity name is SR\_FF.

- Two inputs have been implemented in the module file which are S(set), R(reset).
- One output has been used which is Q.

## Module: JK Flip Flop

My VHDL entity name is JK\_FF.

- Three inputs have been implemented in the module file which are J, K, clk (clock).
- I developed a process block where I set the clk (clock) as rising edge and I defined the conditions or behavior of Q and Qn(Complement of Q) based on the inputs such as J and K.



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Binary2GrayConversion is
5  port(
6      BinaryInput : in std_logic_vector(3 downto 0); -- binary input
7      GrayOutput  : out std_logic_vector(3 downto 0) -- gray code output
8  );
9  end Binary2GrayConversion;
10
11 architecture gate_level of Binary2GrayConversion is
12 begin
13     -- XOR gates.
14     GrayOutput(3) <= BinaryInput(3);
15     GrayOutput(2) <= BinaryInput(3) xor BinaryInput(2);
16     GrayOutput(1) <= BinaryInput(2) xor BinaryInput(1);
17     GrayOutput(0) <= BinaryInput(1) xor BinaryInput(0);
18 end;

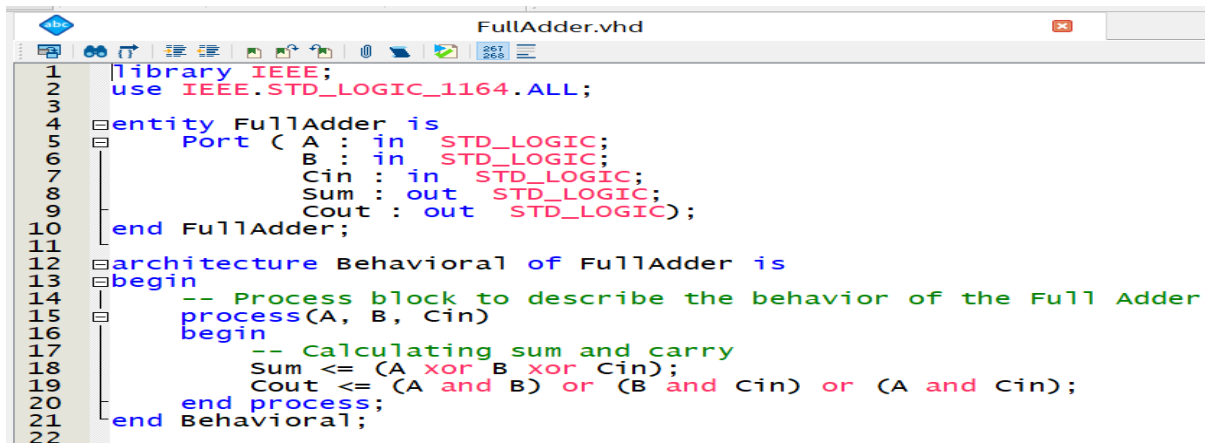
```

Figure 4: Binary2GrayConversion

Module: Binary2GrayConversion

My VHDL entity name is Binary2GrayConversion.

- One input has been implemented in the module file which is the 4-bit input (BinaryInput).
- One output has been used which is the 4-bit gray output (GrayOutput).
- XOR gates has been used to convert from each digit or bit of the binary to the correspondent gray code.



```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity FullAdder is
5  Port ( A : in STD_LOGIC;
6        B : in STD_LOGIC;
7        Cin : in STD_LOGIC;
8        Sum : out STD_LOGIC;
9        Cout : out STD_LOGIC);
10 end FullAdder;
11
12 architecture Behavioral of FullAdder is
13 begin
14     -- Process block to describe the behavior of the Full Adder
15     process(A, B, Cin)
16     begin
17         -- Calculating sum and carry
18         Sum <= (A xor B xor Cin);
19         Cout <= (A and B) or (B and Cin) or (A and Cin);
20     end process;
21 end Behavioral;
22

```

Figure 5: Full Adder with Process Statement

Module Code: Full Adder with process

My VHDL entity name is FullAdder.

- Three inputs have been implemented in the module file which are A, B, and Cin.
- Two outputs have been used which are Sum (represents the sum bit of the addition) and Cout (represents the carry-out bits).
- The process statement into the behavioral architecture is sensitive to changes to any input (A, B, Cin).
- We calculate the Sum inside the process through an XOR operation whereas for Cout we use a combination of OR and AND.



## TESTBENCH FILES

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity tb_MUX_8_to_1 is
5 end tb_MUX_8_to_1;
6 architecture BEHAVIOR of tb_MUX_8_to_1 is
7     signal F0, F1, F2, F3, F4, F5, F6, F7 : STD_LOGIC := '0';
8     signal S : STD_LOGIC_VECTOR(2 downto 0) := (others => '0');
9     signal Y : STD_LOGIC;
10    component MUX_8_to_1
11    port (
12        F0 : in STD_LOGIC;
13        F1 : in STD_LOGIC;
14        F2 : in STD_LOGIC;
15        F3 : in STD_LOGIC;
16        F4 : in STD_LOGIC;
17        F5 : in STD_LOGIC;
18        F6 : in STD_LOGIC;
19        F7 : in STD_LOGIC;
20        S : in STD_LOGIC_VECTOR(2 downto 0);
21        Y : out STD_LOGIC
22    );
23 end component;
24 begin
25     UUT: MUX_8_to_1 Port Map (
26         F0 => F0,
27         F1 => F1,
28         F2 => F2,
29         F3 => F3,
30         F4 => F4,
31         F5 => F5,
32         F6 => F6,
33         F7 => F7,
34         S => S,
35         Y => Y
36     );
37     process
38     begin
39         F0 <= '1';
40         F1 <= '0';
41         F2 <= '1';
42         F3 <= '0';
43         F4 <= '1';
44         F5 <= '0';
45         F6 <= '1';
46         F7 <= '0';
47         S <= "000"; wait for 10 ns;
48         S <= "001"; wait for 10 ns;
49         S <= "010"; wait for 10 ns;
50         S <= "011"; wait for 10 ns;
51         S <= "100"; wait for 10 ns;
52         S <= "101"; wait for 10 ns;
53         S <= "110"; wait for 10 ns;
54         S <= "111"; wait for 10 ns;
55         wait;
56     end process;
57 end BEHAVIOR;
```

