ASIA PACIFIC UNIVERSITY OF TECHNOLOGY & INNOVATION

VLSI DESIGN

LAB REPORT

NAME & STUDENT ID	Su Xin Hong
INTAKE	APD3F2305CE
LECTURER	Ts. Dr. Kalaivani
HAND IN DATE	11th June 2023

Table of Contents

List Of Tables

List Of Figures

Introduction

This lab report recorded the lab activity of utilizing the Intel Quartus Prime software and Field-programmable gate arrays (FPGA) to create and simulate several circuit logics which includes 8X1 multiplexer, 1X8 demultiplexer, full adder, full subtractor, SR flip flop, JK flip flop and 4-bit binary to gray code logic. The objective of this lab activity is to demonstrate the functions of the digital logics mentioned above by utilizing HDL code.

Hardware Description Language (HDL) is a programming language that is specifically designed to represent or describes hardware. HDL can describe the characteristics, behaviour, nature, structure and timing of electronic circuits such as digital logic circuits (GeeksForGeeks, 2023). Hardware Description Language is also used to create or design a variety of digital circuits, motherboards, processors and so on. There are various types of HDL are there currently where one of the most popular one is known as Verilog HDL (VHDL). HDLs provides many functionalities, and ways for digital logic circuit designers to design, simulate, verify, analyse, optimize and so on to make the process of designing digital circuit easier.

Intel Quartus prime is a software designed by Intel Corporation. The software is designed to program the Field-programmable gate arrays (FPGA) and other devices such as System-on-a-Chip (SOC) and Complex Programmable Logic Devices (CPLD) (Intel, n.d.). However, in this lab report only FPGA will be used. Intel Quartus Prime also provides a complete set of tools to design, create, test and implement digital logics on the FPGA. There are many ways of designing tools provided in the software which includes block diagrams, schematics so does hardware description languages (HDL) such as Verilog HDL.

The schematic of the digital logics will be created using block diagram and all the waveforms will be generated by using the software. Then the logic is tested on FPGA. The HDL code is also generated using the software. All the block diagrams, waveforms, testing results and HDL codes are recorded in this lab report. The results were also discussed in detail in this report.

Objectives

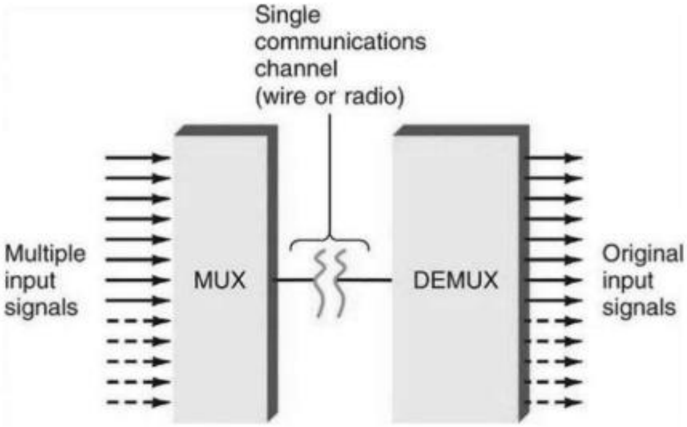
1. Demonstrate and compile the working principle of 8X1 Multiplexer and 1X8 Demultiplexer by utilizing HDL code.
2. Demonstrate and compile the working principle of Full Adder and Full Subtractor by utilizing HDL code.
3. Demonstrate and compile the working principle of SR and JK flip flops by utilizing HDL code.
4. Demonstrate and compile the working principle of 4-Bit Binary to Gray Code converter by utilizing HDL code.

Apparatus / Components

1. Software Intel Quartus Prime
2. FPGA

Multiplexer (8X1) & Demultiplexer (1X8)

Procedure Multiplexer & Demultiplexer



1. The 8X1 multiplexer and 1X8 demultiplexer circuit diagram were drawn using Quartus Prime block diagram.
2. The block diagram was compiled to check if there is any error.
3. The University Program VWF file is created.
4. The inputs were given.
5. The simulation was started to generate the waveforms.
6. The computer was connected to a FPGA.
7. All the inputs and outputs were assigned to pins in the FPGA.
8. The circuit is verified on the FPGA.
9. The HDL code was generated.

Schematic (Circuit Diagram) - Multiplexer

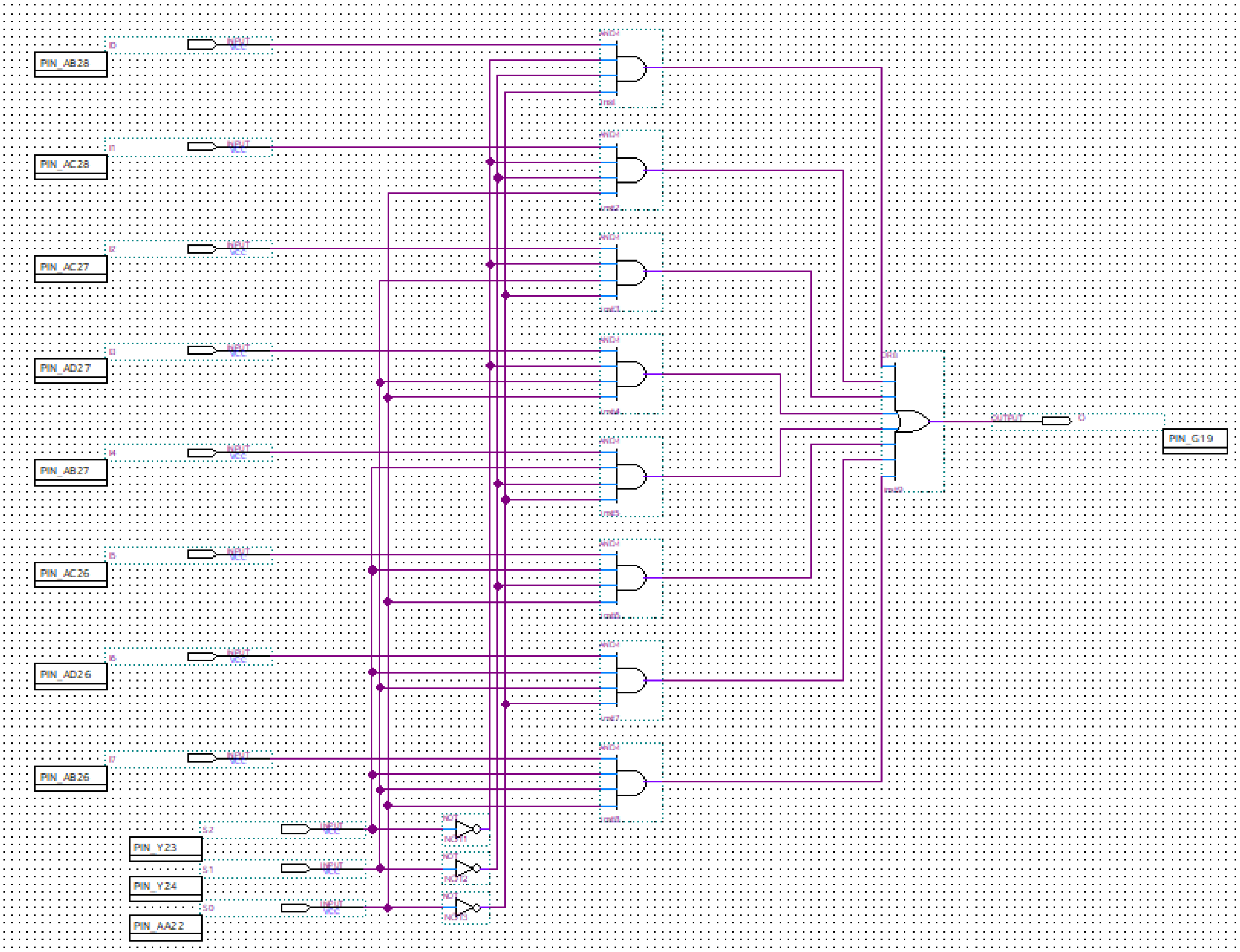
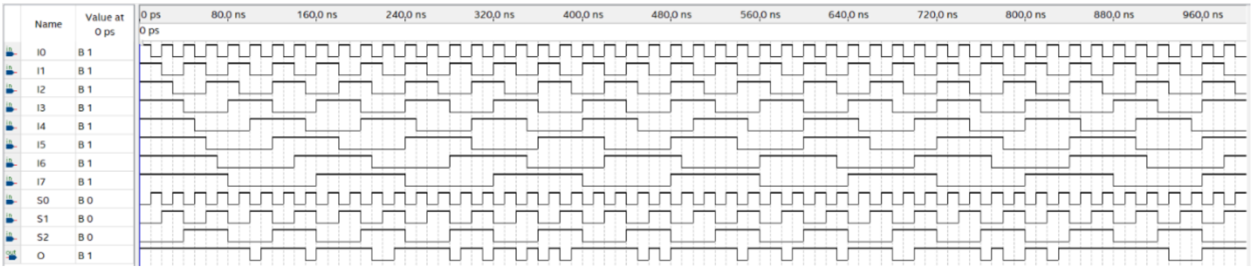


Figure 1: 8X1 Multiplexer Circuit Diagram

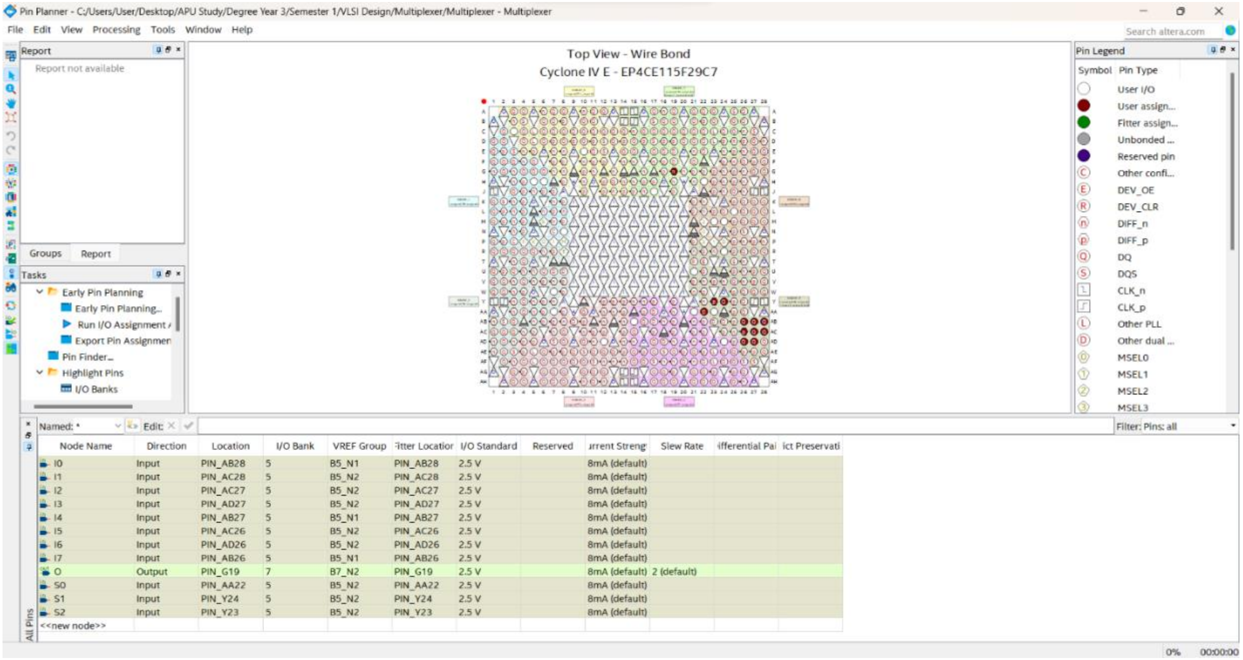
Output Signals (Waveform) - Multiplexer



Name	Value at 0 ps
I0	B 1
I1	B 1
I2	B 1
I3	B 1
I4	B 1
I5	B 1
I6	B 1
I7	B 1
S0	B 0
S1	B 0
S2	B 0
O	B 1

Figure 2: 8X1 Multiplexer Output Waveform

Pin Planner - Multiplexer



Pin Planner - C:/Users/User/Desktop/APU Study/Degree Year 3/Semester 1/VLSI Design/Multiplexer/Multiplexer - Multiplexer

Top View - Wire Bond
Cyclone IV E - EP4CE115F29C7

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
I0	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			
I1	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V		8mA (default)			
I2	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V		8mA (default)			
I3	Input	PIN_AD27	5	B5_N2	PIN_AD27	2.5 V		8mA (default)			
I4	Input	PIN_AB27	5	B5_N1	PIN_AB27	2.5 V		8mA (default)			
I5	Input	PIN_AC26	5	B5_N2	PIN_AC26	2.5 V		8mA (default)			
I6	Input	PIN_AD26	5	B5_N2	PIN_AD26	2.5 V		8mA (default)			
I7	Input	PIN_AB26	5	B5_N1	PIN_AB26	2.5 V		8mA (default)			
O	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
S0	Input	PIN_AA22	5	B5_N2	PIN_AA22	2.5 V		8mA (default)			
S1	Input	PIN_Y24	5	B5_N2	PIN_Y24	2.5 V		8mA (default)			
S2	Input	PIN_Y23	5	B5_N2	PIN_Y23	2.5 V		8mA (default)			
<<new node>>											

Figure 3: 8X1 Multiplexer Pin Planner

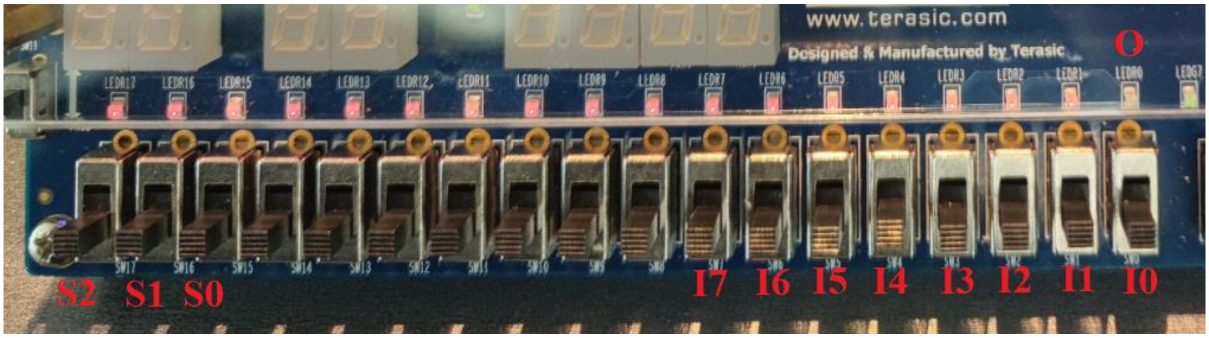


Figure 4: 8X1 Multiplexer FPGA Pins

Truth Table – Multiplexer

Table 1: 8X1 Multiplexer Truth Table

Inputs			Output
S2	S1	S0	O
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Results from FPGA Device - Multiplexer

Table 2: 8X1 Multiplexer FPGA Results

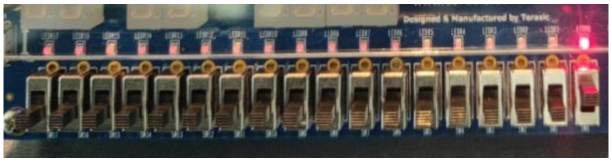
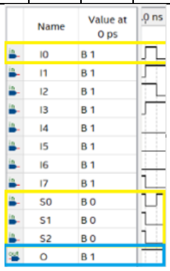


Figure 5: 8X1 Multiplexer FPGA Result 1

S2	S1	S0	I7	I6	I5	I4	I3	I2	I1	I0	O
0	0	0	0	0	0	0	0	0	0	1	1



Name	Value at 0 ps
I0	B 1
I1	B 1
I2	B 1
I3	B 1
I4	B 1
I5	B 1
I6	B 1
I7	B 1
S0	B 0
S1	B 0
S2	B 0
O	B 1

Figure 7: 8X1 Multiplexer FPGA Result Waveform 1

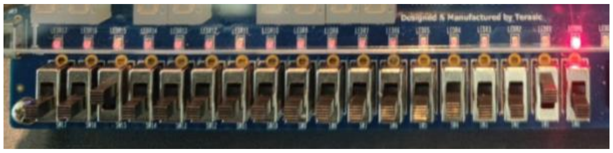
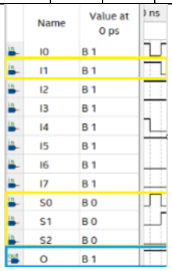


Figure 6: 8X1 Multiplexer FPGA Result 2

S2	S1	S0	I7	I6	I5	I4	I3	I2	I1	I0	O
0	0	1	0	0	0	0	0	0	1	0	1



Name	Value at 0 ps
I0	B 1
I1	B 1
I2	B 1
I3	B 1
I4	B 1
I5	B 1
I6	B 1
I7	B 1
S0	B 0
S1	B 0
S2	B 0
O	B 1

Figure 8: 8X1 Multiplexer FPGA Result Waveform 2

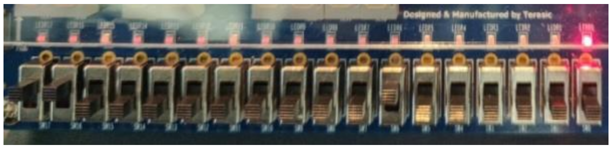
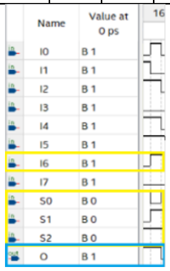


Figure 9: 8X1 Multiplexer FPGA Result 3

S2	S1	S0	I7	I6	I5	I4	I3	I2	I1	I0	O
1	1	0	0	1	0	0	0	0	0	0	1



Name	Value at 0 ps
I0	B 1
I1	B 1
I2	B 1
I3	B 1
I4	B 1
I5	B 1
I6	B 1
I7	B 1
S0	B 0
S1	B 0
S2	B 0
O	B 1

Figure 11: 8X1 Multiplexer FPGA Result Waveform 3

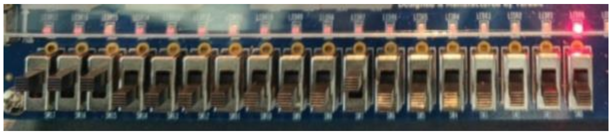
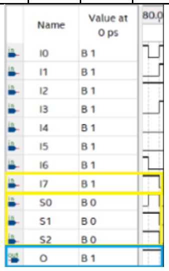


Figure 10: 8X1 Multiplexer FPGA Result 4

S2	S1	S0	I7	I6	I5	I4	I3	I2	I1	I0	O
1	1	1	1	0	0	0	0	0	0	0	1



Name	Value at 0 ps
I0	B 1
I1	B 1
I2	B 1
I3	B 1
I4	B 1
I5	B 1
I6	B 1
I7	B 1
S0	B 0
S1	B 0
S2	B 0
O	B 1

Figure 12: 8X1 Multiplexer FPGA Result Waveform 4

Figure 1 shows the circuit diagram with the logic circuit of an 8 to 1 multiplexer. The circuit diagram consists of 8 inputs for the input to the multiplexer (I0, I1, I2, I3, I4, I5, I6, I7), 3 inputs for the selector of the multiplexer (S0, S1, S2), 1 output of the multiplexer (O). The circuit diagram consists of 8 AND gates with 4 inputs, 3 NOT gates and one OR gate with 8 inputs. Figure 2 shows the waveform that is generated when the inputs were given. The input given into all the inputs were count inputs where the inputs count from 0 to 1 and then 0 repetitively but with different counting time. For I0 the counting time is 10ns, I1 is 20ns, I2 is 30ns, I3 is 40ns, I4 is 50ns, I5 is 60ns, I6 is 70ns and I7 is 80ns. While for the selector input the counting time for S0 is 10ns, S1 is 20ns and S3 is 40ns. The reason of setting the selector input as double of the counting time of the previous input is to make the selector as a 3-bit binary counter to make the waveform result easier to be understood.