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity tb_DEMUX_1_to_8 is
7 end tb_DEMUX_1_to_8;
8 architecture BEHAVIOR of tb_DEMUX_1_to_8 is
9     signal I : STD_LOGIC := '0';
10    signal S : STD_LOGIC_VECTOR(2 downto 0) := (others => '0');
11    signal F : STD_LOGIC_VECTOR(7 downto 0);
12    component DEMUX_1_to_8
13    port (
14        I : in STD_LOGIC;
15        S : in STD_LOGIC_VECTOR(2 downto 0);
16        F : out STD_LOGIC_VECTOR(7 downto 0)
17    );
18 end component;
19 begin
20     UUT: DEMUX_1_to_8 Port Map (
21         I => I,
22         S => S,
23         F => F
24     );
25     process
26     begin
27         I <= '1';
28         S <= "000";
29         wait for 10 ns;
30         I <= '1';
31         S <= "001";
32         wait for 10 ns;
33         I <= '1';
34         S <= "010";
35         wait for 10 ns;
36         I <= '1';
37         S <= "011";
38         wait for 10 ns;
39         I <= '1';
40         S <= "100";
41         wait for 10 ns;
42         I <= '1';
43         S <= "101";
44         wait for 10 ns;
45         I <= '1';
46         S <= "110";
47         wait for 10 ns;
48         I <= '1';
49         S <= "111";
50         wait for 10 ns;
51         wait;
52     end process;
53 end BEHAVIOR;
```

Figure 6: Testbenches: Multiplexer and Demultiplexer

### Testbench: 1\_to\_8 Demultiplexer.

- I used a single input which is “I” and 3-bit selected signal which is “S”;
- The output is an 8-bit vector which is “F”.
- My testbench for the demultiplexer goes through different selected values, S, to test the input, “I”, across the 8-bit output.

### Testbench: Multiplexer 1\_to\_8.

- Eight inputs have been implemented from F0 to F7, and 3-bit selected signal which is “S”;
- There is only one single output, “Y”.
- My testbench tests various input combinations and it goes through different selected signals.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity FullAdder_tb is
5 end FullAdder_tb;
6
7 architecture Behavioral of FullAdder_tb is
8 component FullAdder
9     Port ( A : in STD_LOGIC;
10           B : in STD_LOGIC;
11           Cin : in STD_LOGIC;
12           Sum : out STD_LOGIC;
13           Cout : out STD_LOGIC);
14 end component;
15 signal A : STD_LOGIC := '0';
16 signal B : STD_LOGIC := '0';
17 signal Cin : STD_LOGIC := '0';
18 signal Sum : STD_LOGIC;
19 signal Cout : STD_LOGIC;
20 begin
21     uut: FullAdder port map (A => A, B => B, Cin => Cin, Sum => Sum, Cout => Cout);
22     stim_proc: process
23     begin
24         wait for 100 ns;
25         A <= '0'; B <= '0'; Cin <= '0'; wait for 10 ns;
26         A <= '0'; B <= '0'; Cin <= '1'; wait for 10 ns;
27         A <= '0'; B <= '1'; Cin <= '0'; wait for 10 ns;
28         A <= '0'; B <= '1'; Cin <= '1'; wait for 10 ns;
29         A <= '1'; B <= '0'; Cin <= '0'; wait for 10 ns;
30         A <= '1'; B <= '0'; Cin <= '1'; wait for 10 ns;
31         A <= '1'; B <= '1'; Cin <= '0'; wait for 10 ns;
32         A <= '1'; B <= '1'; Cin <= '1'; wait for 10 ns;
33         wait for 100 ns;
34
35         wait;
36     end process;
37 end Behavioral;
38
39