Figure 3 shows the pin planner which displays the pins that are assigned to the inputs and outputs to the circuit diagram when it is connected to the FPGA. While Figure 4 shows the physical locations of the pins on the FPGA. The switch SW0 is the I0 input, SW1 is I1 input, SW2 is I2 input, SW3 is I3 input, SW4 is I4 input, SW5 is I5 input, SW6 is I6 input, SW7 is I7 input, SW15 is S0 selector input, SW16 is S1 selector input and lastly SW17 is S2 selector input. While the output O is at LEDR0.

The logic function of an 8 to 1 multiplexer is to get 8 inputs and choose which inputs as the output according to the input of the selectors (Electronics Tutorials, n.d.). Hence, only the input from the selected stream will affect the output which is shown in Table 1. As an example, if the input from the selector S2, S1 and S0 is 010, the third input to the multiplexer, I2 will be selected as the output.

Table 2 shows 4 examples of the results obtained while testing the multiplexer logic circuit on the FPGA with the corresponding section of the waveforms attached below it. However, the waveform attached does not 100% fulfil the conditions of the 8 inputs to the multiplexer on the FPGA but the selectors inputs were the same. Referring to the logic of a multiplexer, the inputs of the other stream that is not selected by the selector does not affect the output. Hence only the selected input is significant. The significant input was highlighted in Figure 7, Figure 8, Figure 11 and Figure 12 where all outputs was the input that is selected by the selector inputs. While Figure 5, Figure 6, Figure 9 and Figure 10 shows where the output LED is same as the input that is selected by the selector. This indicates the multiplexer circuit diagram is correct. As an example, at Figure 5 only the input I0 at SW0 is turned on which is high (1) while the input S0, S1 and S2 which corresponded to SW15, SW16 and SW17 were all not turned on which is low (0). This results in the output at LER0 which is O to light up where it obtained the output from the input I1 according to the selector inputs which is the binary 000 where the first input will be chosen as the output.

Schematic – Demultiplexer

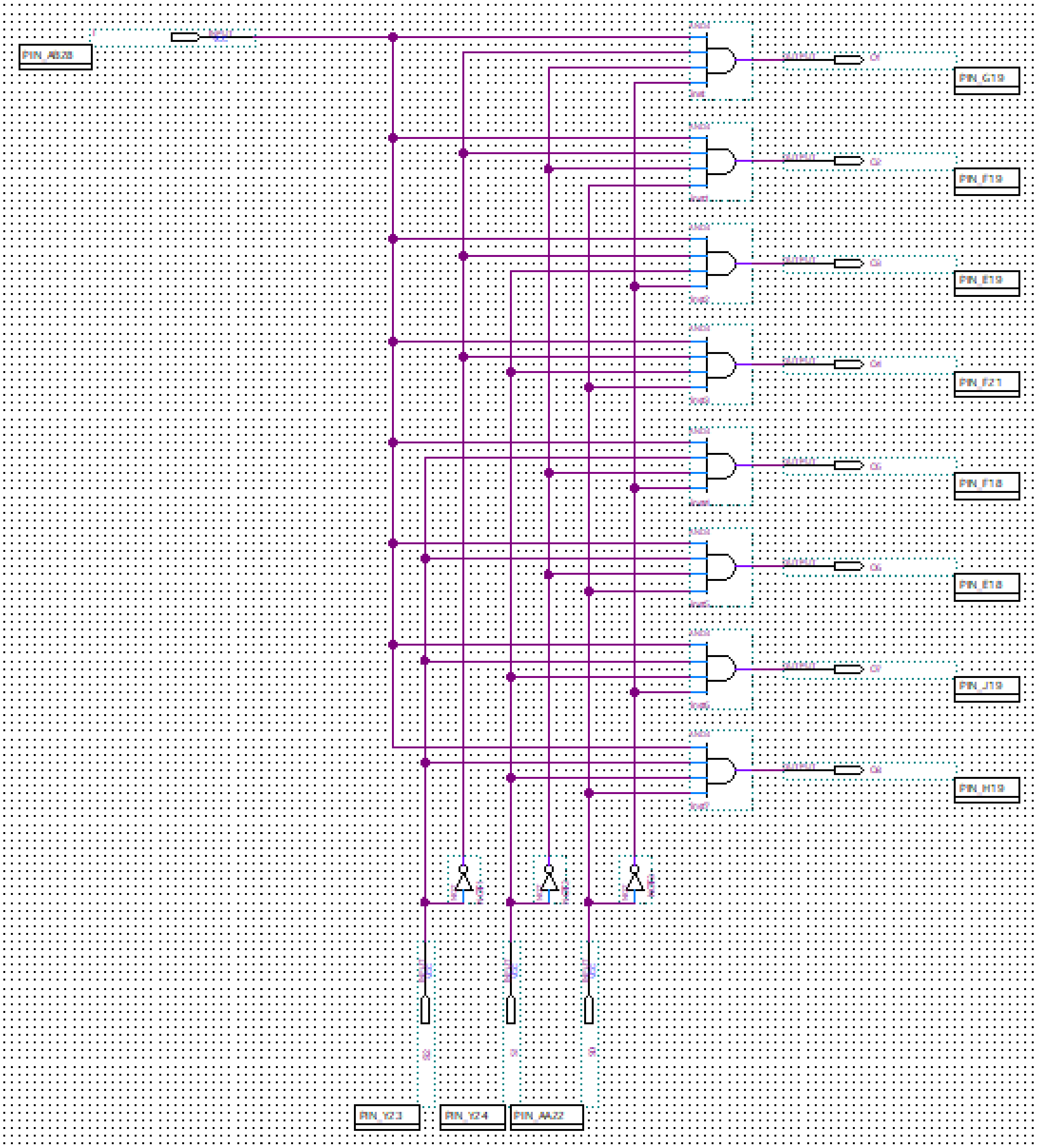
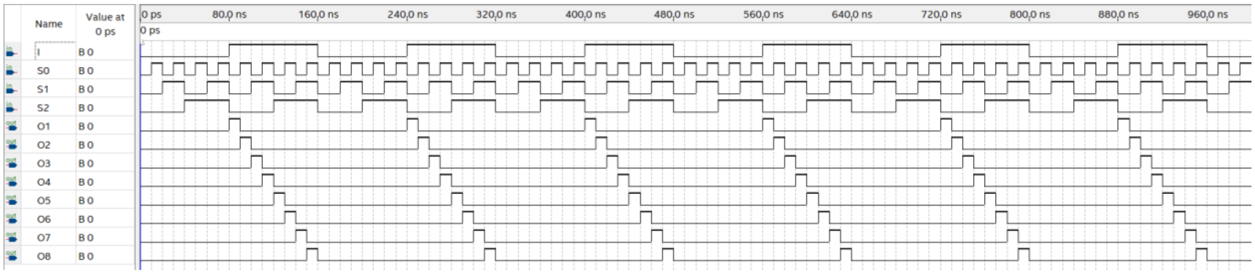


Figure 13: 1X8 Demultiplexer Circuit Diagram

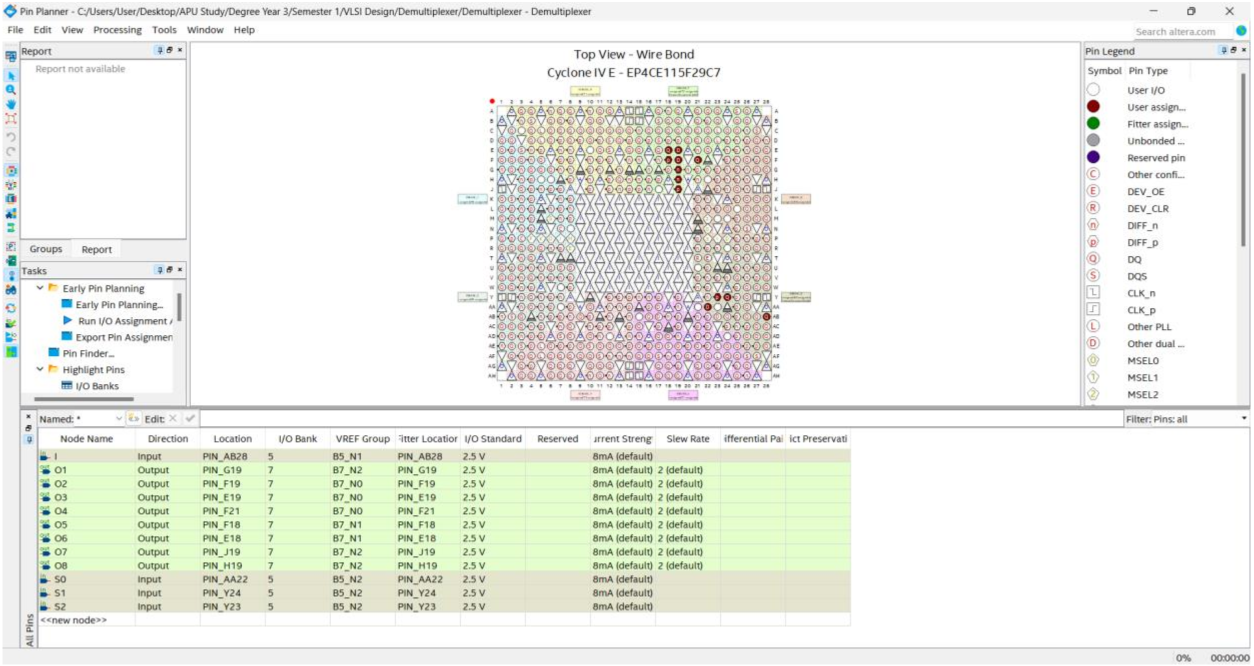
Output Signals (Waveform) – Demultiplexer



Name	Value at 0 ps
I	B 0
S0	B 0
S1	B 0
S2	B 0
O1	B 0
O2	B 0
O3	B 0
O4	B 0
O5	B 0
O6	B 0
O7	B 0
O8	B 0

Figure 14: 1X8 Demultiplexer Output Waveform

Pin Planner – Demultiplexer



Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	ırrent Streng	Slew Rate	ifferential Pai	ict Preservati
I	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			
O1	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
O2	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)		
O3	Output	PIN_E19	7	B7_N0	PIN_E19	2.5 V		8mA (default)	2 (default)		
O4	Output	PIN_F21	7	B7_N0	PIN_F21	2.5 V		8mA (default)	2 (default)		
O5	Output	PIN_F18	7	B7_N1	PIN_F18	2.5 V		8mA (default)	2 (default)		
O6	Output	PIN_E18	7	B7_N1	PIN_E18	2.5 V		8mA (default)	2 (default)		
O7	Output	PIN_J19	7	B7_N2	PIN_J19	2.5 V		8mA (default)	2 (default)		
O8	Output	PIN_H19	7	B7_N2	PIN_H19	2.5 V		8mA (default)	2 (default)		
S0	Input	PIN_AA22	5	B5_N2	PIN_AA22	2.5 V		8mA (default)			
S1	Input	PIN_Y24	5	B5_N2	PIN_Y24	2.5 V		8mA (default)			
S2	Input	PIN_Y23	5	B5_N2	PIN_Y23	2.5 V		8mA (default)			
<<new node>>											

Figure 15: 1X8 Demultiplexer Pin Planner

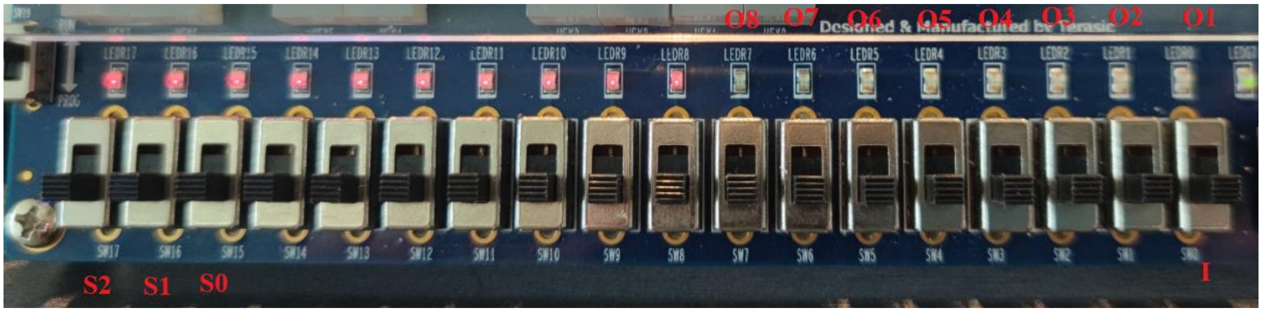


Figure 16: 1X8 Demultiplexer FPGA Pins

Truth Table – Demultiplexer

Table 3: 1X8 Demultiplexer Truth Table

Input				Output							
I	S2	S1	S0	O1	O2	O3	O4	O5	O6	O7	O8
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0

Results from FPGA Device – Demultiplexer

Table 4: 8X1 Demultiplexer FPGA Results

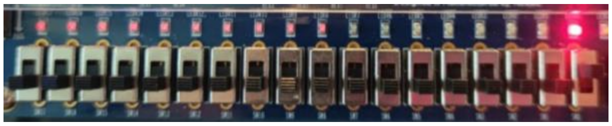


Figure 17: 8X1 Demultiplexer FPGA Result 1

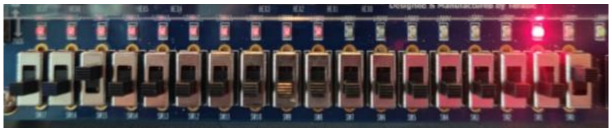
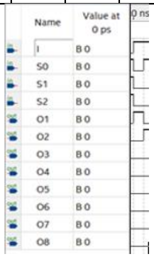


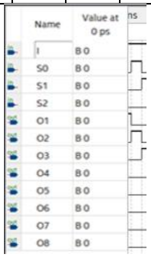
Figure 18: 8X1 Demultiplexer FPGA Result 2

S2	S1	S0	I	O8	O7	O6	O5	O4	O3	O2	O1	S2	S1	S0	I	O8	O7	O6	O5	O4	O3	O2	O1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0



Name	Value at 0 ps
I	B 0
S0	B 0
S1	B 0
S2	B 0
O1	B 0
O2	B 0
O3	B 0
O4	B 0
O5	B 0
O6	B 0
O7	B 0
O8	B 0

Figure 19: 8X1 Demultiplexer FPGA Result Waveform 1



Name	Value at 0 ps
I	B 0
S0	B 0
S1	B 0
S2	B 0
O1	B 0
O2	B 0
O3	B 0
O4	B 0
O5	B 0
O6	B 0
O7	B 0
O8	B 0

Figure 20: 8X1 Demultiplexer FPGA Result Waveform 2

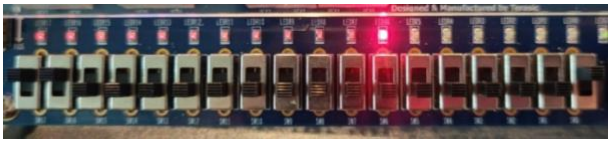


Figure 21: 8X1 Demultiplexer FPGA Result 3

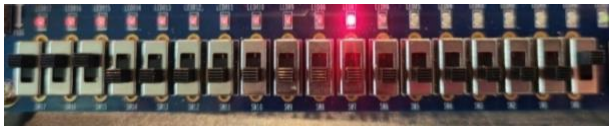
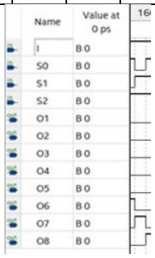


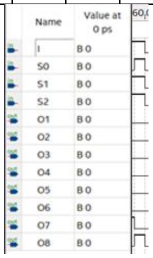
Figure 22: 8X1 Demultiplexer FPGA Result 4

S2	S1	S0	I	O8	O7	O6	O5	O4	O3	O2	O1	S2	S1	S0	I	O8	O7	O6	O5	O4	O3	O2	O1
1	1	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0



Name	Value at 0 ps
I	B 0
S0	B 0
S1	B 0
S2	B 0
O1	B 0
O2	B 0
O3	B 0
O4	B 0
O5	B 0
O6	B 0
O7	B 0
O8	B 0

Figure 23: 8X1 Demultiplexer FPGA Result Waveform 3



Name	Value at 0 ps
I	B 0
S0	B 0
S1	B 0
S2	B 0
O1	B 0
O2	B 0
O3	B 0
O4	B 0
O5	B 0
O6	B 0
O7	B 0
O8	B 0

Figure 24: 8X1 Demultiplexer FPGA Result Waveform 4

Figure 13 shows the circuit diagram with the logic circuit of a 1 to 8 demultiplexer. The circuit diagram consists of 1 input for the input to the demultiplexer (I), 3 inputs for the selector of the demultiplexer (S0, S1, S2), 8 output of the demultiplexer (O1, O2, O3, O4, O5, O6, O7, O8). The circuit diagram consists of 8 AND gates with 4 inputs, 3 NOT gates and one OR gate with 8 inputs. Figure 14 shows the waveform that is generated when the inputs were given. The input given into all the inputs were also count inputs which is similar with what is done in the multiplexer section. For I the counting time is 80ns so that it can test the condition for all combination of input from the selectors (S0, S1, S2). While for the selector input the counting time for S0 is 10ns, S1 is 20ns and S3 is 40ns which is same with what is done in the multiplexer to make it a 3-bit binary counter.

Figure 15 shows the pin planner which displays the pins that are assigned to the inputs and outputs of the demultiplexer circuit when it is connected to the FPGA. While Figure 16 shows the physical locations of the pins on the FPGA. The input switch SW0 is input I, SW15 is input S0, SW16 is input S1, SW17 is input S2. While the output O1 is LEDR0, O2 is LEDR1, O3 is LEDR2, O4 is LEDR3, O5 is LEDR4, O6 is LEDR5, O7 is LEDR6 and O8 is LEDR7.

The logic function of a 1 to 8 demultiplexer is to get 1 input and choose which output to pass the input to according to the input of the selectors (Electronics Tutorials, n.d.). Hence, only the output that is selected will get the input, I as the output which is shown in Table 3.

Table 4 shows 4 examples of the results obtained while testing the demultiplexer logic circuit on the FPGA with the corresponding section of the waveforms attached below it. Referring to the logic of a demultiplexer, the outputs of the other stream that is not selected by the selector will not get the input which causes them to be 0. While the selected output stream will have the output according to the input If the input (I) is 1 then the selected output will also be 1 while if the input (I) is 0 the output will also be 0. The output on the FPGA with different inputs in the selectors was shown in the table which are Figure 17, Figure 18, Figure 21 and Figure 22. The corresponds waveform that represents the conditions were also attached below which are Figure 19, Figure 20, Figure 23 and Figure 24. This indicates the circuit logic is correct where the output of the FPGA corresponds to the truth table and the waveform generated. As an example, at Figure 17 the input I at SW0 is turned on which is high (1) while the input S0, S1 and S2 which corresponded to SW15, SW16 and SW17 were all not turned on which is low (0). This results in the output at LER0 which is O1 to light up where the input from I1 is sent to the first output O1 located at LEDR0 according to the selector inputs which is the binary 000.

Discussion - Multiplexer (8X1) & Demultiplexer (1X8)

Multiplexer VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY Multiplexer IS
        PORT
        (
                S0 :  IN  STD_LOGIC;
                S1 :  IN  STD_LOGIC;
                S2 :  IN  STD_LOGIC;
                I7 :  IN  STD_LOGIC;
                I0 :  IN  STD_LOGIC;
                I1 :  IN  STD_LOGIC;
                I2 :  IN  STD_LOGIC;
                I3 :  IN  STD_LOGIC;
                I4 :  IN  STD_LOGIC;
                I5 :  IN  STD_LOGIC;
                I6 :  IN  STD_LOGIC;
                O :  OUT  STD_LOGIC
        );
END Multiplexer;

ARCHITECTURE bdf_type OF Multiplexer IS

SIGNAL  SYNTHESIZED_WIRE_20 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_21 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_22 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_12 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_13 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_14 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_15 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_16 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_17 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_18 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_19 :  STD_LOGIC;

BEGIN

SYNTHESIZED_WIRE_12 <= I0 AND SYNTHESIZED_WIRE_20 AND SYNTHESIZED_WIRE_21 AND
SYNTHESIZED_WIRE_22;
SYNTHESIZED_WIRE_14 <= I1 AND SYNTHESIZED_WIRE_20 AND SYNTHESIZED_WIRE_21 AND S0;
SYNTHESIZED_WIRE_13 <= I2 AND SYNTHESIZED_WIRE_20 AND S1 AND SYNTHESIZED_WIRE_22;
SYNTHESIZED_WIRE_15 <= I3 AND SYNTHESIZED_WIRE_20 AND S1 AND S0;
SYNTHESIZED_WIRE_17 <= I4 AND S2 AND SYNTHESIZED_WIRE_21 AND SYNTHESIZED_WIRE_22;
SYNTHESIZED_WIRE_16 <= I5 AND S2 AND SYNTHESIZED_WIRE_21 AND S0;
SYNTHESIZED_WIRE_18 <= I6 AND S2 AND S1 AND SYNTHESIZED_WIRE_22;
SYNTHESIZED_WIRE_19 <= I7 AND S2 AND S1 AND S0;

O <= SYNTHESIZED_WIRE_12 OR SYNTHESIZED_WIRE_13 OR SYNTHESIZED_WIRE_14 OR
SYNTHESIZED_WIRE_15 OR SYNTHESIZED_WIRE_16 OR SYNTHESIZED_WIRE_17 OR SYNTHESIZED_WIRE_18
OR SYNTHESIZED_WIRE_19;
SYNTHESIZED_WIRE_20 <= NOT(S2);
SYNTHESIZED_WIRE_21 <= NOT(S1);
SYNTHESIZED_WIRE_22 <= NOT(S0);

END bdf_type;
```

Demultiplexer VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY Demultiplexer IS
        PORT
        (
                I :  IN  STD_LOGIC;
                S0 :  IN  STD_LOGIC;
                S1 :  IN  STD_LOGIC;
                S2 :  IN  STD_LOGIC;
                O1 :  OUT  STD_LOGIC;
                O2 :  OUT  STD_LOGIC;
                O3 :  OUT  STD_LOGIC;
                O4 :  OUT  STD_LOGIC;
                O5 :  OUT  STD_LOGIC;
                O6 :  OUT  STD_LOGIC;
                O7 :  OUT  STD_LOGIC;
                O8 :  OUT  STD_LOGIC
        );
END Demultiplexer;

ARCHITECTURE bdf_type OF Demultiplexer IS

SIGNAL  SYNTHESIZED_WIRE_12 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_13 :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_14 :  STD_LOGIC;

BEGIN

O1 <= I AND SYNTHESIZED_WIRE_12 AND SYNTHESIZED_WIRE_13 AND SYNTHESIZED_WIRE_14;
O2 <= I AND SYNTHESIZED_WIRE_12 AND SYNTHESIZED_WIRE_13 AND S0;
O3 <= I AND SYNTHESIZED_WIRE_12 AND S1 AND SYNTHESIZED_WIRE_14;
O4 <= I AND SYNTHESIZED_WIRE_12 AND S1 AND S0;
O5 <= I AND S2 AND SYNTHESIZED_WIRE_13 AND SYNTHESIZED_WIRE_14;
O6 <= I AND S2 AND SYNTHESIZED_WIRE_13 AND S0;
O7 <= I AND S2 AND S1 AND SYNTHESIZED_WIRE_14;
O8 <= I AND S2 AND S1 AND S0;

SYNTHESIZED_WIRE_12 <= NOT(S2);
SYNTHESIZED_WIRE_13 <= NOT(S1);
SYNTHESIZED_WIRE_14 <= NOT(S0);

END bdf_type;
```

Section above shows the VHDL code of the multiplexer and demultiplexer circuit diagram that is generated by the Intel Quartus Prime software according to Figure 1 and Figure 13. First and foremost, for the multiplexer VHDL code, the code starts with the declaration of library which is “LIBRARY ieee;” and “USE ieee.std_logic_1164.all;”. The library is a collection of packages that are providing functions, datatypes, and components. In this case the code is importing the “std_logic_1164” package that is contained in the “ieee” library. The package is used to handle standard logic types and operations. For line “LIBRARY work;”, “work” is the default library where the entities and the architectures are placed.

While the “ENTITY Multiplexer” to “END Multiplexer” section is an entity declaration that specifies all the input and output ports in the circuit diagram or in the whole logic circuit. In the 8 to 1 multiplexer contains 8 inputs hence there are 8 lines of input declaration lines which are “S0: IN STD_LOGIC” and so on which includes all input data signal I0, I1, I2, I3, I4, I5, I6 and I7. Besides that, there are also 3 inputs for the selection control “S0: IN STD_LOGIC” which includes S0, S1 and S2. Lastly there is also a declaration of an output “O: OUT STD_LOGIC”, O.

Then the code proceeded to the architecture section where it starts with the line “ARCHITECTURE bdf_type OF Multiplexer” where it means that the architecture is named as “bdf_type” that corresponded to the “Multiplexer” entity. This is the section where it defines how the inputs and outputs of an entity are related and how the desired functionality is achieved. The architecture section will contain signal declaration section and behavioural section. The architecture specifies the internal workings of the entity and can include concurrent or sequential statements. Concurrent statement is where multiple tasks were executed at the same time or simultaneously with another parallelly, whereas sequential statement is where tasks only executed one after another linearly.

In the signal declaration section of the architecture section signal declaration was carried out. Take line “SIGNAL SYNTHESIZED_WIRE_20: STD_LOGIC;” as an example, in VHDL architecture “SIGNAL” is the keyword used to declare a signal. Signals are used to represent the internal connections between the entity, inputs and outputs to display the flow of signals or data in a design or a logic circuit. While “SYNTHESIZED_WIRE_20” is the name given to the signal which is declared as “STD_LOGIC” type. They were intermediate wire signals that will be used in the architecture. It can be understood as the signal or the data is stored in a wire to be processed or passed. Hence, by declaring it in VHDL code, a signal that can hold a logic value is created with the name of “SYNTHESIZED_WIRE_20”. It can be assigned with different values to be utilized in arithmetic or logical operations to fulfil the functionalities of the design which is an 8 to 1 multiplexer in this case.

Then the code went into the behaviour section where it is within the “BEGIN” and “END bdf_type;” lines. This line also indicated the end of the whole architecture section. Behaviour section is where all the codes regarding the process and logic such as signals assignments and so on are located. All the codes in this section for multiplexer are concurrent statements which means that all the codes are executed simultaneously or parallelly. As an example of signal assignment process “SYNTHESIZED_WIRE_12 <= I0 AND SYNTHESIZED_WIRE_20 AND SYNTHESIZED_WIRE_21 AND SYNTHESIZED_WIRE_22;” for this line. This line performed the AND logic operation with 4 signals which are “I0”, “SYNTHESIZED_WIRE_20”, “SYNTHESIZED_WIRE_21” and “SYNTHESIZED_WIRE_22”. And the results of the process are assigned to “SYNTHESIZED_WIRE_12”. This code corresponded to the logic of the multiplexer where all the 8 AND gates will have 4 inputs, 1 from the input to the multiplexer I0, I1, I2, I3, I4, I5, I6 or I7 while the other 3 is from the selector input S0, S1 or S2 or their inverse. In this case the input from the selectors were in the form of signals is because the selector inputs were also being given into other AND gates, hence the input is stored in the form of signal. While the input I0 is given directly into the AND gate only thus it is in its original form. “O <= SYNTHESIZED_WIRE_12 OR SYNTHESIZED_WIRE_13 OR SYNTHESIZED_WIRE_14 OR SYNTHESIZED_WIRE_15 OR SYNTHESIZED_WIRE_16 OR SYNTHESIZED_WIRE_17 OR SYNTHESIZED_WIRE_18 OR SYNTHESIZED_WIRE_19;” while for this line there is a total of 8 components which were the output of the 8 AND gates performing an OR operation and the result is assigned to the output “O”. “SYNTHESIZED_WIRE_12 <= NOT(S2);” this is also an example of code that is performing a NOT operation. All the codes in the behaviour section describes the behaviour of the multiplexer circuit created in Figure 1.

While for the demultiplexer HDL code, all the syntax are the same but only with different logics. In demultiplexer HDL code, the imported library and package is the same as what is imported in the multiplexer HDL code. This is the most common library that is used throughout the HDL code in this report. According to the demultiplexer HDL code entity declaration section there are a total of 4 inputs which includes the 3 selector inputs (S0, S1, S2) as an example “S0: IN STD_LOGIC;” and 1 input to the multiplexer (I), “I: IN STD_LOGIC;”. Besides, the demultiplexer is a 1 to 8 demultiplexer, hence there were 8 outputs declared in the HDL code (O1, O2, O3, O4, O5, O6, O7, O8) as an example “O1: OUT STD_LOGIC;” as output O1. All of them are declared in “STD_LOGIC” type. The structure of the declaration is fulfilling the logic of a 1 to 8 demultiplexer where there were 3 selector inputs, 1 input and 8 output.

Then the code proceeded to architecture section where it starts with the line “ARCHITECTURE bdf_type OF Demultiplexer” which its architecture name is also “bdf_type” and corresponded to “Demultiplexer” entity. Then the architecture section code started with the signal declaration section. As an example, at line “SIGNAL SYNTHESIZED_WIRE_12: STD_LOGIC;” a signal with the name of “SYNTHESIZED_WIRE_12” is declared with the type of “STD_LOGIC” to store logical value.

Then moving on to the behavioural section which started at the “BEGIN” line and ended at the “END bdf_type;” line which also indicated the end of the whole architecture section. This is where all the logics and operations of the demultiplexer were performed. All the behaviour of the demultiplexer is described in concurrent signal assignment statements. The output signals (O1, O2, O3, O4, O5, O6, O7, O8) are assigned values based on the input (I) and the inputs of the selection signals (S0, S1, S2). While for these 3 lines “SYNTHESIZED_WIRE_12 <= NOT(S2);”, “SYNTHESIZED_WIRE_13 <= NOT(S1);” and “SYNTHESIZED_WIRE_14 <= NOT(S0);”, the synthesized wires “SYNTHESIZED_WIRE_12”, “SYNTHESIZED_WIRE_13” and “SYNTHESIZED_WIRE_14” were assigned with the inverse values of “S2”, “S1” and “S0” respectively. Then they are used to carry out AND operations with the input (I) to fulfil the logic of a 1 to 8 demultiplexer. As an example, “O7 <= I AND S2 AND S1 AND SYNTHESIZED_WIRE_14;”, output “O7” is assigned with the AND values of “I”, “S2”, “S1” and “SYNTHESIZED_WIRE_14”. The codes in the behaviour section fully described the behaviour of the demultiplexer circuit drafted in Figure 13.

Full Adder & Full Subtractor

Procedure Full Adder & Full Subtractor

1. The Full Adder and Full Subtractor circuit diagram were drawn using Quartus Prime block diagram.
2. The block diagram was compiled to check if there is any error.
3. The University Program VWF file is created.
4. The inputs were given.
5. The simulation was started to generate the waveforms.
6. The computer was connected to a FPGA.
7. All the inputs and outputs were assigned to pins in the FPGA.
8. The circuit is verified on the FPGA.
9. The HDL code was generated.

Schematic – Full Adder

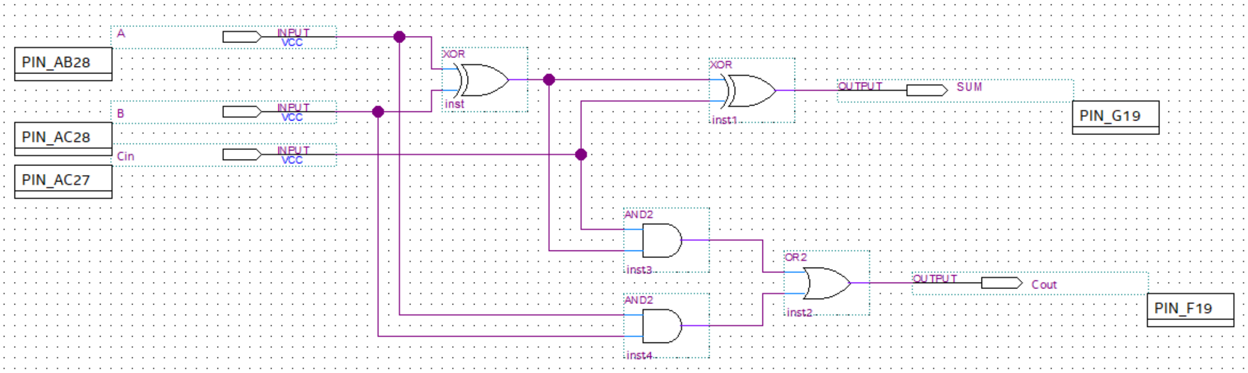
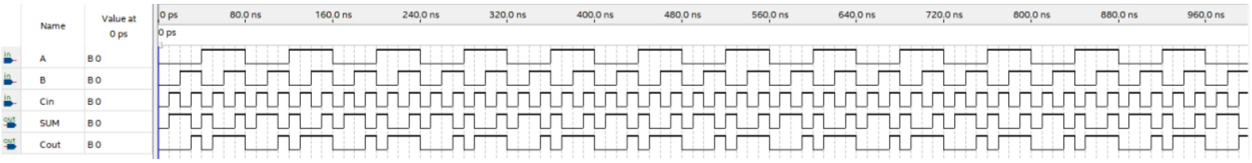


Figure 25: Full Adder Circuit Diagram

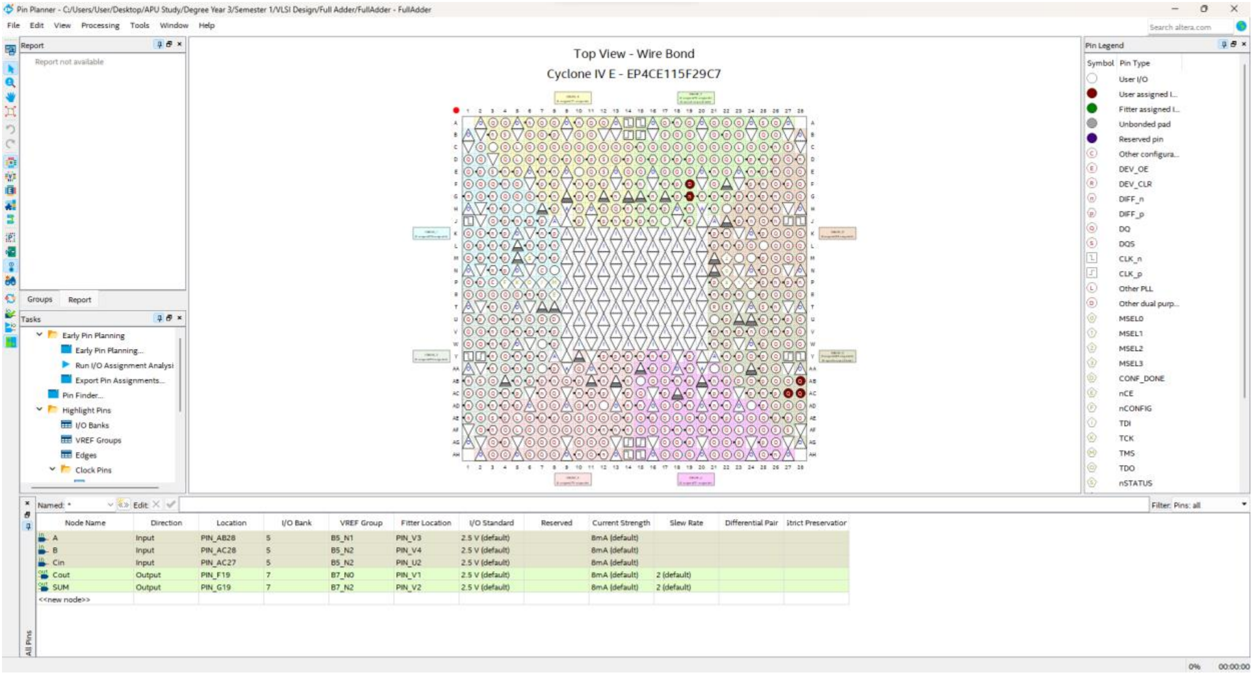
Output Signals (Waveform) – Full Adder



Name	Value at 0 ps
A	B 0
B	B 0
Cin	B 0
SUM	B 0
Cout	B 0

Figure 26: Full Adder Output Waveform

Pin Planner – Full Adder



Pin Planner - C:/Users/User/Desktop/APU Study/Degree Year 3/Semester 1/VLSI Design/Full Adder/FullAdder - FullAdder

Named: * Edit:

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
A	Input	PIN_AB28	5	B5_N1	PIN_V3	2.5 V (default)		8mA (default)			
B	Input	PIN_AC28	5	B5_N2	PIN_V4	2.5 V (default)		8mA (default)			
Cin	Input	PIN_AC27	5	B5_N2	PIN_U2	2.5 V (default)		8mA (default)			
Cout	Output	PIN_F19	7	B7_N0	PIN_V1	2.5 V (default)		8mA (default)	2 (default)		
SUM	Output	PIN_G19	7	B7_N2	PIN_V2	2.5 V (default)		8mA (default)	2 (default)		
<<new node>>											