```

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity FullSubtractor_tb is
5 end FullSubtractor_tb;
6
7 architecture Behavioral of FullSubtractor_tb is
8 component FullSubtractor
9     Port ( A : in STD_LOGIC;
10           B : in STD_LOGIC;
11           Bin : in STD_LOGIC;
12           Diff : out STD_LOGIC;
13           Bout : out STD_LOGIC);
14 end component;
15 signal A, B, Bin, Diff, Bout : STD_LOGIC;
16 begin
17     uut: FullSubtractor port map (A => A, B => B, Bin => Bin, Diff => Diff, Bout => Bout);
18     stim_proc: process
19     begin
20         A <= '0';
21         B <= '0';
22         Bin <= '0';
23         wait for 10 ns;
24         A <= '0';
25         B <= '0';
26         Bin <= '1';
27         wait for 10 ns;
28         A <= '0';
29         B <= '1';
30         Bin <= '0';
31         wait for 10 ns;
32         A <= '0';
33         B <= '1';
34         Bin <= '1';
35         wait for 10 ns;
36         A <= '1';
37         B <= '0';
38         Bin <= '0';
39         wait for 10 ns;
40         A <= '1';
41         B <= '0';
42         Bin <= '1';
43         wait for 10 ns;
44         A <= '1';
45         B <= '1';
46         Bin <= '0';
47         wait for 10 ns;
48         A <= '1';
49         B <= '1';
50         Bin <= '1';
51         wait for 10 ns;
52         wait;
53     end process;
54 end Behavioral;
55

```

Figure 7: Testbenches: Full Adder Dataflow Modelling and Full Subtractor

### Testbench: Full Adder

- Three inputs which are A, B, and Cin (carry-in).
- Two outputs that are Sum and Cout (carry-out).
- In the testbench we have all the possible combinations of the inputs to test the full adder functionality.

### Testbench: Full Subtractor

- Three inputs which are A, B, and Bin (borrow-in).
- Two outputs which are Diff (difference) and Bout (borrow-out).
- My testbench tests all the possible combinations of inputs to test the full subtractor functionality.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FullAdder_TB is
end FullAdder_TB;
architecture behavior of FullAdder_TB is
    component FullAdder
        Port ( A : in  STD_LOGIC;
              B : in  STD_LOGIC;
              Cin : in  STD_LOGIC;
              Sum : out STD_LOGIC;
              Cout : out STD_LOGIC);
    end component;
    signal A : STD_LOGIC := '0';
    signal B : STD_LOGIC := '0';
    signal Cin : STD_LOGIC := '0';
    signal Sum : STD_LOGIC;
    signal Cout : STD_LOGIC;
    constant clk_period : time := 10 ns;
begin
    uut: FullAdder Port Map (
        A => A,
        B => B,
        Cin => Cin,
        Sum => Sum,
        Cout => Cout
    );
    stim_proc: process
    begin
        wait for clk_period*10;
        A <= '0'; B <= '0'; Cin <= '0';
        wait for clk_period*10;
        A <= '0'; B <= '0'; Cin <= '1';
        wait for clk_period*10;
        A <= '0'; B <= '1'; Cin <= '0';
        wait for clk_period*10;
        A <= '0'; B <= '1'; Cin <= '1';
        wait for clk_period*10;
        A <= '1'; B <= '0'; Cin <= '0';
        wait for clk_period*10;
        A <= '1'; B <= '0'; Cin <= '1';
        wait for clk_period*10;
        A <= '1'; B <= '1'; Cin <= '0';
        wait for clk_period*10;
        A <= '1'; B <= '1'; Cin <= '1';
        wait for clk_period*10;
        wait;
    end process;
end behavior;

```

Figure 8: Testbench for Full Adder Behavioral Model

### Testbench: Full Adder

- Three inputs which are A, B, and Cin (carry-in).
- Two outputs that are Sum and Cout (carry-out).
- In the testbench we have all the possible combinations of the inputs to test the full adder functionality.
- The input signals are all initiated from zero (input signal).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity SR_FF_tb is
5 end SR_FF_tb;
6
7 architecture Behavioral of SR_FF_tb is
8     component SR_FF
9     port
10         S : in STD_LOGIC;
11         R : in STD_LOGIC;
12         Q : out STD_LOGIC
13     );
14 end component;
15 signal S, R : STD_LOGIC := '0';
16 signal Q : STD_LOGIC;
17 begin
18     uut: SR_FF port map (
19         S => S,
20         R => R,
21         Q => Q
22     );
23     stim_proc: process
24     begin
25         S <= '0'; R <= '0';
26         wait for 10 ns;
27         S <= '0'; R <= '1';
28         wait for 10 ns;
29         S <= '1'; R <= '0';
30         wait for 10 ns;
31         S <= '1'; R <= '1';
32         wait for 10 ns;
33         wait;
34     end process;
35 end Behavioral;

```

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity JK_FF_tb is
5 end JK_FF_tb;
6
7 architecture behavior of JK_FF_tb is
8     component JK_FF
9     port
10         J : in STD_LOGIC;
11         K : in STD_LOGIC;
12         clk : in STD_LOGIC;
13         Q : out STD_LOGIC;
14         Qn : out STD_LOGIC
15     );
16 end component;
17
18 signal J : STD_LOGIC := '0';
19 signal K : STD_LOGIC := '0';
20 signal clk : STD_LOGIC := '0';
21 signal Q : STD_LOGIC;
22 signal Qn : STD_LOGIC;
23 begin
24     uut: JK_FF port map (
25         J => J,
26         K => K,
27         clk => clk,
28         Q => Q,
29         Qn => Qn
30     );
31     clk_process: process
32     begin
33         while true loop
34             clk <= '0';
35             wait for 10 ns;
36             clk <= '1';
37             wait for 10 ns;
38         end loop;
39     end process;
40
41     stim_proc: process
42     begin
43         wait for 5 ns; -- Adjust delay for proper alignment with clock
44         J <= '0'; K <= '0'; -- No change
45         wait for 20 ns; -- Period for J=0, K=0
46         J <= '1'; K <= '0'; -- Set
47         wait for 20 ns; -- Period for J=1, K=0
48         J <= '0'; K <= '1'; -- Reset
49         wait for 20 ns; -- Period for J=0, K=1
50         J <= '1'; K <= '1'; -- Toggle
51         wait for 40 ns; -- Period for J=1, K=1 including no change
52         -- Repeat the sequence or finish simulation
53         -- wait;
54         assert false report "End of simulation" severity failure;
55     end process;
56 end behavior;

```

Figure 9: Testbenches: SR and JK Flip Flop

### Testbench: SR Flip Flop

- Two inputs which are S and R.
- Only one output, which is Q.
- My testbench test all the condition of Set and Reset inputs to visualize the changes in the flip flop.

### Testbench: JK Flip Flop

- Three inputs which are J, R, and clk (clock signal).
- Two outputs, which are Q, and Qn(a complement of Q).
- My testbench test all the condition of J and RK inputs to visualize the changes in the flip flop which can be Set, Reset, No change, and Toggle.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Binary2GrayConversion_tb is
5  end Binary2GrayConversion_tb;
6  architecture behavior of Binary2GrayConversion_tb is
7      component Binary2GrayConversion is
8      port(
9          BinaryInput : in std_logic_vector(3 downto 0);
10         GrayOutput : out std_logic_vector(3 downto 0)
11     );
12     end component;
13     signal BinaryInput, GrayOutput: std_logic_vector(3 downto 0) := (others => '0');
14 begin
15     uut: Binary2GrayConversion port map (
16         BinaryInput => BinaryInput,
17         GrayOutput => GrayOutput
18     );
19     stim_proc: process
20     begin
21         BinaryInput <= "0000"; wait for 10 ns;
22         BinaryInput <= "0001"; wait for 10 ns;
23         BinaryInput <= "0010"; wait for 10 ns;
24         BinaryInput <= "0011"; wait for 10 ns;
25         BinaryInput <= "0100"; wait for 10 ns;
26         BinaryInput <= "0101"; wait for 10 ns;
27         BinaryInput <= "0110"; wait for 10 ns;
28         BinaryInput <= "0111"; wait for 10 ns;
29         BinaryInput <= "1000"; wait for 10 ns;
30         BinaryInput <= "1001"; wait for 10 ns;
31         BinaryInput <= "1010"; wait for 10 ns;
32         BinaryInput <= "1011"; wait for 10 ns;
33         BinaryInput <= "1100"; wait for 10 ns;
34         BinaryInput <= "1101"; wait for 10 ns;
35         BinaryInput <= "1110"; wait for 10 ns;
36         BinaryInput <= "1111"; wait for 10 ns;
37         wait;
38     end process;
39 end architecture behavior;
40

```

Figure 10: Testbench for Binary to Gray Converter

### Testbench: 4-bit Binary to Gray converter

- A single input which is 4-bit Binary Input.
- A single output which is 4-bit Gray code.