Figure 27: Full Adder Pin Planner

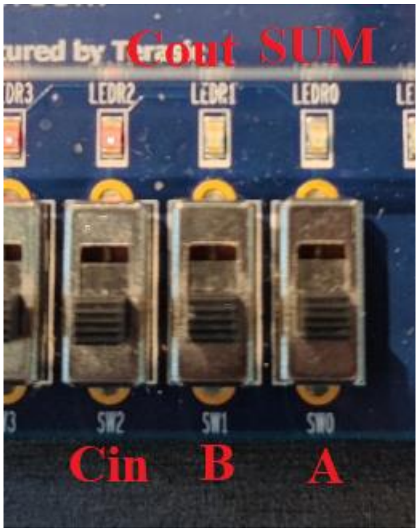


Figure 28: Full Adder FPGA Pins

Truth Table – Full Adder

Table 5: Full Adder Truth Table

Inputs			Outputs	
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

Results from FPGA Device – Full Adder

Table 6: Full Adder FPGA Results

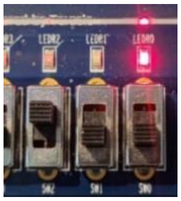
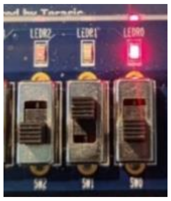
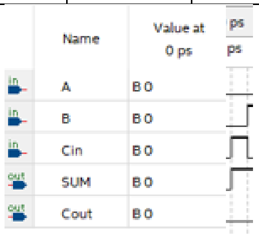
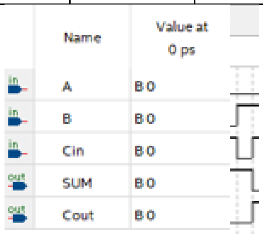
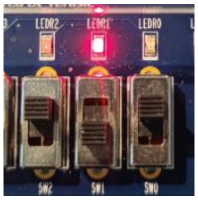
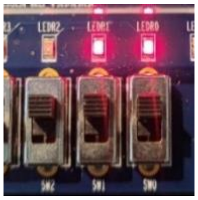
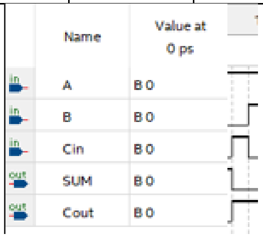
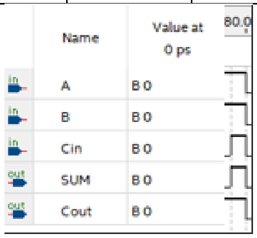
<table>
<tr><td colspan="5">
<em>Figure 29: Full Adder FPGA Result 1</em></td><td colspan="5">
<em>Figure 30: Full Adder FPGA Result 2</em></td></tr>
<tr><td>Cin</td><td>B</td><td>A</td><td>Cout</td><td>SUM</td><td>Cin</td><td>B</td><td>A</td><td>Cout</td><td>SUM</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in Cin B 0
out SUM B 0
out Cout B 0
<em>Figure 31: Full Adder FPGA Result Waveform 1</em></td><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in Cin B 0
out SUM B 0
out Cout B 0
<em>Figure 32: Full Adder FPGA Result Waveform 2</em></td></tr>
<tr><td colspan="5">
<em>Figure 33: Full Adder FPGA Result 3</em></td><td colspan="5">
<em>Figure 34: Full Adder FPGA Result 4</em></td></tr>
<tr><td>Cin</td><td>B</td><td>A</td><td>Cout</td><td>SUM</td><td>Cin</td><td>B</td><td>A</td><td>Cout</td><td>SUM</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in Cin B 0
out SUM B 0
out Cout B 0
<em>Figure 35: Full Adder FPGA Result Waveform 3</em></td><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in Cin B 0
out SUM B 0
out Cout B 0
<em>Figure 36: Full Adder FPGA Result Waveform 4</em></td></tr>
</table>

Figure 25 shows the circuit diagram with the logic circuit of a full adder. The circuit diagram consists of 3 inputs (A, B, Cin) and 2 outputs (SUM, Cout). The whole circuit diagram contains 2 AND gates, 2 XOR gates and 1 OR gate. All the gates only accept 2 inputs. Figure 26 shows the waveform that is generated when the count inputs were given where the signal will only count from 0 to 1 and to 0 again repetitively. Count inputs were given to the different inputs with different counting times. For A the counting time is 40ns, B is 20ns and for Cin is 10ns. The reason for setting the input of the full adder half of the previous counting time input is to make the inputs run like a 3-bit binary counter to make the waveform generated easier to be interpreted and compared with the truth table.

Figure 27 shows the pin planner which shows how the input and output in the circuit diagram are assigned. Whereas Figure 28 indicates the physical locations of the pins when it is connected to the FPGA. Switch SW0 is the input A, SW1 is input B while SW2 is input Cin. While for output LEDR0 is SUM while LEDR1 is Cout.

The full adder is a combinational logic that carries out the addition of 3 input bits which in this case are A, B and Cin. The logic will produce 2 outputs which are the sum (SUM) and the carry out (Cout). The sum output will consider all the inputs A, B and Cin while the carry out (Cout) will take accounts on the inputs and any carry that propagates from the previous stage (Geeks For Geeks, 2023). As an example, when A and B are both 1, then the sum output will be 0 and the carry out will be 1. This is because in binary 1 + 1 will result in 10, hence the sum will get the output of 0 and the 1 will be propagated to the carry out. The truth table is shown in Table 5.

Table 6 displays 4 examples of the result that is obtained from the FPGA with the corresponded input and output waveform attached. The output on the FPGA with different inputs of A, B and Cin are shown in Figure 29, Figure 30, Figure 33 and Figure 34 while their corresponded waveform are shown in Figure 31, Figure 32, Figure 35 and Figure 36. All the results obtained from the FPGA and the generated waveform are verified with Table 5. This indicated that the logic circuit of the full adder is correct. As an example, at Figure 29 the input to Cin at SW2 is high where both A and B at SW0 and SW1 are low. This resulted in the output to be high at SUM and low at Cout where LEDR0 lighted up and LEDR1 did not.

Schematic – Full Subtractor

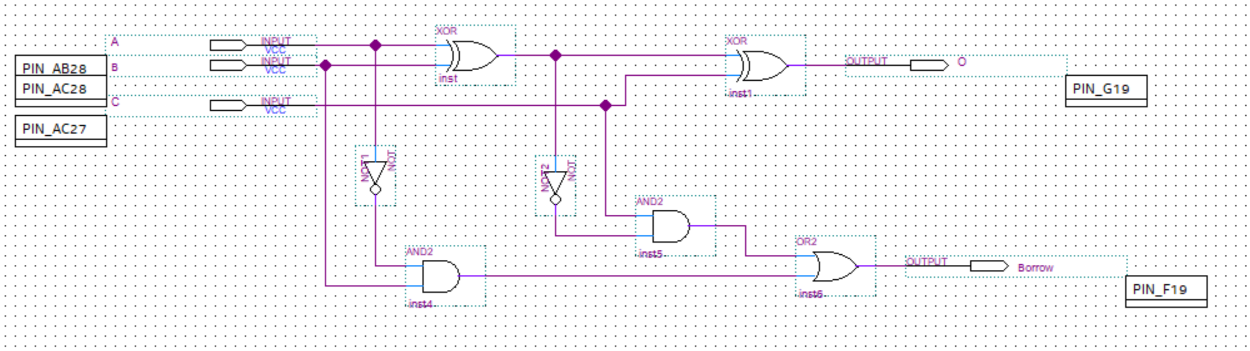


Figure 37: Full Subtractor Circuit Diagram

Output Signals (Waveform) – Full Subtractor

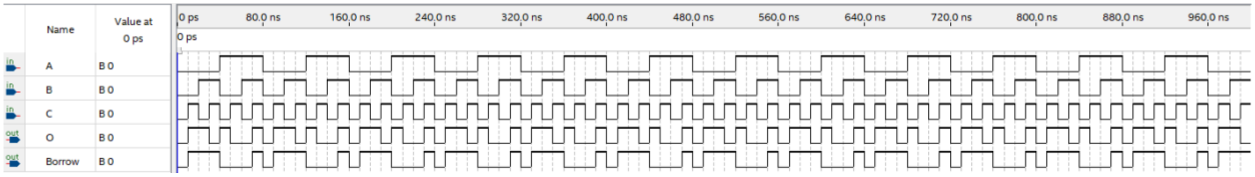
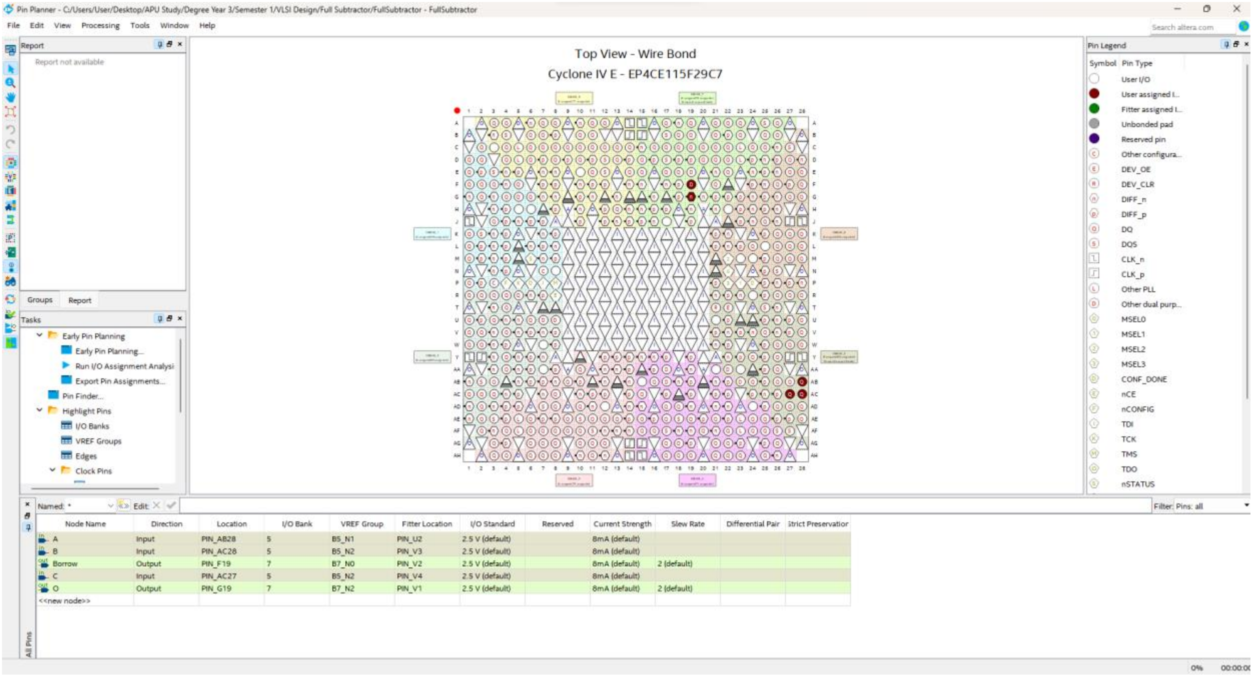


Figure 38: Full Subtractor Output Waveform

Pin Planner – Full Subtractor



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservatior
A	Input	PIN_AB28	5	B5_N1	PIN_U2	2.5 V (default)		8mA (default)			
B	Input	PIN_AC28	5	B5_N2	PIN_V3	2.5 V (default)		8mA (default)			
Borrow	Output	PIN_F19	7	B7_N0	PIN_V2	2.5 V (default)		8mA (default)	2 (default)		
C	Input	PIN_AC27	5	B5_N2	PIN_V4	2.5 V (default)		8mA (default)			
O	Output	PIN_G19	7	B7_N2	PIN_V1	2.5 V (default)		8mA (default)	2 (default)		
<<new node>>											

Figure 39: Full Subtractor Pin Planner

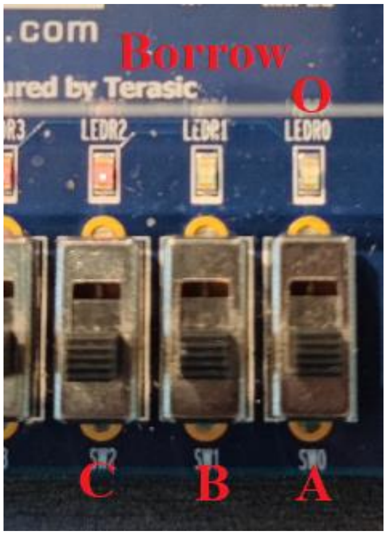


Figure 40: Full Subtractor FPGA Pins

Truth Table – Full Subtractor

Table 7: Full Subtractor Truth Table

Inputs			Outputs	
A	B	C	O	Borrow
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

Results from FPGA Device – Full Subtractor

Table 8: Full Subtractor FPGA Results

<table>
<tr><td colspan="5">

Figure 41: Full Subtractor FPGA Result 1</td><td colspan="5">

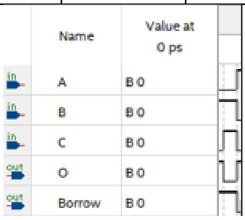
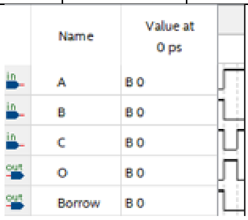
Figure 42: Full Subtractor FPGA Result 2</td></tr>
<tr><td>C</td><td>B</td><td>A</td><td>Borrow</td><td>O</td><td>C</td><td>B</td><td>A</td><td>Borrow</td><td>O</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in C B 0
out O B 0
out Borrow B 0
Figure 43: Full Subtractor FPGA Result Waveform 1</td><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in C B 0
out O B 0
out Borrow B 0
Figure 44: Full Subtractor FPGA Result Waveform 2</td></tr>
<tr><td colspan="5">

Figure 45: Full Subtractor FPGA Result 3</td><td colspan="5">

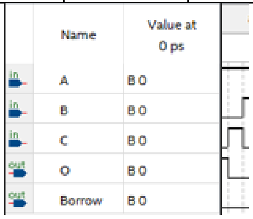
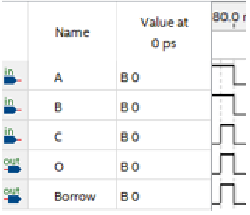
Figure 46: Full Subtractor FPGA Result 4</td></tr>
<tr><td>C</td><td>B</td><td>A</td><td>Borrow</td><td>O</td><td>C</td><td>B</td><td>A</td><td>Borrow</td><td>O</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in C B 0
out O B 0
out Borrow B 0
Figure 47: Full Subtractor FPGA Result Waveform 3</td><td colspan="5">
Name Value at 0 ps
in A B 0
in B B 0
in C B 0
out O B 0
out Borrow B 0
Figure 48: Full Subtractor FPGA Result Waveform 4</td></tr>
</table>

Figure 37 shows the logic circuit diagram of full subtractor. The circuit consists of 3 inputs (A, B, C) and 2 outputs (O, Borrow). The whole circuit diagram contains 2 NOT gates, 2 AND gates, 2 XOR gates and 1 OR gate. All the gates only accept 2 inputs. Figure 38 shows the waveform that is generated when the count inputs were given. The counting time is like what is done in full adder where A is 40ns, B is 20ns and C is 10ns to make the inputs run like a 3-bit binary counter for the ease of interpreting the signals.

Figure 39 displays the pin planner which indicates which pins are the input and output in the full subtractor circuit assigned to. While Figure 40 displays the physical locations of the pins on the FPGA. Switch SW0 is the input A, SW1 is input B while SW2 is input C. While for output LEDR0 is o while LEDR1 is Borrow.

The full subtractor is a combinational logic that carries out subtraction of three input bits which in this case are A, B and C. 2 outputs will be generated by this logic which are O and Borrow. The full subtractor is used to subtract binary numbers and account if there were any borrow from the previous stage of subtraction (Geeks For Geeks, 2023). The difference output (O) represented the result of the subtraction while the borrow-out (Borrow) output represented the borrow generated in the process of subtraction. The difference output (O) considered all the inputs A, B and C where C represented borrow-in. While the borrow-out output (Borrow) takes account the inputs and is there any borrow that propagated from the previous stage. As an example, when the input of A is 0 while the input of B is 1 and the input of C, the borrow-in is 0. The output (O) will be 1 and the borrow-out output (Borrow) will be 1 because 0 – 1 is not possible, hence borrow is needed which resulted the borrow-output to be 1. Hence the equation will be 10 – 1 = 1, which also tells that the output (O) is 1. The truth table of full subtractor is shown in Table 7.

Table 8 displayed the examples of the result that is obtained from the FPGA with its corresponding waveform attached. The input and output of the FPGA are shown in Figure 41, Figure 42, Figure 45 and Figure 46 while their corresponding waveform are shown in Figure 43, Figure 44, Figure 47 and Figure 48. All obtained results are verified with the truth table in Table 7. The logic circuit of full subtractor is verified correct. As an example, at Figure 41 the input to C at SW2 and B at SW1 is high where A at SW0 is low. This resulted in the output to be high at Borrow and low at O where LEDR1 lighted up and LEDR0 did not.

Discussion – Full Adder & Full Subtractor

Full Adder VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY FullAdder IS
        PORT
        (
                A :  IN  STD_LOGIC;
                B :  IN  STD_LOGIC;
                Cin :  IN  STD_LOGIC;
                SUM :  OUT  STD_LOGIC;
                Cout :  OUT  STD_LOGIC
        );
END FullAdder;

ARCHITECTURE bdf_type OF FullAdder IS

SIGNAL          SYNTHESIZED_WIRE_4 :  STD_LOGIC;
SIGNAL          SYNTHESIZED_WIRE_1 :  STD_LOGIC;
SIGNAL          SYNTHESIZED_WIRE_2 :  STD_LOGIC;

BEGIN

SYNTHESIZED_WIRE_4 <= A XOR B;
SUM <= SYNTHESIZED_WIRE_4 XOR Cin;
Cout <= SYNTHESIZED_WIRE_1 OR SYNTHESIZED_WIRE_2;
SYNTHESIZED_WIRE_2 <= Cin AND SYNTHESIZED_WIRE_4;
SYNTHESIZED_WIRE_1 <= A AND B;

END bdf_type;
```

Full Subtractor VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY FullSubtractor IS
        PORT
        (
                A :  IN  STD_LOGIC;
                B :  IN  STD_LOGIC;
                C :  IN  STD_LOGIC;
                Borrow :  OUT  STD_LOGIC;
                O :  OUT  STD_LOGIC
        );
END FullSubtractor;

ARCHITECTURE bdf_type OF FullSubtractor IS

SIGNAL      SYNTHESIZED_WIRE_6 :  STD_LOGIC;
SIGNAL      SYNTHESIZED_WIRE_1 :  STD_LOGIC;
SIGNAL      SYNTHESIZED_WIRE_2 :  STD_LOGIC;
SIGNAL      SYNTHESIZED_WIRE_3 :  STD_LOGIC;
SIGNAL      SYNTHESIZED_WIRE_4 :  STD_LOGIC;

BEGIN

SYNTHESIZED_WIRE_6 <= A XOR B;
O <= SYNTHESIZED_WIRE_6 XOR C;
SYNTHESIZED_WIRE_3 <= SYNTHESIZED_WIRE_1 AND B;
SYNTHESIZED_WIRE_4 <= C AND SYNTHESIZED_WIRE_2;
Borrow <= SYNTHESIZED_WIRE_3 OR SYNTHESIZED_WIRE_4;
SYNTHESIZED_WIRE_1 <= NOT(A);
SYNTHESIZED_WIRE_2 <= NOT(SYNTHESIZED_WIRE_6);

END bdf_type;
```

Section above shows the VHDL code of full adder and full subtractor that is generated by Intel Quartus Prime software. Firstly, for full adder the code started with library declaration “LIBRARY ieee;”. Then the package “std_logic_1164” in the library is imported at line “USE ieee.std_logic_1164.all;”. The reason of declaring the library and importing the package is to allow the use of “STD_LOGIC” type and its related functions to represent digital signals.