- My testbench tests the possible 4-bit binary values that correspond to each 4-bit Gray code.

## RESULTS

Table 2: Multiplexer

Inputs, F	Select, S	Output, Y
1	000	F0
0	001	F1
1	010	F2
0	011	F3
1	100	F4
0	101	F5
1	110	F6
0	111	F7

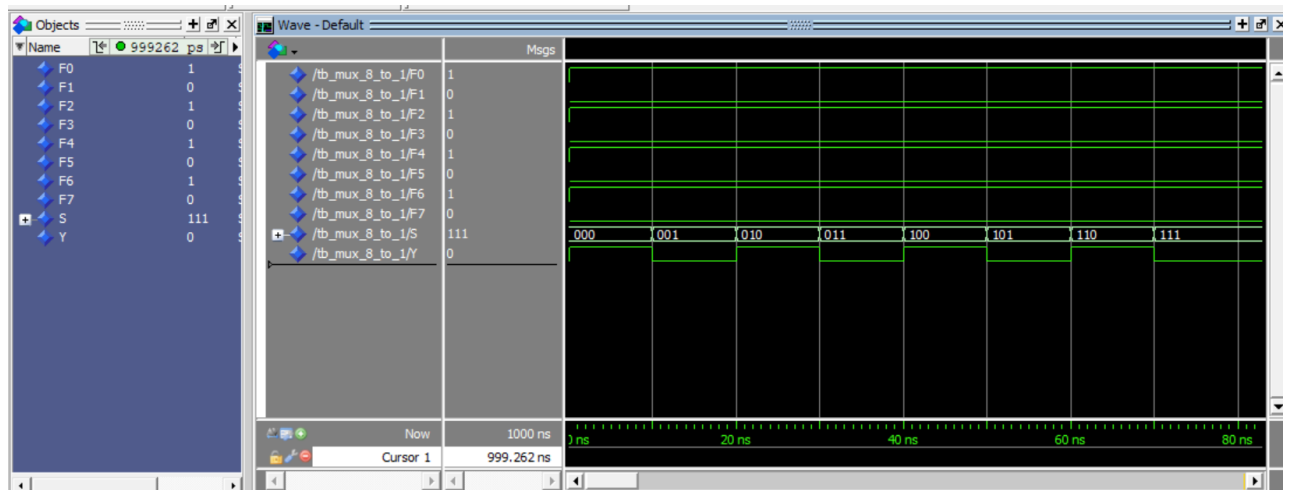


Figure 11: Waveform for Multiplexer

Table 2: Demultiplexer

I (input)	S (select)	F (output)
1	000	00000001
1	001	00000010
1	010	00000100
1	011	00001000
1	100	00010000
1	101	00100000
1	110	01000000
1	111	10000000

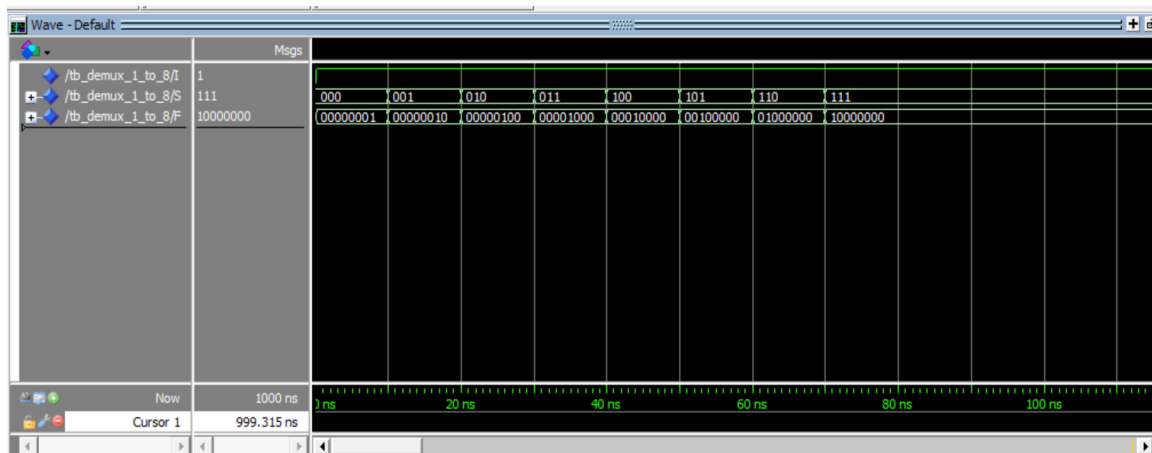


Figure 12: Waveform for Demultiplexer

Table 3: Full Adder (Same for both styles)

A	B	Cin	Sum	Cout
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

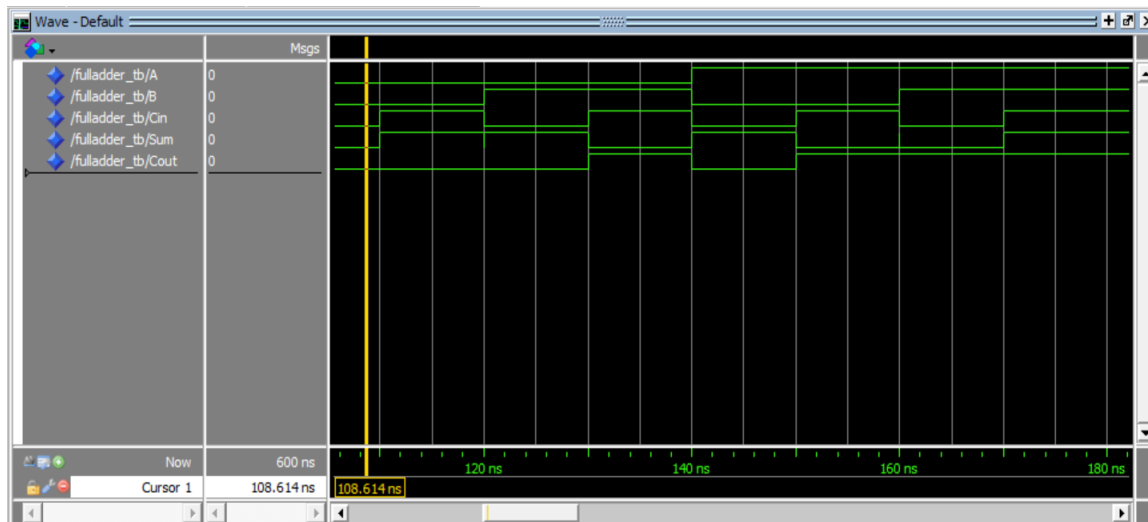


Figure 13: Waveform for Full Adder Dataflow Model



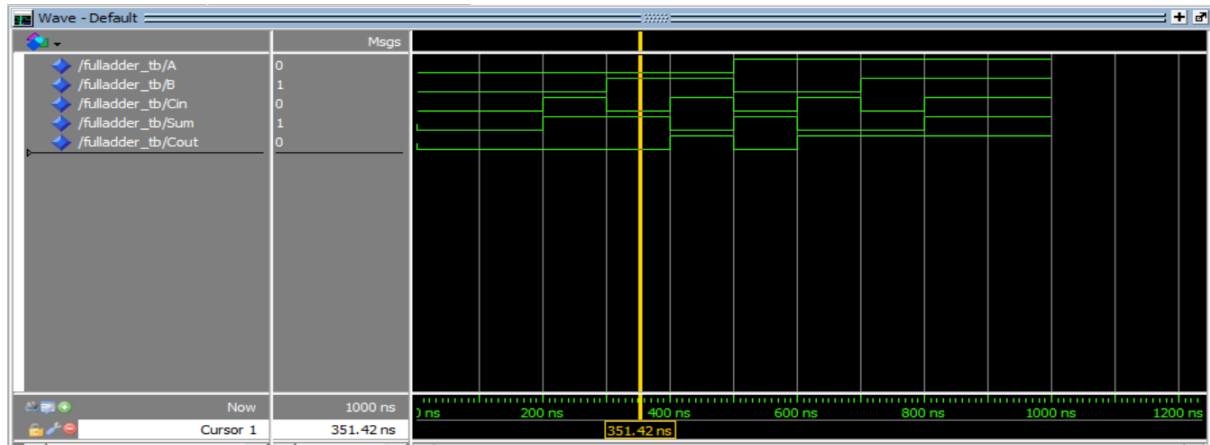


Figure 14: Waveform for Full Adder Behavioural Model

Table 4: Full Subtractor

A	B	Bin	Diff	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

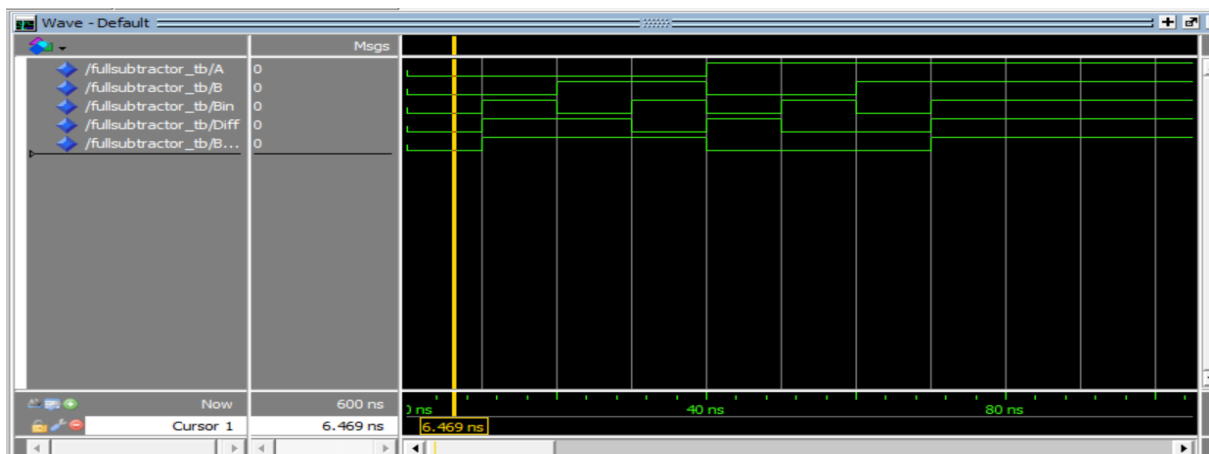


Figure 15: Waveform for Full Subtractor

Table 5: JK Flip Flop

J	K	clk	Q	Qn
0	0	High	0	1
0	1	High	0	1
1	0	High	1	0
1	1	High	1	0

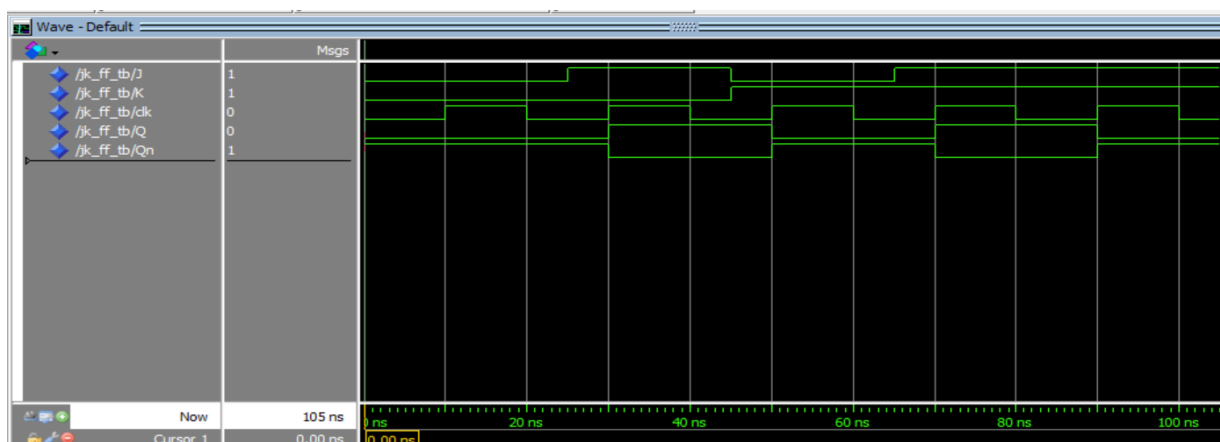


Figure 16: Waveform for JK FF

Table 6: SR Flip Flop

S	R	Q
0	0	0
0	1	0
1	0	1
1	1	X

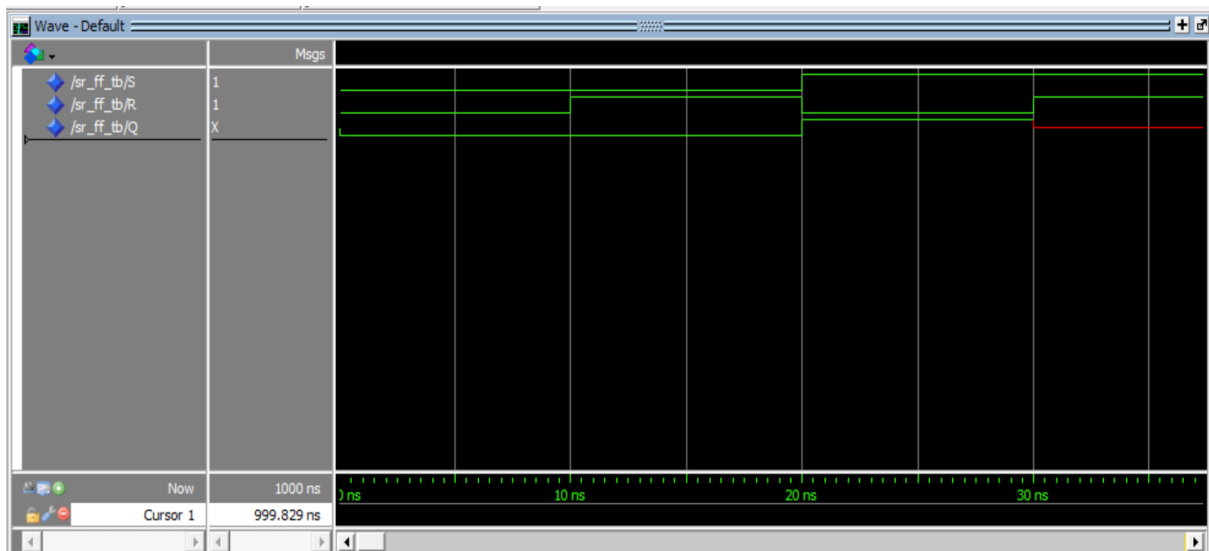


Figure 17: Waveform for SR FF

Table 7: 4-bit Binary to Gray

Binary	Gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

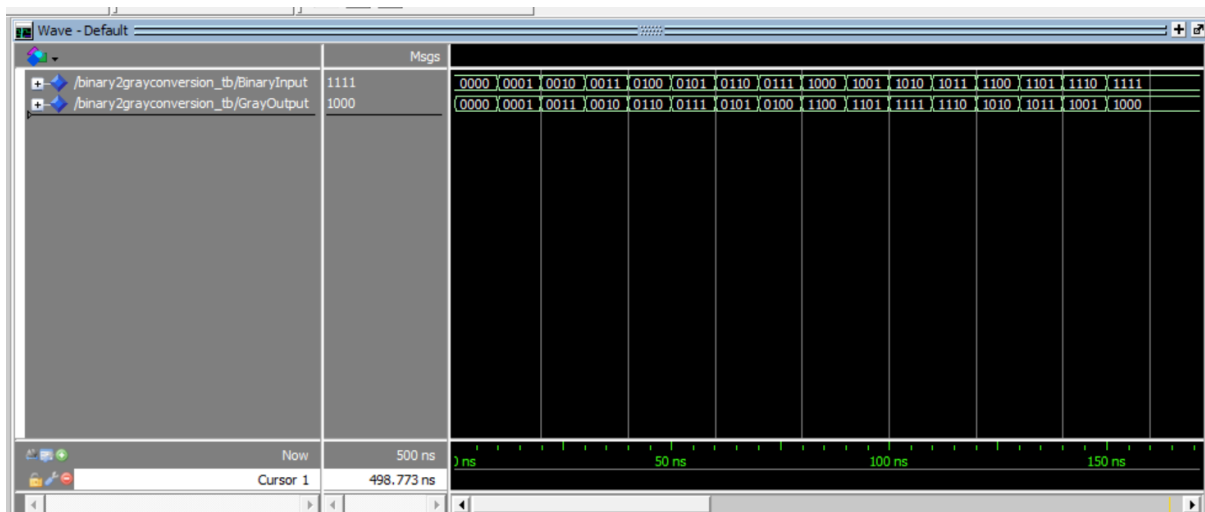


Figure 18: Waveform for 4-bit Binary to Gray

## DISCUSSION

I am going to conduct a discussion separately from each component based on the output obtained from the truth table and their respective waveforms:

### **MULTIPLEXER 8x1**

In the waveform for the multiplexer from waveform 1, we can appreciate an alignment on what a multiplexer should look like. Besides that, that waveform indicates a proper selection of the inputs which means that it shows the selection of the input starting from the first one until the last one (F0 to F7) in which each one is assigned to a value that is going to direct the output. The select line S is the one that controls the input that is going to be connected to the output Y. Moreover, the observer behaviour in the waveform effectively matches the truth table where each combination of