Then the code proceeded to the entity declaration section at line “ENTITY FullAdder IS” where the name of the entity is “FullAdder”. The line “PORT” indicated the start of the declaration process. There was a total of 3 inputs and 2 outputs in a full adder logic where the inputs were declared as A, B and Cin. Whereas the outputs were declared as SUM and Cout. As an example, line “A: IN STD_LOGIC;” is where the “A” is declared as an input with the type of “STD_LOGIC” while line “SUM: OUT STD_LOGIC;” is declared as an output also with the type of “STD_LOGIC”. The reason of declaring them in “STD_LOGIC” type is to allow them to store logical values. The structure of the entity declared fulfil the full adder logic where there are 3 inputs and 2 outputs.

Moreover, the code moved on to the architecture declaration part where it started at line “ARCHITECTURE bdf_type OF FullAdder IS” which indicated the name of the architecture is “ndf_type” and corresponded to the “FullAdder” entity. The architecture section will contain 2 other section which are signal declaration section and behavioural section. The whole architecture section ended at line “END bdf_type”.

In the architecture section, signal declaration started first. Take line “SIGNAL SYNTHESIZED_WIRE_4: STD_LOGIC;” as an example, a signal with the name of “SYNTHESIZED_WIRE_4” is declared using the keyword “SIGNAL” which is commonly used in VHDL to declare signals. Signals is like a variable that used to store values. But in this case, it only stores logical values. 3 signals were declared in this section which were “SYNTHESIZED_WIRE_4”, “SYNTHESIZED_WIRE_1” and “SYNTHESIZED_WIRE_2”. They were intermediate wire signals that will be used in the architecture.

Furthermore, the behavioural section started at line “BEGIN”. This is where the behaviour of the full adder is described using concurrent signal assignment statements. This means that all lines were running parallelly at the same time. As an example, for line “SYNTHESIZED_WIRE_4 <= A XOR B;”, “SYNTHESIZED_WIRE_4” is assigned the result of the XOR operation between inputs A and B. At line “SUM <= SYNTHESIZED_WIRE_4 XOR Cin;”, The sum output “SUM” is assigned the result of the XOR operation between “SYNTHESIZED_WIRE_4” and the carry-in “Cin”. At line “Cout <= SYNTHESIZED_WIRE_1 OR SYNTHESIZED_WIRE_2;”, The carry-out “Cout” is assigned the result of the logical OR operation between “SYNTHESIZED_WIRE_1” and “SYNTHESIZED_WIRE_2”. At line “SYNTHESIZED_WIRE_2 <= Cin AND SYNTHESIZED_WIRE_4;”, “SYNTHESIZED_WIRE_2” is assigned the result of the logical AND operation between “Cin” and “SYNTHESIZED_WIRE_4”. Lastly at line “SYNTHESIZED_WIRE_1 <= A AND B;”, “SYNTHESIZED_WIRE_1” is assigned the result of the logical AND operation between inputs “A” and “B”. Then these codes described and concluded the behaviour of a full adder circuit drafted in Figure 25.

The format of VHDL code for the full subtractor is much similar to the code for full adder just with different behaviour and signal names. The code also started with library declaration at line “LIBRARY ieee;” and “USE ieee.std_logic_1164.all;”. The IEEE standard is used, which provides the STD_LOGIC type and related functions for representing digital signals.

Then the code moved into the entity declaration section to declare all the input and output of full subtractor. The declaration starts with “ENTITY FullSubtractor IS” and “PORT” which will contain all the input and output ports. In the code 3 input ports A, B and C are declared and also 2 output ports Borrow and O. C represents the borrow-in while Borrow represents the borrow-out in full subtractor logic. As an example, in line “A: IN STD_LOGIC;” the input port “A” is declared as “STD_LOGIC” type to store logical values. While at line “Borrow: OUT STD_LOGIC;”, “Borrow” output port is declared also as a “STD_LOGIC” type.

Moving on to the architecture section, the line “ARCHITECTURE bdf_type OF FullSubtractor IS” indicated that the architecture is named as “bdf_type” and it is corresponded to the “FullSubtractor” entity. The architecture ended at line “END bdf_type”.

The code continued with the signal declaration in the architecture section. Within the signal declaration section in full subtractor, 5 signals were declared which were “SYNTHESIZED_WIRE_6”, “SYNTHESIZED_WIRE_1”, “SYNTHESIZED_WIRE_2”, “SYNTHESIZED_WIRE_3” and “SYNTHESIZED_WIRE_4”. All the signals were declared with the type of “STD_LOGIC” to store logical values for intermediate computations.

Lastly the code proceeded to behavioural section where it started at line “BEGIN”. The behaviour of full subtractor were described here with also with concurrent signal assignment statements. There are a total of 7 statements which is same with the number of logical components or gates that is used in the full subtractor circuit. For line “SYNTHESIZED_WIRE_6 <= A XOR B;”, “SYNTHESIZED_WIRE_6” is assigned the result of the XOR operation between inputs “A” and “B”. For line “O <= SYNTHESIZED_WIRE_6 XOR C;”, the difference output “O” is assigned the result of the XOR operation between “SYNTHESIZED_WIRE_6” and “C”. For line “SYNTHESIZED_WIRE_3 <= SYNTHESIZED_WIRE_1 AND B;”, SYNTHESIZED_WIRE_3 is assigned the result of the logical AND operation between “SYNTHESIZED_WIRE_1” and “B”. While line “SYNTHESIZED_WIRE_4 <= C AND SYNTHESIZED_WIRE_2;”, “SYNTHESIZED_WIRE_4” is assigned the result of the logical AND operation between “C” and “SYNTHESIZED_WIRE_2”. Whereas for line “Borrow <= SYNTHESIZED_WIRE_3 OR SYNTHESIZED_WIRE_4;”, the borrow-out “Borrow” is assigned the result of the logical OR operation between “SYNTHESIZED_WIRE_3” and “SYNTHESIZED_WIRE_4”. Then line “SYNTHESIZED_WIRE_1 <= NOT(A);”, “SYNTHESIZED_WIRE_1” is assigned the inverse (NOT) of “A”. Lastly, for line “SYNTHESIZED_WIRE_2 <= NOT(SYNTHESIZED_WIRE_6);”, SYNTHESIZED_WIRE_2 is assigned the inverse (NOT) of “SYNTHESIZED_WIRE_6”. Then these lines fully described the logic circuit in Figure 37 using VHDL code.

SR & JK Flip Flop

Procedure SR & JK Flip Flop

1. The SR and JK Flip Flop circuit diagram were drawn using Quartus Prime block diagram.
2. The block diagram was compiled to check if there is any error.
3. The University Program VWF file is created.
4. The inputs were given.
5. The simulation was started to generate the waveforms.
6. The computer was connected to a FPGA.
7. All the inputs and outputs were assigned to pins in the FPGA.
8. The circuit is verified on the FPGA.
9. The HDL code was generated.

Schematic – SR Flip Flop

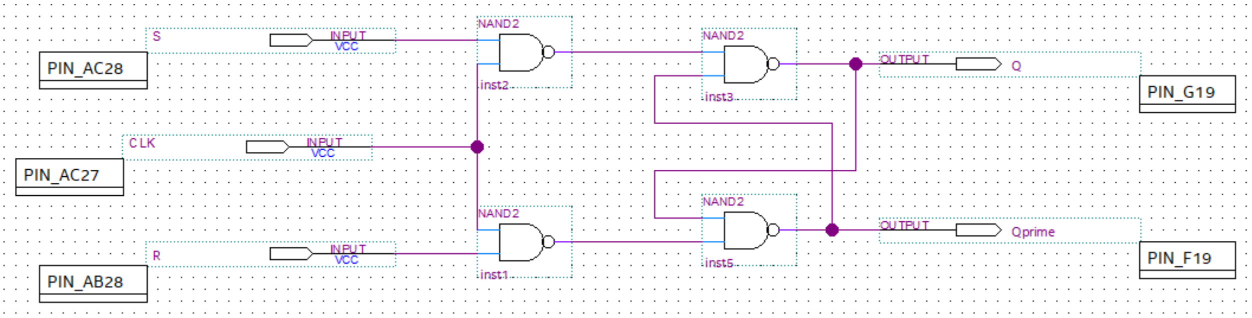
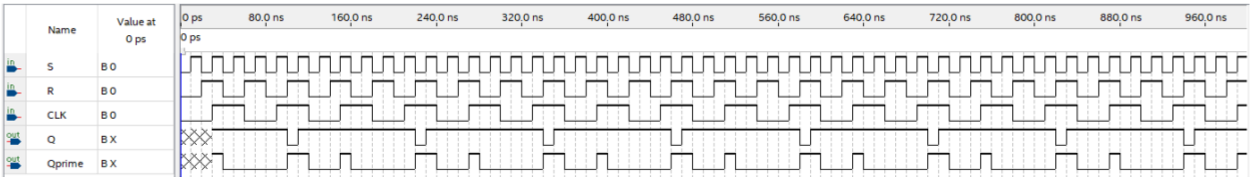


Figure 49: SR Flip Flop Circuit Diagram

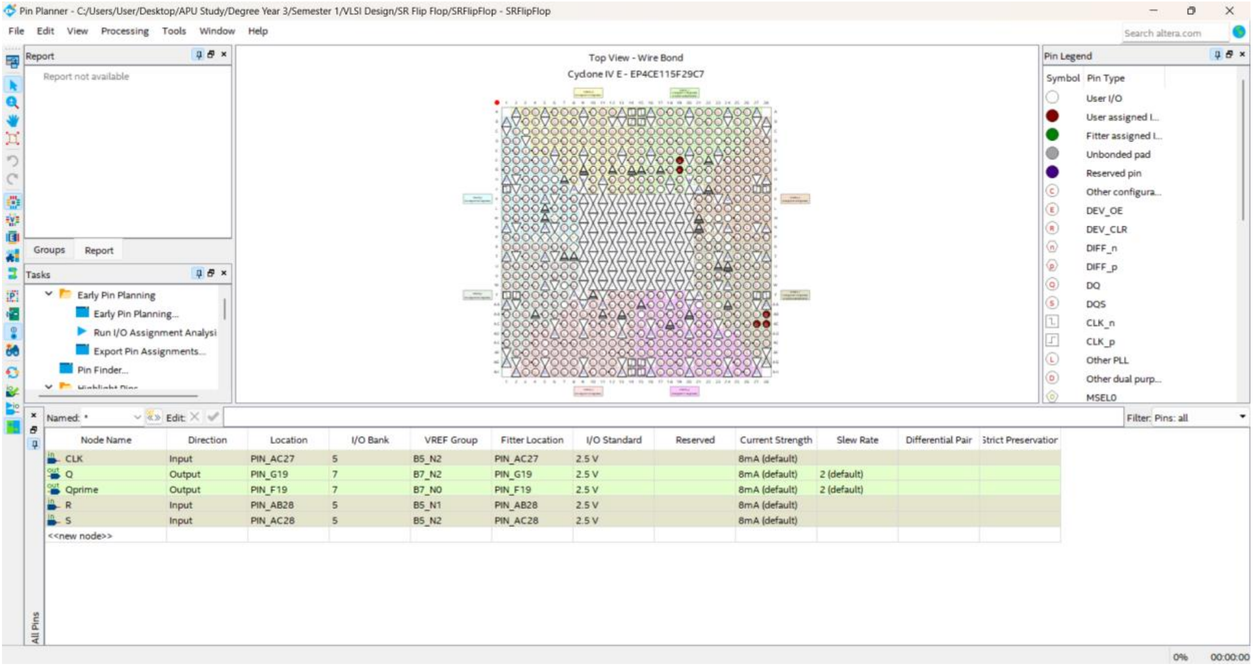
Output Signals (Waveform) – SR Flip Flop



Name	Value at 0 ps
S	B 0
R	B 0
CLK	B 0
Q	B X
Qprime	B X

Figure 50: SR Flip Flop Output Waveform

Pin Planner – SR Flip Flop



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservatior
CLK	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V		8mA (default)			
Q	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
Qprime	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)		
R	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			
S	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V		8mA (default)			
<<new node>>											

Figure 51: SR Flip Flop Pin Planner

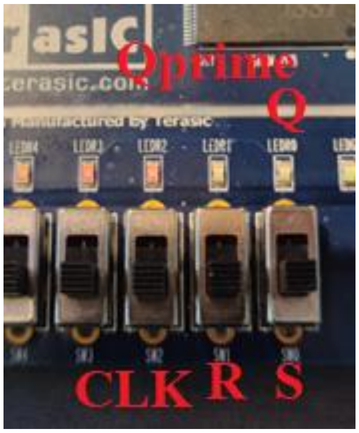


Figure 52: SR Flip Flop FPGA Pins

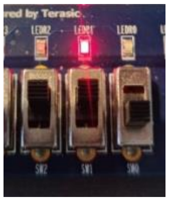
Truth Table – SR Flip Flop

Table 9: SR Flip Flop Truth Table

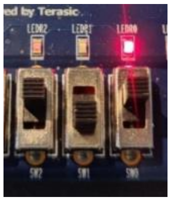
Inputs				Outputs		State
CLK	S	R	Qprevious	Q	Qprime	
1	0	0	0	0	1	No Change
1	0	0	1	1	0	
1	0	1	0	0	1	Reset
1	0	1	1	0	1	
1	1	0	0	1	0	Set
1	1	0	1	1	0	
1	1	1	0	X	X	Invalid
1	1	1	1	X	X	
0	X	X	0	0	1	No Change
0	X	X	1	1	0	

Results from FPGA Device – SR Flip Flop

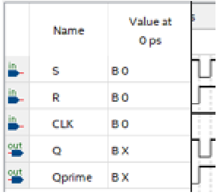
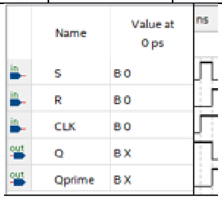
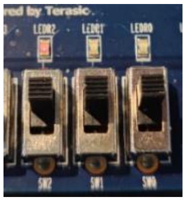
Table 10: SR Flip Flop FPGA Results

<table>
<tr><td colspan="5">

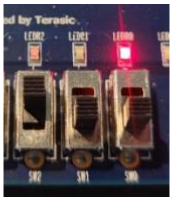
red by Terasic
LEDR2
LEDR1
LEDR0
SW2
SW1
SW0

Figure 53: SR Flip Flop FPGA Result 1
Reset</td><td colspan="5">

ed by Terasic
LEDR2
LEDR1
LEDR0
SW2
SW1
SW0

Figure 54: SR Flip Flop FPGA Result 2
Set</td></tr>
<tr><td>CLK</td><td>R</td><td>S</td><td>Qprime</td><td>Q</td><td>CLK</td><td>R</td><td>S</td><td>Qprime</td><td>Q</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
S B 0
R B 0
CLK B 0
Q B X
Qprime B X
Figure 55: SR Flip Flop FPGA Result Waveform 1</td><td colspan="5">
Name Value at 0 ps
S B 0
R B 0
CLK B 0
Q B X
Qprime B X
Figure 56: SR Flip Flop FPGA Result Waveform 2</td></tr>
<tr><td colspan="5">

red by Terasic
LEDR2
LEDR1
LEDR0
SW2
SW1
SW0

Figure 57: SR Flip Flop FPGA Result 3
Invalid</td><td colspan="5">

ed by Terasic
LEDR2
LEDR1
LEDR0
SW2
SW1
SW0

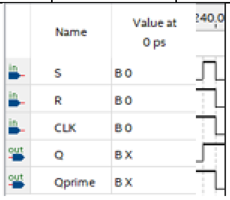
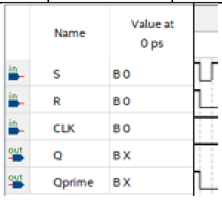
Figure 58: SR Flip Flop FPGA Result 4
No Change</td></tr>
<tr><td>CLK</td><td>R</td><td>S</td><td>Qprime</td><td>Q</td><td>CLK</td><td>R</td><td>S</td><td>Qprime</td><td>Q</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps
S B 0
R B 0
CLK B 0
Q B X
Qprime B X
Figure 59: SR Flip Flop FPGA Result Waveform 3</td><td colspan="5">
Name Value at 0 ps
S B 0
R B 0
CLK B 0
Q B X
Qprime B X
Figure 60: SR Flip Flop FPGA Result Waveform 4</td></tr>
</table>

Figure 49 shows the logic circuit diagram of a SR flip flop. The logic circuit of a SR flip flop consisted of 3 inputs which are “S”, “R”, “CLK” and also 2 outputs “Q” and “Qprime” which is the inverse of “Q”. The whole circuit diagram contains 4 NAND gates. All the gates have 2 input ports. Figure 50 shows the waveform that was generated after the count inputs were given. Count input is input that only counts from 0 to 1 and then 0 repetitively. The inputs were given with different counting time which are 10ns for “S”, 20ns for “R” and 30ns for “CLK”. The reason of giving count input with different counting times is to obtain the output with all different types of inputs.

Figure 51 shows the pin planner of SR flip flop. It shows which pins are the inputs and outputs assigned to. Figure 52 shows the physical location of the pins with their corresponding inputs and outputs on the FPGA. Switch SW0 is input “S”, SW1 is input “R” while SW2 is input “CLK”. Whereas for output, output “Q” is LEDR0 while output “Qprime” is LEDR1.

The SR flip flop, which is also commonly known as the Set-Reset flip flop is one of the basic sequential logic circuits that can store 1 bit of information. A SR flip flop will accept 3 inputs which are S (Set), R (Reset) and CLK (Clock Signal). The clock signal determines when the inputs to S and R are allowed to affect the output state of the SR flip flop (Geeks For Geeks, 2023). “S” (Set) is the input that sets the output state of the flip flop to high which is 1. When input of 1 is given into “S” and input of 0 is given into “R”, the output of the flip flop “Q” will be 1. “R” (Reset) is the input that resets the output of the flip flop to low which is 0. When input of 1 is given into “R” and input of 0 is given into “S”, the output of the flip flop “Q” will be 0. While CLK (Clock) is where the input signal controls when the output state of the flip flop will be updated. Normally there are 2 types of triggering methods which are rising edge and falling edge. Which means that the SR flip flop will update the output state according to the inputs given into “S” and “R” when there is a rising edge where the signal turns from 0 to 1 or falling edge where signal turns from 1 to 0 in the “CLK” clock input depending on the type of the SR flip flop. The SR flip flop will only update the output state when the required triggering method is achieved. Hence, there should be no changes at the output stage even when the input given into “S” and “R” is changed into neither set nor reset when there was no rising edge or falling edge at the clock input. When input to “S” and “R” were both logically low (0) when there is a rising or falling edge, the output state remains unchanged. However, when both “S” and “R” inputs are active (logic high) is generally undefined or considered invalid. This situation is known as the "forbidden state" and should be avoided to ensure proper operation of the flip flop. The truth table of SR flip flop is shown in Table 9. According to Figure 50, the output of “Q” and “Qprime” also shows invalid state when both “S” and “R” have the input of 1 where both “Q” and “Qprime” both shows the output of 1 which is invalid because the “Qprime” must always be the complementary of Q.