the select lines is linked to the input that is going to direct or take the output Y. As an example, when the select, S is assigned to 001, then the input F1 which is 0 is going to go to the output Y which is going to be zero as well. In the waveform, you can observe how the set of inputs is reflected from the active to low signal as we go through each select line.

### **DEMULTIPLEXER 1x8**

The behaviour of the demultiplexer is the exact opposite of the multiplexer because, in a demultiplexer as the one that is illustrated in Waveform 2, we consider or take only a single input that is going to be distributed to the output based on the selected line, S. In my project, I set the input as one which is being taken as an input and it is going to be directed towards the corresponding output based on the select value. The waveform validates the behaviour of the truth table where the more we get different select lines in binary, it will affect the location of the input which is going to be changing its position. As an example, if

the input I, which is 1 decides to target the select line 000, we will get 00000001 which means that we will have the LSB set as 1 while the other will remain zero as stated in the module code for demultiplexer in figure 1.

## **FULL ADDER**

In waveform 3, we can observe that it is an implementation of the binary values such as A, B, and Cin which represent the addition of the three variables A, B, and Cin. In the waveform, we can track the value of the variable inputs which are all set to zero according to the waveform's figure 3 since it is the beginning of the stage where all of them are zero. Let us analyze a case where we have the values of A=0, B=1, Cin=0. Basically, in this situation, to get the Sum and the Cout we must follow the next requirements:

0+1 is equal to 1, now when we make an addition of that 1 plus 0, we will get 1 in the sum while we won't have any carry out which is going to give us 0, so basically that example summarizes the behaviour of the full adder as it also matches with the truth values in the table which correspond to the waveform displayed accordingly. Every possible combination of the variable inputs corresponds to the sum and carry out. On the other hand, there is another style of full adder where I included a process statement that contains the flow on how the changes will be made according to the addition.

## **FULL SUBTRACTOR**

For waveform 4, we can see an illustration of what a full subtractor looks like where I implemented three variables such as A, B, and Bin. The waveform for the full subtractor starts from zero, and so all the variables. If we move the point in such a place where A and B are equal to zero while the Bin is set to 1. The Diff (difference), and the Bout (borrow out) will be 1 as well. This is because if we subtract zero (A) minus zero (0) is going to be zero, so if we have zero (B) minus 1 we will need to have a borrow value from Bout which is one and the difference is going to be 1.

We can observe that the truth table for full subtractor matches with the waveform 4 where we initially set all the variables as zero, and that can be proved and shown in the truth table. The waveform represents accurately the binary subtraction process indicating whether a borrow is required for each one of the inputs as I explained in the example mentioned earlier.

## **JK FLIP FLOP**

In JK flip-flop there are multiple conditions, but one of the most important ones is the toggle behaviour which can be reflected in the waveform provided in waveform Figure 5. Starting with the toggle behaviour when J and K are high, then we will get the present state, Q as low while the next state is going to be high when the clock is low. In another case, using the same condition when the clock is high as well as J and K the Q value is going to be high whereas the Qn will be low. This behaviour of the JK flip flop can be observed in the waveform which matches with the truth table of the JK flip flop.

## **SR FLIP FLOP**

We can identify the waveform for SR Flip Flop in waveform 6 which demonstrates the set and reset behaviour provided in the truth table for SR. I implemented only three conditions in the waveform as we have in the truth table. As an example, we can observe in the table the condition where the S is high and the R is low, the final value of Q is going to be set to high when the clock is not rising. On the other hand, when the S is low and the R is high, then the Q will be low when the clock is rising. Basically, you can see how the truth table is fulfilled based on the waveform defined for SR Flip Flop.