Nevertheless, when the logic is tested on the FPGA, it reacted more similar to a gated SR latch where the output will always be affected by the change of input in “S” and “R” once the input to the “CLK” is always 1. This means that it is level triggered instead of edge triggered. This is due to there are no edge detector integrated in the SR flip flop circuit. Hence the whole circuit acted like a gated SR latch. This can also be acknowledged by looking at the waveform generated in Figure 50 where the output “Q” and “Qprime” changed when the “CLK” is 1 even there were no edges.

Table 10 displays 4 examples of the result that is obtained from connecting the circuit to the FPGA device with its corresponding waveform attached. The input and output of FPGA are shown in Figure 53, Figure 54, Figure 57 and Figure 58. While their corresponding waveform according to the input and output in the FPGA are shown in Figure 55, Figure 56, Figure 59 and Figure 60. As an example, at Figure 53 the input to “CLK” and “R” which corresponded to SW2 and SW1 is high while the input “S” at SW0 is low. This resulted in the SR flip flop to be in reset state where the output “Q” at LEDR0 is not lighted up indicating low and its complementary “Qprime” at LEDR1 is lighted up indicating high. It is verified that the result obtained from the FPGA are the same with the waveform generated by the Intel Quartus Prime software so does with the truth table in Table 9. Hence the logic circuit is correct but only with a flaw which it acts like a gated SR latch due to lack of an edge detector component where it should be integrated with the clock input (CLK) to detect only the rising edge or the falling edge of the clock input.

Schematic – JK Flip Flop (Circuit)

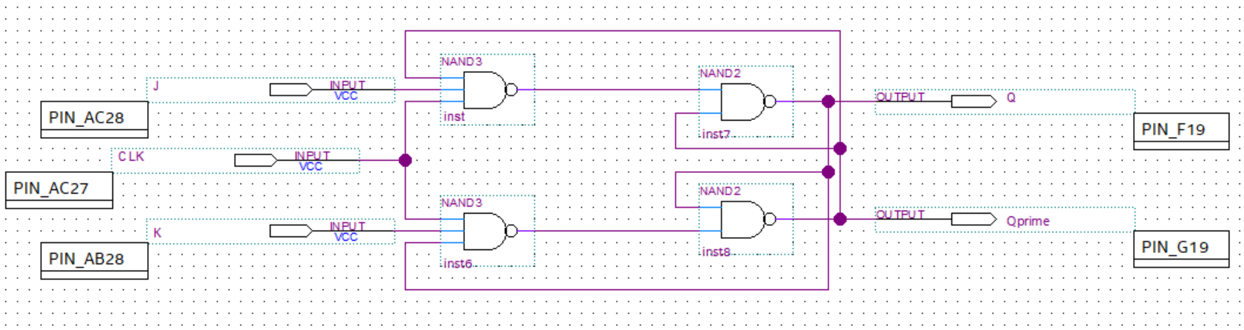


Figure 61: JK Flip Flop Circuit Diagram (Circuit)

Schematic – JK Flip Flop (Block)

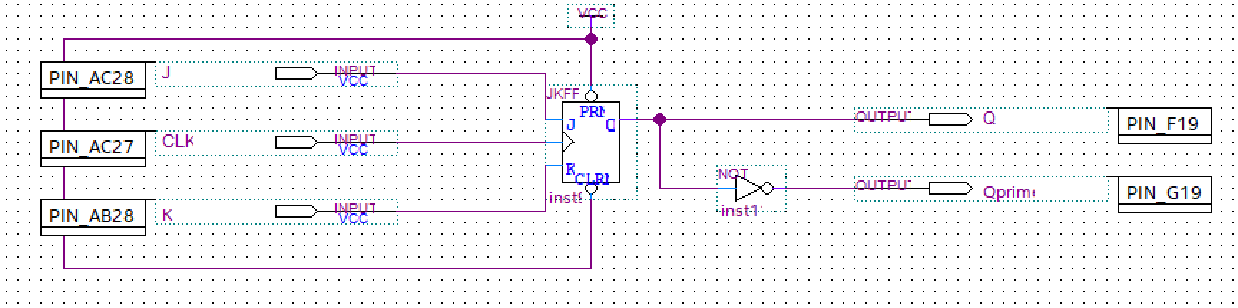


Figure 62: JK Flip Flop Circuit Diagram (Block)

Output Signals (Waveform) – JK Flip Flop (Circuit)

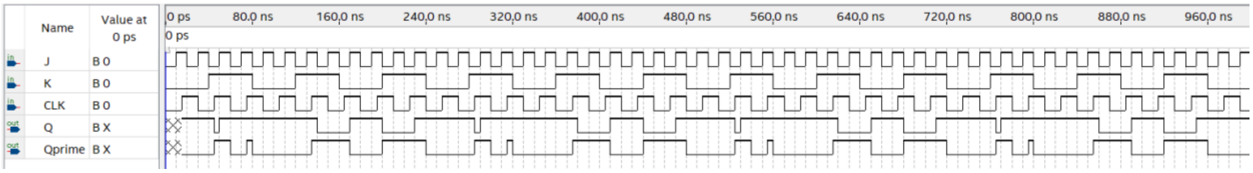


Figure 63: JK Flip Flop (Circuit) Output Waveform

Output Signals (Waveform) – JK Flip Flop (Block)

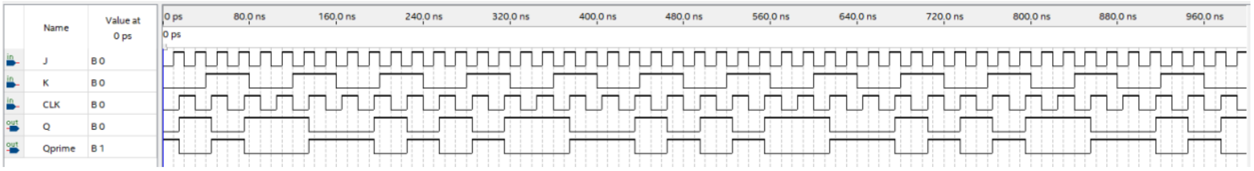
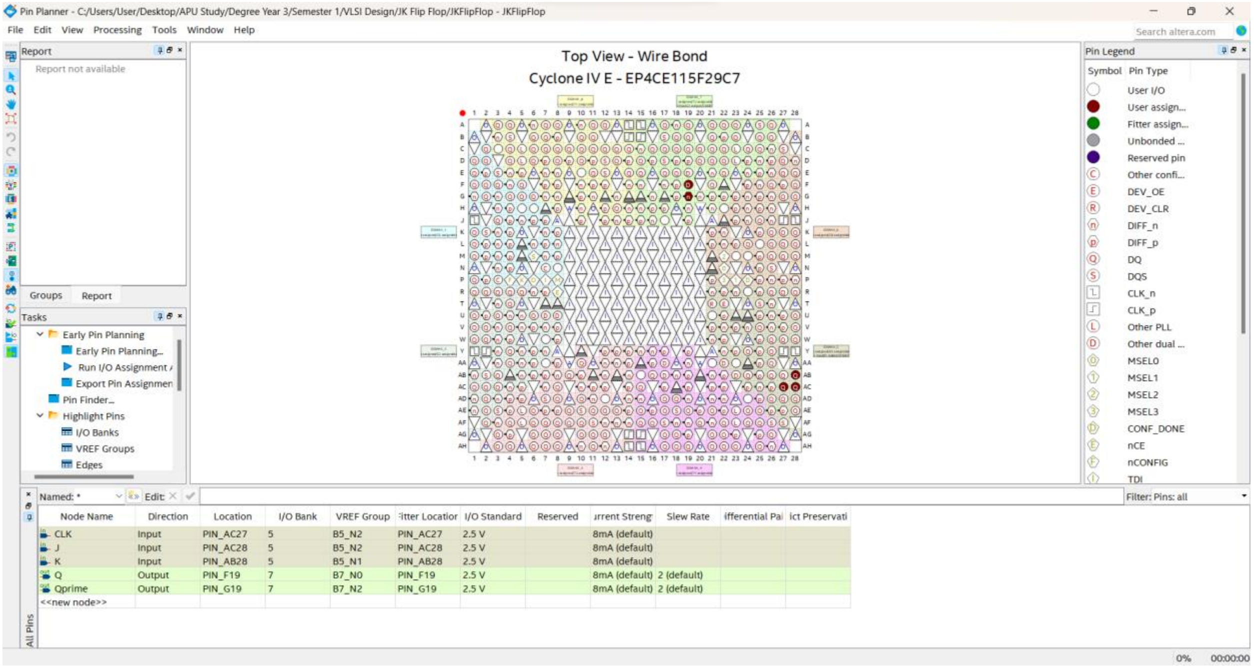


Figure 64: JK Flip Flop (Block) Output Waveform

Pin Planner – JK Flip Flop



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
CLK	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V		8mA (default)			
J	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V		8mA (default)			
K	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V		8mA (default)			
Q	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V		8mA (default)	2 (default)		
Qprime	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V		8mA (default)	2 (default)		
<<new node>>											

Figure 65: JK Flip Flop Pin Planner

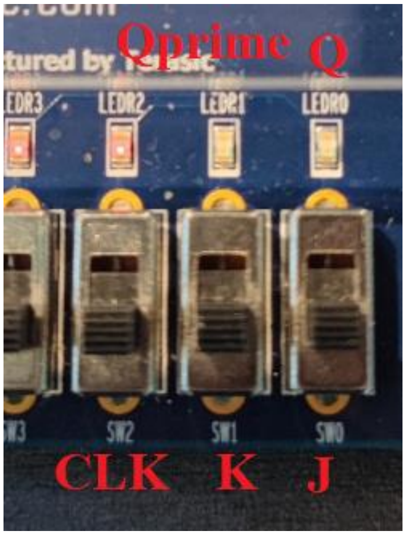


Figure 66: JK Flip Flop FPGA Pins

Truth Table – JK Flip Flop

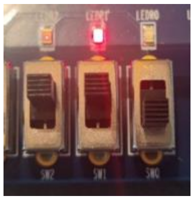
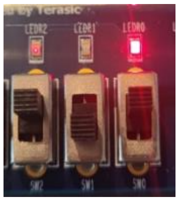
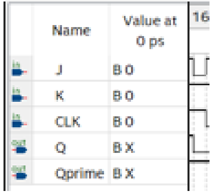
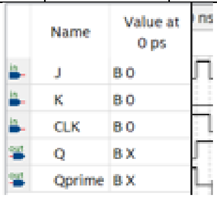
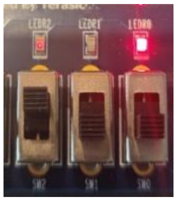
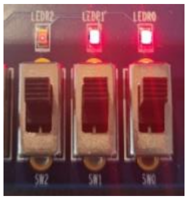
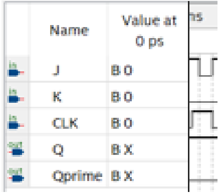
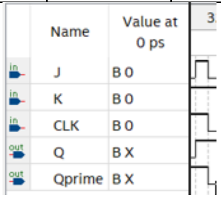
Table 11: JK Flip Flop Truth Table

Inputs				Outputs		State
CLK	J	K	Qprevious	Q	Qprime	
1	0	0	0	0	1	No Change
1	0	0	1	1	0	
1	0	1	0	0	1	Reset
1	0	1	1	0	1	
1	1	0	0	1	0	Set
1	1	0	1	1	0	
1	1	1	0	1	0	Toggle
1	1	1	1	0	1	
0	X	X	0	0	1	No Change
0	X	X	1	1	0	

Results from FPGA Device – JK Flip Flop

<u>Circuit</u>

Table 12: JK Flip Flop FPGA Results (Circuit)

 *Figure 67: JK Flip Flop (Circuit) FPGA Result 1* Reset					 *Figure 68: JK Flip Flop (Circuit) FPGA Result 2* Set				
CLK	K	J	Qprime	Q	CLK	K	J	Qprime	Q
1	1	0	1	0	1	0	1	0	1
 Name Value at 0 ps J B 0 K B 0 CLK B 0 Q B X Qprime B X *Figure 69: JK Flip Flop (Circuit) FPGA Result Waveform 1*					 Name Value at 0 ps J B 0 K B 0 CLK B 0 Q B X Qprime B X *Figure 70: JK Flip Flop (Circuit) FPGA Result Waveform 2*				
 *Figure 71: JK Flip Flop (Circuit) FPGA Result 3* No Change					 *Figure 72:: JK Flip Flop (Circuit) FPGA Result 4* Rapid Toggle				
CLK	K	J	Qprime	Q	CLK	K	J	Qprime	Q
1	0	0	0	1	1	0	0	0	1
 Name Value at 0 ps J B 0 K B 0 CLK B 0 Q B X Qprime B X *Figure 73: JK Flip Flop (Circuit) FPGA Result Waveform 3*					 Name Value at 0 ps J B 0 K B 0 CLK B 0 Q B X Qprime B X *Figure 74: JK Flip Flop (Circuit) FPGA Result Waveform 4*				

Block

Table 13: JK Flip Flop FPGA Results (Block)

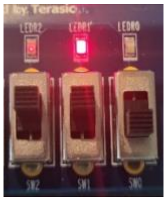
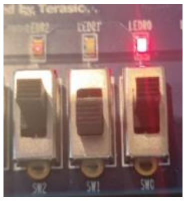
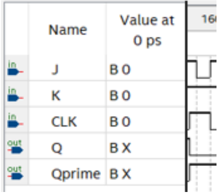
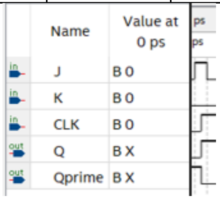
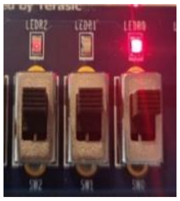
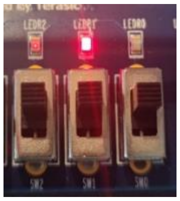
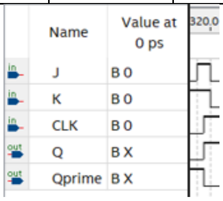
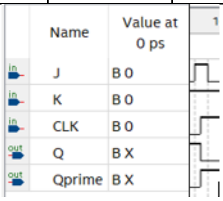
<table>
<tr><td colspan="5">
Figure 75: JK Flip Flop (Block) FPGA Result 1
Reset</td><td colspan="5">
Figure 76: JK Flip Flop (Block) FPGA Result 2
Set</td></tr>
<tr><td>CLK</td><td>K</td><td>J</td><td>Qprime</td><td>Q</td><td>CLK</td><td>K</td><td>J</td><td>Qprime</td><td>Q</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td colspan="5">
Name Value at 0 ps 16
J B 0
K B 0
CLK B 0
Q B X
Qprime B X
Figure 77: JK Flip Flop (Block) FPGA Result Waveform 1</td><td colspan="5">
Name Value at 0 ps ps
J B 0
K B 0
CLK B 0
Q B X
Qprime B X
Figure 78: JK Flip Flop (Block) FPGA Result Waveform 2</td></tr>
<tr><td colspan="5">
Figure 79: JK Flip Flop (Block) FPGA Result 3
Toggle 1</td><td colspan="5">
Figure 80: JK Flip Flop (Block) FPGA Result 4
Toggle 2</td></tr>
<tr><td>CLK</td><td>K</td><td>J</td><td>Qprime</td><td>Q</td><td>CLK</td><td>K</td><td>J</td><td>Qprime</td><td>Q</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">
Name Value at 0 ps 320,0
J B 0
K B 0
CLK B 0
Q B X
Qprime B X
Figure 81: JK Flip Flop (Block) FPGA Result Waveform 3</td><td colspan="5">
Name Value at 0 ps 1
J B 0
K B 0
CLK B 0
Q B X
Qprime B X
Figure 82: JK Flip Flop (Block) FPGA Result Waveform 4</td></tr>
</table>

Figure 61 shows the logic circuit diagram of JK flip flop. While Figure 62 also shows the circuit diagram of JK flip flop but using the JK flip flop provided by the Intel Quartus Prime software instead of using logic gates to compile the logic circuit of the JK flip flop. Similar to SR flip flop, JK flip flop also consisted of 3 inputs and 2 outputs. The inputs were “J”, “K” and “CLK”. The outputs consisted of “Q” and its complementary “Qprime”. The logic circuit diagram Figure 61 consisted of 4 NAND gates 2 with 3 inputs and 2 with 2 inputs. Where the diagram in Figure 62 only consisted of a JK flip flop block and 1 NOT gate to get the inverse output of “Q” which is “Qprime”. VCC is also connected to the JK flip flop block to supply power to the JK flip flop to turn it on. Figure 63 and Figure 64 display the generated waveform for both circuit when count input is given. The counting time input into both circuit is the same where the counting time for “J” is 10ns, for “K” is 40ns and “CLK” is 20ns. This is to ensure all type of input combination is available in the waveform to generate all possible outputs.

Figure 65 shows the pin planner for both JK flip flop circuit diagram where both of the circuit have the same amount of input, output and names just with different type of presentation of logics. Figure 66 displays the physical pins locations of the inputs and outputs on the FPGA for easier understanding on the FPGA result section. Switch SW0 is the input “J”, SW1 is input “K” while SW2 is input "CLK”. For output “Q” is LEDR0 and “Qprime” is LEDR1.

JK flip flop is also a type of sequential logic circuit that can store 1 bit of information similar to the SR flip flop. The main difference between a SR and JK flip flop is that JK flip flop does not have an invalid state and have the ability to toggle its state based on the input signals (Geeks For Geeks, 2020). In short there are a total of 4 valid states in JK flip flop which are set, reset, toggle and no change, where there are only 3 valid states in SR flip flop which are set, reset and no change. JK flip flop also have 2 types of triggering methods which are rising edge and falling edge according to the clock input “CLK” depending on what type of JK flip flop is used. When the JK flip flop is triggered, “J” = 0 and “K” = 1, the flip flop is in reset state where the output “Q” will be reset to 0 and “Qprime” will be 1. When JK flip flop is triggered, “J” = 1, “K” = 0, the flip flop is in set state where “Q” is set to 1 and “Qprime” is 0. When “J” and “K” are both input with 1 when triggered, the flip flop is in toggle state. If the current state is “Q” = 0, it changes to “Q” = 1 and “Qprime” = 0. If the current state is “Q” = 1, it changes to “Q” = 0 and “Qprime” = 1. Lastly when “J” = 0 and “K” = 0 when triggered, the flip-flop maintains its current state and the outputs “Q” and “Qprime” remain unchanged. The flip flop will also remain unchanged even there are changes in “J” and “K”, but the flip flop is not triggered by rising or falling edge depending on the type of JK flip flop. There is no invalid state in JK flip flop. The truth table of JK flip flop is shown in Table 11.

Table 12 shows 4 examples of the output results from FPGA and their corresponding output waveform while connecting the circuit diagram in Figure 61. While connecting the circuit diagram in Figure 61 to the FPGA, the set, reset and no change state works normal as what is listed in truth table Table 11. However, when it comes to toggle state, the JK flip flop toggle at a rapid speed due to the absence of the edge detector in the circuit. Causing the JK flip flop to toggle whenever the input to “CLK” is 1. The JL flip flop too fast until both output “Q” and “Qprime” in the FPGA to light up indicating the output is 1 as shown in Figure 72. In Figure 74, the corresponding waveform also indicated that both output “Q” and “Qprime” are both 1 due to the absence of edge detector. In addition, according to Figure 74 it is also known that the output “Q” and “Qprime” also changed even there were no presence of edge at “CLK”. The results were verified correct with the truth table Table 11 except toggle state.

Table 13 shows 4 examples of the FPGA output results while connecting the circuit diagram in Figure 62 that is using the JK flip flop block provided by the software. All the states, set, reset, toggle and no change were working normally where the output will only change when there is an edge detected in the “CLK” input. The outputs “Q” and “Qprime” will not change even the input to “J” and “K” is changed when there are no changes or edge detected in the “CLK” clock input. According to Figure 64, the JK flip flop is positive edge triggered where the output will only change when the “CLK” changed from 0 to 1. This indicated that there was a positive edge detector component in the provided JK flip flop block where the output will only be affected when there was a positive edge. The results were verified all correct with the truth table in Table 11.

Discussion – SR & JK Flip Flop

SR Flip Flop VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY SRFlipFlop IS
        PORT
        (
                S :  IN  STD_LOGIC;
                R :  IN  STD_LOGIC;
                CLK :  IN  STD_LOGIC;
                Q :  OUT  STD_LOGIC;
                Qprime :  OUT  STD_LOGIC
        );
END SRFlipFlop;

ARCHITECTURE bdf_type OF SRFlipFlop IS

SIGNAL          SYNTHESIZED_WIRE_0 :  STD_LOGIC;
SIGNAL          SYNTHESIZED_WIRE_1 :  STD_LOGIC;
SIGNAL          SYNTHESIZED_WIRE_2 :  STD_LOGIC;
SIGNAL          SYNTHESIZED_WIRE_3 :  STD_LOGIC;

BEGIN

Q <= SYNTHESIZED_WIRE_3;
Qprime <= SYNTHESIZED_WIRE_0;
SYNTHESIZED_WIRE_2 <= NOT(R AND CLK);
SYNTHESIZED_WIRE_1 <= NOT(CLK AND S);
SYNTHESIZED_WIRE_3 <= NOT(SYNTHESIZED_WIRE_0 AND
SYNTHESIZED_WIRE_1);
SYNTHESIZED_WIRE_0 <= NOT(SYNTHESIZED_WIRE_2 AND
SYNTHESIZED_WIRE_3);

END bdf_type;
```

JK Flip Flop VHDL (Circuit)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY JKFlipFlop IS
        PORT
        (
                CLK :  IN  STD_LOGIC;
                J :  IN  STD_LOGIC;
                K :  IN  STD_LOGIC;
                Q :  OUT  STD_LOGIC;
                Qprime :  OUT  STD_LOGIC
        );
END JKFlipFlop;

ARCHITECTURE bdf_type OF JKFlipFlop IS

SIGNAL        SYNTHESIZED_WIRE_6 :  STD_LOGIC;
SIGNAL        SYNTHESIZED_WIRE_7 :  STD_LOGIC;
SIGNAL        SYNTHESIZED_WIRE_3 :  STD_LOGIC;
SIGNAL        SYNTHESIZED_WIRE_4 :  STD_LOGIC;

BEGIN

Q <= SYNTHESIZED_WIRE_7;
Qprime <= SYNTHESIZED_WIRE_6;
SYNTHESIZED_WIRE_3 <= NOT(SYNTHESIZED_WIRE_6 AND J AND CLK);
SYNTHESIZED_WIRE_4 <= NOT(CLK AND K AND SYNTHESIZED_WIRE_7);
SYNTHESIZED_WIRE_7 <= NOT(SYNTHESIZED_WIRE_6 AND
SYNTHESIZED_WIRE_3);
SYNTHESIZED_WIRE_6 <= NOT(SYNTHESIZED_WIRE_4 AND
SYNTHESIZED_WIRE_7);

END bdf_type;
```

JK Flip Flop VHDL (Block)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY JKFlipFlopBlock IS
        PORT
        (
                CLK :  IN  STD_LOGIC;
                J :  IN  STD_LOGIC;
                K :  IN  STD_LOGIC;
                Q :  OUT  STD_LOGIC;
                Qprime :  OUT  STD_LOGIC
        );
END JKFlipFlopBlock;

ARCHITECTURE bdf_type OF JKFlipFlopBlock IS

SIGNAL        JKFF_inst9 :  STD_LOGIC;
SIGNAL        SYNTHESIZED_WIRE_2 :  STD_LOGIC;

BEGIN
Q <= JKFF_inst9;
SYNTHESIZED_WIRE_2 <= '1';
Qprime <= NOT(JKFF_inst9);
PROCESS(CLK,SYNTHESIZED_WIRE_2,SYNTHESIZED_WIRE_2)
VARIABLE synthesized_var_for_JKFF_inst9 : STD_LOGIC;

BEGIN
IF (SYNTHESIZED_WIRE_2 = '0') THEN
        synthesized_var_for_JKFF_inst9 := '0';
ELSIF (SYNTHESIZED_WIRE_2 = '0') THEN
        synthesized_var_for_JKFF_inst9 := '1';
ELSIF (RISING_EDGE(CLK)) THEN
        synthesized_var_for_JKFF_inst9 := (NOT(synthesized_var_for_JKFF_inst9) AND J) OR
(synthesized_var_for_JKFF_inst9 AND (NOT(K)));
END IF;
        JKFF_inst9 <= synthesized_var_for_JKFF_inst9;
END PROCESS;