## **BINARY TO GRAY**

In the waveform for Binary to Gray, we can appreciate a direct conversion from Binary to Gray output. The behaviour behind that conversion is that, as the input from any binary is changing, we can observe how the output of Gray values is being adjusted or aligned based on the binary input which is due to the implementation of the XOR gate that has been used in the process of conversion from binary to Gray. In the waveform, we have a proper validation of the correct implementation of 4-bit binary input to its corresponding Gray equivalent output.

## **CONCLUSION**

I successfully demonstrated the application of all components such as full adder, full subtractor, JK, and SR flip flop, as well as the 4-bit binary to Gray conversion by using VHDL code simulation. Besides that, I was able to get suitable waveforms for each of them. So, I explored and analyzed the validation and accuracy of those waveforms, being able to compare them with the truth table for every component. The truth tables and the waveforms are matched which is an achievement of completing the application of this lab assignment.

Finally, I was able to fully understand the process behind all the components as well as gain some knowledge in the use of the software Quartus Prime where I implemented the lab project. The VHDL language is a powerful tool that helped me in designing the components. I gained a lot of understanding of the techniques of using VHDL coding and I am looking forward to involving myself in future projects or labs where I can use VHD and be able to explore a lot more about designing and testing the logic of digital circuits.

## **REFERENCES**

Mano, M. m., & Ciletti, M. D. (2017). Digital design: With an introduction to the Verilog HDL. Pearson.

Mano, M. M., & Kime, C. R. (2019). Digital and Computer Design, Pearson.

Brown, S. D., & Vranesic, Z. G. (2013). Fundamentals of digital logic with Verilog design. McGraw-Hill Education.

Tocci, R. J., Widmer, N. S., & Moss, G. L. (2017). Digital systems: Principles and applications. Pearson

Tokheim, R. L. (2017). Digital electronics: Principles and applications. McGraw-Hill Educations.