END bdf_type;
```

Section above shows the VHDL code of the SR flip flop, JK flip flop using logic circuit and JK flip flop using provided block that is generated using Intel Quartus Prime software. First and foremost, for the VHDL code of SR flip flop, the code started with the library declaration at line “LIBRARY ieee;” and importing the “std_logic_1164” package at line “USE ieee.std_logic_1164.all;”. This will allow the use of “STD_LOGIC” type and its functions. Then the line “LIBRARY work;” is the default library where the current entity and architecture is placed.

Then the code started with the entity declaration section at line “ENTITY SRFlipFlop IS” continue with “PORT” line. The “SRFlipFlop” entity is declared with a total of 5 ports including 3 input ports and 2 output ports. Input ports included “S”, “R” and “CLK” where “S” and “R” corresponded to the S and R input of a SR flip flop while “CLK” corresponded to the clock input. Output ports included “Q” and its complementary “Qprime”. For an example in line “S: IN STD_LOGIC;”, “S” is declared as an input port with the type of “STD_LOGIC” to store logical values and to be used for intermediate computation. While for line “Q: OUT STD_LOGIC;”, “Q” is also declared as “STD_LOGIC” type but as output. The whole entity section fulfilled the characteristics of a SR flip flop.

Then the code moved into the architecture section at line “ARCHITECTURE bdf_type OF SRFlipFlop IS;”. This indicated that the architecture is named as “bdf_type” and is corresponded to the “SRFlipFlop” entity. Then the code will move on to the signal declaration section and the behavioural section which is also part of the architecture section.

Moving on to the signal declaration, a total of 4 signals “SYNTHESIZED_WIRE_0”, “SYNTHESIZED_WIRE_1”, “SYNTHESIZED_WIRE_2” and “SYNTHESIZED_WIRE_3” were declared to hold logical values and to be used for intermediate computations. For example, at line “SIGNAL SYNTHESIZED_WIRE_0: STD_LOGIC;”, keyword “SIGNAL” is used to declare the signal which is named as “SYNTHESIZED_WIRE_0” as “STD_LOGIC” type. They were intermediate wire signals that is used in the architecture.

Moreover, the code proceeded to the behavioural section where it describes the behaviour of the SR flip flop. The behavioural section started at line “BEGIN”. All the behaviour of the SR flip flop is described in concurrent signal assignment statements which means that all the statement will run at the same time simultaneously. At line “Q <= SYNTHESIZED_WIRE_3;”, “Q” output is assigned the value of “SYNTHESIZED_WIRE_3”, which represented the stored value of the

SR flip flop. For line “Qprime <= SYNTHESIZED_WIRE_0;” the “Qprime” output is assigned with the value of “SYNTHESIZED_WIRE_0” which represented the complement value of the stored value in the SR flip flop. Line “SYNTHESIZED_WIRE_2 <= NOT (R AND CLK);” indicated that “SYNTHESIZED_WIRE_2” is assigned the complement or inverse of the result of AND between “R” and “CLK”. At line “SYNTHESIZED_WIRE_1 <= NOT (CLK AND S);”, “SYNTHESIZED_WIRE_1” is assigned the complement of the logical AND between “CLK” and “S”. While at line “SYNTHESIZED_WIRE_3 <= NOT (SYNTHESIZED_WIRE_0 AND SYNTHESIZED_WIRE_1);”, “SYNTHESIZED_WIRE_3” is assigned the complement of the product of AND between “SYNTHESIZED_WIRE_0” and “SYNTHESIZED_WIRE_1”. Lastly, at line, “SYNTHESIZED_WIRE_0 <= NOT (SYNTHESIZED_WIRE_2 AND SYNTHESIZED_WIRE_3);”, the complement of the AND product between “SYNTHESIZED_WIRE_2” and “SYNTHESIZED_WIRE_3” is assigned to “SYNTHESIZED_WIRE_0”. Then the whole architecture section ended at line “END bdf_type”. These codes fully described the behaviour of the SR flip flop which acted like gated SR latch in Figure 49. The line that assigned the complement values of the AND product indicated the NAND gate used in the circuit diagram.

While for the VHDL code for the JK flip flop complied using logic gates, the code structure was similar to what is in SR flip flop just with different logic, behaviour and signal names. The code also started with library declaration and importing package to allow the use of standard logic types and functions from “IEEE” library.

Then the code also moved on to the entity declaration section to declare all the input output ports in the whole circuit. There was a total of 5 ports that included 3 input ports “J”, “K”, and “CLK” and 2 output ports “Q” and “Qprime”. The “J” and “K” corresponded to the J and K input to the JK flip flop and “CLK: is the input port for clock signal. “Q” is the output of the JK flip flop and “Qprime” is its complementary. All the input and output ports were also declared with “STD_LOGIC” type to store logical values to be used for intermediate computations.

Then the architecture section started at line “ARCHITECTURE bdf_type OF JKFlipFlop IS” which indicated that the architecture name is “bdf_type” and is corresponded to the “JKFlipFlop” entity.

Furthermore, the code moved on to the signal declaration section in the architecture section. A total of 4 signals were declared in this section which included "SYNTHESIZED_WIRE_6, "SYNTHESIZED_WIRE_7", "SYNTHESIZED_WIRE_3" and "SYNTHESIZED_WIRE_4". All the signals were also declared in "STD_LOGIC" type to store logical values to carry out intermediate computations.

The behavioural section started at line "BEGIN" where it described the behaviour of the JK flip flop circuit in Figure 61. The behaviour was also described in concurrent signal assignment statements. Firstly, at line "Q <= SYNTHESIZED_WIRE_7;", output "Q" is assigned the value of "SYNTHESIZED_WIRE_7;", which is the stored value of the JK flip-flop. While in line "Qprime <= SYNTHESIZED_WIRE_6;", the "Qprime" output is assigned the value of "SYNTHESIZED_WIRE_6", which represented the complement of the stored value in the JK flip flop. At line "SYNTHESIZED_WIRE_3 <= NOT (SYNTHESIZED_WIRE_6 AND J AND CLK);", "SYNTHESIZED_WIRE_3" is assigned the complement of AND product between "SYNTHESIZED_WIRE_6", "J", and "CLK". Then in line "SYNTHESIZED_WIRE_4 <= NOT (CLK AND K AND SYNTHESIZED_WIRE_7);", "SYNTHESIZED_WIRE_4" is assigned the complement of the AND product between "CLK", "K", and "SYNTHESIZED_WIRE_7". Whereas in line "SYNTHESIZED_WIRE_7 <= NOT (SYNTHESIZED_WIRE_6 AND SYNTHESIZED_WIRE_3);", "SYNTHESIZED_WIRE_7" is assigned the complement of the product of AND between "SYNTHESIZED_WIRE_6" and "SYNTHESIZED_WIRE_3". Lastly for line "SYNTHESIZED_WIRE_6 <= NOT (SYNTHESIZED_WIRE_4 AND SYNTHESIZED_WIRE_7);", "SYNTHESIZED_WIRE_6" is assigned the complement of the AND product between "SYNTHESIZED_WIRE_4" and "SYNTHESIZED_WIRE_7". The assignment of the complement of AND product indicated the use of NAND gates in the logic circuit. These VHDL codes fully described the behaviour of the JK flip flop circuit in Figure 61.

Finally, the code for the JK flip flop that used the block provided is different with the code for SR flip flop and the JK flip flop that used logic gates. First and foremost, the code also started with library declarations at line "LIBRARY ieee;" to include the "IEEE" library. Then the "std_logic_1164" package from the "IEEE" library is imported. The "std_logic_1164" package provide "STD_LOGIC" type and its related operations and functions.

Then the code also proceeded to the entity declaration section at line “ENTITY JKFlipFlopBlock IS” where the name of the entity is “JKFlipFlopBlock”. Then the line “PORT” indicated the start of the declaration process. A total of 5 ports which included 3 input ports “J”, “K”, “CLK” and 2 output ports “Q” and its complementary “Qprime”. “J” and “K” represented the J and K inputs for the JK flip flop while “CLK” represented the clock input to the JK flip flop. All the ports were declared in “STD_LOGIC” type to store logical values for intermediate computations.

Then the code continued to the architecture section at line “ARCHITECTURE bdf_type OF JKFlipFlopBlock IS”. The first section in the architecture section is the signal declaration section where signals were declared. There was a total of 2 signals declared which were “JKFF_inst9” which is used to represent the internal state of the JK flip flop. While “SIGNAL SYNTHESIZED_WIRE_2” is also declared as an intermediate wire signal used in the architecture.

Moreover, the code proceeded to the behavioural section at line “BEGIN”. BEGIN: This line marks the beginning of the architecture body. Line “Q <= JKFF_inst9;” assigned the value of “JKFF_inst9” to the output signal Q. While for line “SYNTHESIZED_WIRE_2 <= '1';”, it assigned the logical high value “1” to signal “SYNTHESIZED_WIRE_2”. At line “Qprime <= NOT (JKFF_inst9);” the complement of “JKFF_inst9” is assigned to the output signal “Qprime”. Whereas for line “PROCESS (CLK, SYNTHESIZED_WIRE_2, SYNTHESIZED_WIRE_2)” it defines a process block that is sensitive to changes in “CLK”, “SYNTHESIZED_WIRE_2”, and “SYNTHESIZED_WIRE_2”. “PROCESS” is a VHDL construct used to define a process, which represents a concurrent block of code that executes sequentially. “CLK”, “SYNTHESIZED_WIRE_2” and “SYNTHESIZED_WIRE_2” is the sensitivity list specified in parentheses after the PROCESS keyword. It defines the signals that the process is sensitive to, meaning that the process will be triggered and executed whenever there is a change in any of these signals. Then for line “VARIABLE synthesized_var_for_JKFF_inst9: STD_LOGIC;” it declared a variable named “synthesized_var_for_JKFF_inst9” of type “STD_LOGIC” within the process block. It represented a temporary variable used for computations.

Then the process block body started at line “BEGIN”. “IF (SYNTHESIZED_WIRE_2 = '0') THEN” this line starts an if statement that checks if “SYNTHESIZED_WIRE_2” is equal to '0'. Then line “synthesized_var_for_JKFF_inst9: = '0';” assigned the value '0' to “synthesized_var_for_JKFF_inst9” if the condition in the if statement is true. While line “ELSIF (SYNTHESIZED_WIRE_2 = '0') THEN” started an ELSEIF statement that checks if “SYNTHESIZED_WIRE_2” is equal to '0'. Then “synthesized_var_for_JKFF_inst9: = '1';” line assigned the value '1' to “synthesized_var_for_JKFF_inst9” if the condition in the ELSEIF statement is true. While “ELSIF (RISING_EDGE(CLK)) THEN” line starts an ELSEIF statement again that checks if there is a rising edge on the “CLK” signal. Then “synthesized_var_for_JKFF_inst9: = (NOT (synthesized_var_for_JKFF_inst9) AND J) OR (synthesized_var_for_JKFF_inst9 AND (NOT (K)));” line assigned the result of the JK flip flop operation to “synthesized_var_for_JKFF_inst9”. It toggles the value of “synthesized_var_for_JKFF_inst9” based on the inputs J and K when there is a rising edge on the clock signal. Then the conditional statement ended. Line “JKFF_inst9 <= synthesized_var_for_JKFF_inst9;” assigned the value of “synthesized_var_for_JKFF_inst9” to the internal signal “JKFF_inst9”. This line is also a sequential statement where it described the flow of data and control within a process or architecture. They executed in sequence and are sensitive to the current and previous values of signals or variables. In this case, the sequential statement updates the internal state of the JK flip-flop based on the computed value “synthesized_var_for_JKFF_inst9”.

In short, the whole process started with if else conditional statement to determine the value of "synthesized_var_for_JKFF_inst9". The value of "synthesized_var_for_JKFF_inst9" based on the values of “SYNTHESIZED_WIRE_2” and the rising edge of “CLK”. If “SYNTHESIZED_WIRE_2” is “0”, the variable is assigned “0”. Else if “SYNTHESIZED_WIRE_2” is “0”, the variable is assigned “1”. If there is a rising edge on “CLK”, the variable is assigned the result of the JK flip-flop operation, where “J” and “K” inputs are used to toggle the output value. The output of the JK flip flop block, “JKFF_inst9”, is assigned the value of "synthesized_var_for_JKFF_inst9". The “Q” output is connected to “JKFF_inst9”, and “Qprime” is assigned the complement of “JKFF_inst9”. Even the logic is weird, but it is still logically correct as it produced the output waveform that is intended and correct for a JK flip flop. The reason of having non readable code is because the code is generated by the software itself. Further declaration can be found in Figure 95 in the appendix.

4-Bit Binary to Gray Code Converter

Procedure 4-Bit Binary to Gray Code Converter

1. The 4-Bit Binary to Gray Code Converter circuit diagram were drawn using Quartus Prime block diagram.
2. The block diagram was compiled to check if there is any error.
3. The University Program VWF file is created.
4. The inputs were given.
5. The simulation was started to generate the waveforms.
6. The computer was connected to a FPGA.
7. All the inputs and outputs were assigned to pins in the FPGA.
8. The circuit is verified on the FPGA.
9. The HDL code was generated.

Schematic – 4-Bit Binary to Gray Code Converter

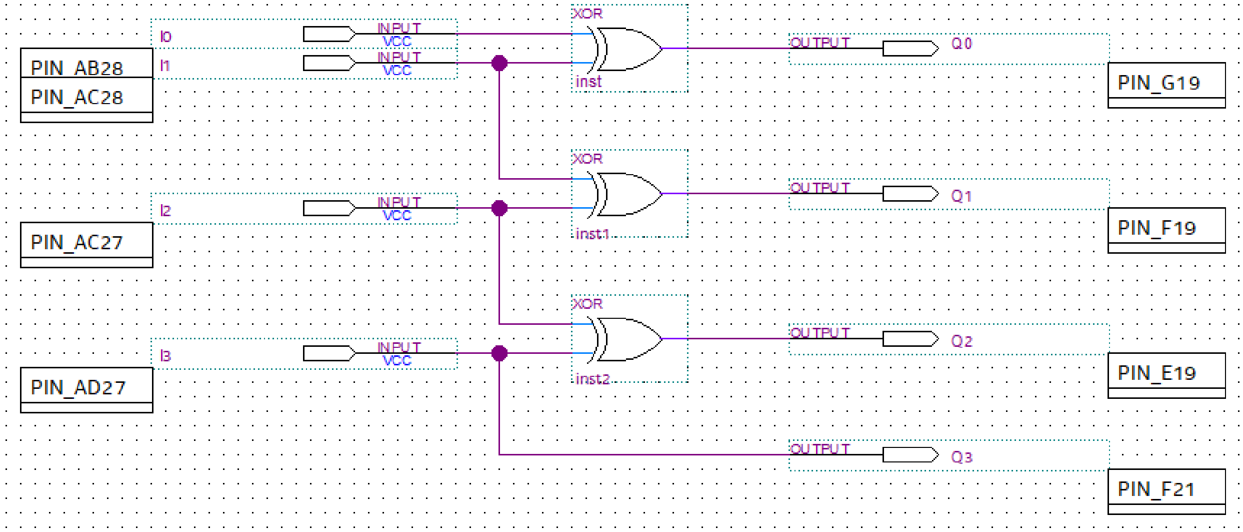


Figure 83: 4-Bit Binary to Gray Code Converter Circuit Diagram

Output Signals (Waveform) – 4-Bit Binary to Gray Code Converter

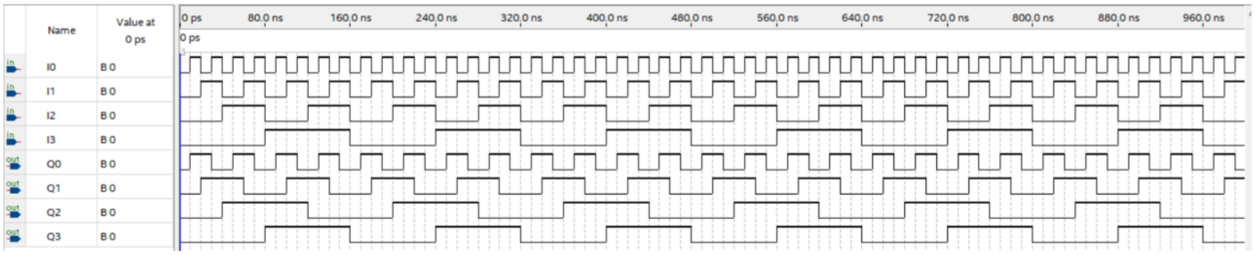
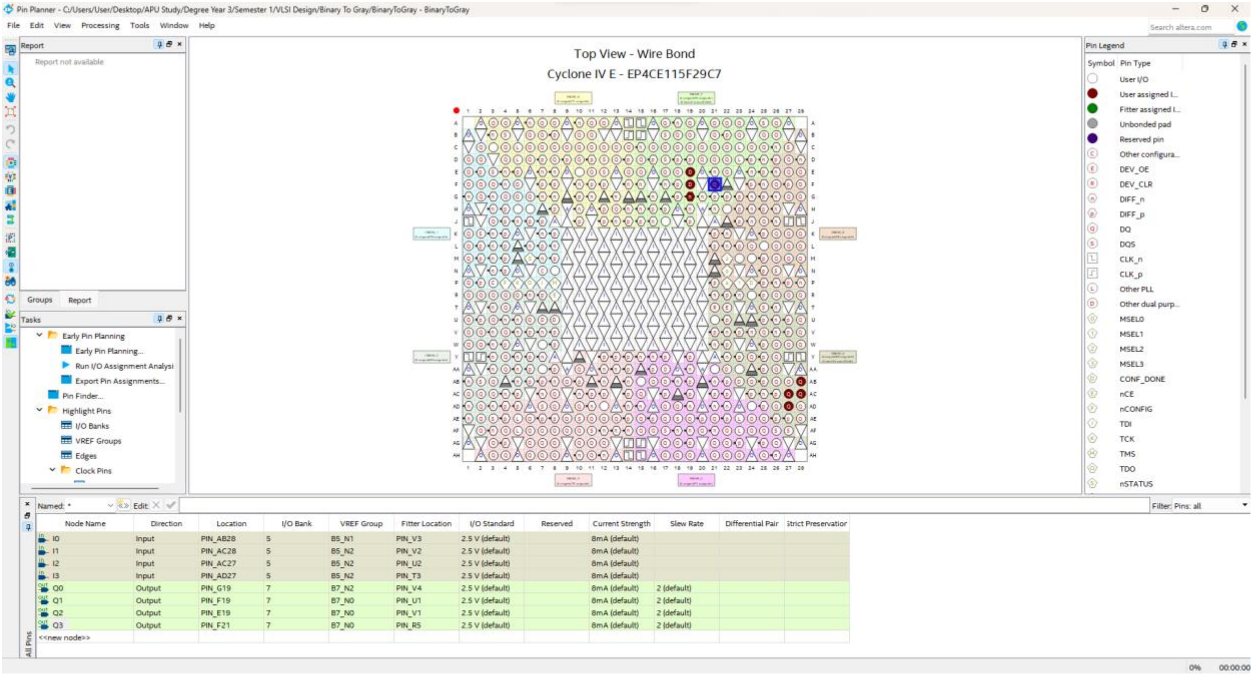


Figure 84: 4-Bit Binary to Gray Code Converter Output Waveform

Pin Planner – 4-Bit Binary to Gray Code Converter



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
I0	Input	PIN_AB28	5	B5_N1	PIN_V3	2.5 V (default)		8mA (default)			
I1	Input	PIN_AC28	5	B5_N2	PIN_V2	2.5 V (default)		8mA (default)			
I2	Input	PIN_AC27	5	B5_N2	PIN_U2	2.5 V (default)		8mA (default)			
I3	Input	PIN_AD27	5	B5_N2	PIN_T3	2.5 V (default)		8mA (default)			
Q0	Output	PIN_G19	7	B7_N2	PIN_V4	2.5 V (default)		8mA (default)	2 (default)		
Q1	Output	PIN_F19	7	B7_N0	PIN_U1	2.5 V (default)		8mA (default)	2 (default)		
Q2	Output	PIN_E19	7	B7_N0	PIN_V1	2.5 V (default)		8mA (default)	2 (default)		
Q3	Output	PIN_F21	7	B7_N0	PIN_R5	2.5 V (default)		8mA (default)	2 (default)		
<<new node>>											

Figure 85: 4-Bit Binary to Gray Code Converter Pin Planner

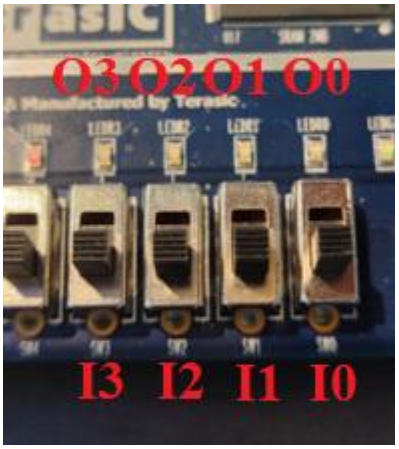


Figure 86: 4-Bit Binary to Gray Code Converter FPGA Pins

Truth Table – 4-Bit Binary to Gray Code Converter

Table 14: 4-Bit Binary to Gray Code Converter Truth Table

Binary				Gray Code			
I3	I2	I1	I0	Q3	Q2	Q1	Q0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Results from FPGA Device – 4-Bit Binary to Gray Code Converter

Table 15: 4-Bit Binary to Gray Code Converter FPGA Results

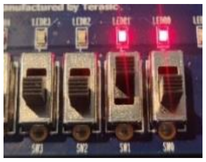
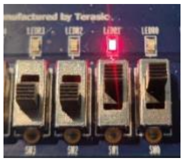
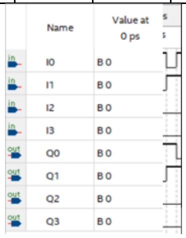
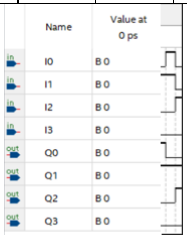
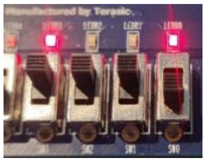
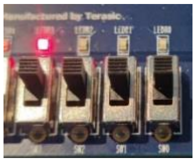
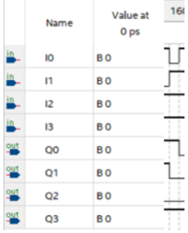
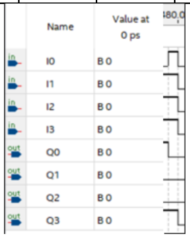
<table>
<tr><td colspan="8">
Figure 87: 4-Bit Binary to Gray Code Converter FPGA Result 1</td><td colspan="8">
Figure 88: 4-Bit Binary to Gray Code Converter FPGA Result 2</td></tr>
<tr><td>I3</td><td>I2</td><td>I1</td><td>I0</td><td>O3</td><td>O2</td><td>O1</td><td>O0</td><td>I3</td><td>I2</td><td>I1</td><td>I0</td><td>O3</td><td>O2</td><td>O1</td><td>O0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">
Name Value at 0 ps
I0 B 0
I1 B 0
I2 B 0
I3 B 0
Q0 B 0
Q1 B 0
Q2 B 0
Q3 B 0
Figure 89: 4-Bit Binary to Gray Code Converter FPGA Result Waveform 1</td><td colspan="8">
Name Value at 0 ps
I0 B 0
I1 B 0
I2 B 0
I3 B 0
Q0 B 0
Q1 B 0
Q2 B 0
Q3 B 0
Figure 90: 4-Bit Binary to Gray Code Converter FPGA Result Waveform 2</td></tr>
<tr><td colspan="8">
Figure 91: 4-Bit Binary to Gray Code Converter FPGA Result 3</td><td colspan="8">
Figure 92: 4-Bit Binary to Gray Code Converter FPGA Result 4</td></tr>
<tr><td>I3</td><td>I2</td><td>I1</td><td>I0</td><td>O3</td><td>O2</td><td>O1</td><td>O0</td><td>I3</td><td>I2</td><td>I1</td><td>I0</td><td>O3</td><td>O2</td><td>O1</td><td>O0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="8">
Name Value at 0 ps
I0 B 0
I1 B 0
I2 B 0
I3 B 0
Q0 B 0
Q1 B 0
Q2 B 0
Q3 B 0
Figure 93: 4-Bit Binary to Gray Code Converter FPGA Result Waveform 3</td><td colspan="8">
Name Value at 0 ps
I0 B 0
I1 B 0
I2 B 0
I3 B 0
Q0 B 0
Q1 B 0
Q2 B 0
Q3 B 0
Figure 94: 4-Bit Binary to Gray Code Converter FPGA Result Waveform 4</td></tr>
</table>

Figure 83 shows the logic circuit diagram of 4-bit binary to gray code converter. The circuit consisted of 4 inputs I0, I1, I2, I3 and 4 outputs O0, O1, O2, O3. The whole circuit contained 3 XOR gates with 2 inputs. Figure 84 displayed the waveform generated when count inputs were given. Every input will be given count input with the count time double of their previous one to make the 4 inputs as a 4-bit binary counter. Hence the count time for I0 is 10ns, I1 is 20ns, I2 is 40ns and I3 is 80ns.

Figure 85 shows the pin planner which were assigned for each input and output in the circuit. Figure 86 displayed the physical locations of the input and output pins on the FPGA when it is connected. For the inputs, input I0 is SW0, I1 is SW1, I2 is SW2 and I3 is SW3. While for the output O0 is LEDR0, O1 is LEDR1, O2, is LEDR2 and O3 is LEDR3.

A 4-bit binary to gray code converter is a combinational circuit that takes a 4-bit binary input and produces a 4-bit gray code output. Gray code is a binary numeral system where adjacent binary numbers differ by only one bit (Tutorials Point, 2020). As an example, 2 in 4-bit binary code is 0010 while is 0011 in gray code. Table 14 shows the truth table of the 4-bit binary to gray code converter.

Table 15 displays 4 examples of the result obtained from FPGA also with their corresponded output waveform attached below. The output on the FPGA with different input are shown in Figure 87, Figure 88, Figure 91 and Figure 92 and their corresponding waveforms are shown in Figure 89, Figure 90, Figure 93 and Figure 94. All the results obtained through FPGA and the generated waveform were verified with the truth table in Table 14. This indicated that the logic circuit is correct. As an example, in Figure 87 the input in I1 which is SW1 is high while the other inputs were low which indicating the binary of 0010 which is 2 in decimal. While the output is high at O0 and O1 which corresponded to LEDR0 and LEDR1 while output O2 and O3 which corresponded to LEDR2 and LEDR3 are low. This indicated the gray code 0011 which is also 2 in decimal. This verified that the results obtained from the FPGA are correct.

Discussion – 4-Bit Binary to Gray Code Converter

4-Bit Binary to Gray Code Converter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY BinaryToGray IS
        PORT
        (
                I1 :  IN  STD_LOGIC;
                I0 :  IN  STD_LOGIC;
                I2 :  IN  STD_LOGIC;
                I3 :  IN  STD_LOGIC;
                Q0 :  OUT  STD_LOGIC;
                Q1 :  OUT  STD_LOGIC;
                Q2 :  OUT  STD_LOGIC;
                Q3 :  OUT  STD_LOGIC
        );
END BinaryToGray;

ARCHITECTURE bdf_type OF BinaryToGray IS

BEGIN

Q3 <= I3;
Q0 <= I0 XOR I1;
Q1 <= I1 XOR I2;
Q2 <= I2 XOR I3;

END bdf_type;
```

Section above shows the VHDL code of the 4-bit binary to gray code converter that generated by Intel Quartus Prime software. The code started with the declaration of library at line “LIBRARY ieee;”. Then line “USE ieee.std_logic_1164.all;” imported the “std_logic_1164” package that provided “STD_LOGIC” type and its functions and operations.

Then the code started with entity declaration section where the input and output ports in the circuit were declared. According to line “ENTITY BinaryToGray IS” the entity is named as “BinaryToGray” and the line “PORT” indicated the start of the declaration process. There was a total of 8 ports including 4 input ports and 4 output ports. 4 of the input ports were “I0”, “I1”, “I2” and “I3” which represented the 4-bit binary input while the 4 output ports were “Q0”, “Q1”, “Q2” and “Q3” represented the 4-bit gray code output. As an example, for line “I0: IN STD_LOGIC;”, “I0” is declared as an input with the keyword “IN” and as “STD_LOGIC” type to allow it to store logical values for intermediate computations. While line “Q0: OUT STD_LOGIC;” is declaring “Q0” as output using the “OUT” keyword and as “STD_LOGIC” type. The whole entity corresponded to the circuit diagram where it also contained 4 input ports and 4 output ports which is the amount of input output ports that a 4-bit binary to gray code converter should have.

Then the code proceeded to the architecture section at line “ARCHITECTURE bdf_type OF BinaryToGray ID”. This line indicated that the architecture is named as “bdf_type” and is corresponded to the “BinaryToGray” entity. In this case the architecture didn’t contain the signal declaration section because there were no intermediate wire signals needed in this logic circuit to be used in the architecture.

Furthermore, the code started the behavioural section where it described the behaviour of the 4-bit binary to gray code converter circuit. The behaviour was also described with concurrent signal assignment statement where all the statement was running simultaneously. Firstly, at line “Q3 <= I3”, output “Q3” is directly assigned with the input “I3”. For line “Q0 <= I0 XOR I1;” output “Q0” is assigned with the result of XOR operation between inputs “I0” and “I1”. Then for line “Q1 <= I1 XOR I2;”, output “Q1” is assigned with the result of XOR between “I1” and “I2”. Lastly for line “Q2 <= I2 XOR I3;”, the result of XOR between “I2” and “I3” is assigned to output “Q2”. Then the behavioural section ended at line “END bdf_type”. The code fully described the circuit diagram in Figure 83.

Conclusion

In a nutshell, this lab report aimed to analyse various digital logic circuits using VHDL code. The circuits investigated included an 8-to-1 multiplexer, a 1-to-8 demultiplexer, a full adder, a full subtractor, an SR flip flop, a JK flip flop, and a 4-bit binary to gray code converter. All the objectives of compiling the combinational and sequential logic above by utilizing HDL codes were achieved and the codes were discussed and analysed in detail in this report.

Through VHDL code implementation and analysis, the behaviour and functionality of each circuit is understood. By studying their architectures and the interconnections between inputs and outputs, insights into the underlying logic and operations performed by these circuits are gained.

8 to 1 multiplexer and 1 to 8 demultiplexer are combinational circuits. The 8 to 1 multiplexer allowed for the selection of a single input from eight possible inputs based on control signals. The 1 to 8 demultiplexer, on the other hand, distributed a single input to one of eight output lines based on control signals.

Full adder and full subtractor are also combinational circuits. The full adder circuit combined two binary inputs and a carry in to produce a sum output and a carry out. Conversely, the full subtractor circuit performed subtraction using two binary inputs and a borrow in to generate a difference output and a borrow out.

The SR flip flop provided sequential logic behaviour, with set and reset inputs controlling the state of its outputs according to the clock input. However, there were no edge detector integrated in the SR flip flop circuit which make the flip flop created acted like a gated SR latch where it is level triggered instead of edge triggered. While JK flip flop is another sequential circuit, utilized J and K inputs to determine the output state based on clock input. There were 2 ways of compiling the HDL code for JK flip flop in this report which were using the logic gates and the JK flip flop block provided by the software. Same as in SR flip flop the circuit that uses logic gates to compile the HDL code for JK flip flop also not integrated with the edge detector which also causes it to be level triggered and causing it to have malfunction in toggling. While for the HDL code that is compiled using the JK flip flop block there were some weird statements, but they were logically correct. Besides that, this is also the only HDL code that contains sequential statements in the whole report whereas all other HDL code in this report only contained concurrent statements.

Lastly, the 4-bit binary to gray code converter circuit converted a 4-bit binary input to the corresponding gray code output using XOR operations.

By analysing these circuits and HDL codes, the understanding of digital logic is deepened and valuable insights into their design and implementation were gained. HDL provided a powerful tool for modelling, describing and simulating these circuits, enabling users or designers to verify and describe the functionality and behaviour of the circuits.

Overall, this lab report allowed the process of exploring and analysing a variety of important digital logic circuits using HDL code, contributing to understanding digital systems in depth.

References

Intel. (n.d.). *Everything You Need to Design for Intel® FPGAs, SoCs, and CPLDs*. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

Geeks For Geeks. (2023). *Hardware Description Language.* https://www.geeksforgeeks.org/hardware-description-language/

Electronics Tutorials. (n.d.). *The Multiplexer.* https://www.electronics-tutorials.ws/combination/comb_2.html

Electronics Tutorials. (n.d.). *The Demultiplexer.* https://www.electronics-tutorials.ws/combination/comb_3.html

Geeks For Geeks. (2023). *Full Adder in Digital Logic.* https://www.geeksforgeeks.org/full-adder-in-digital-logic/

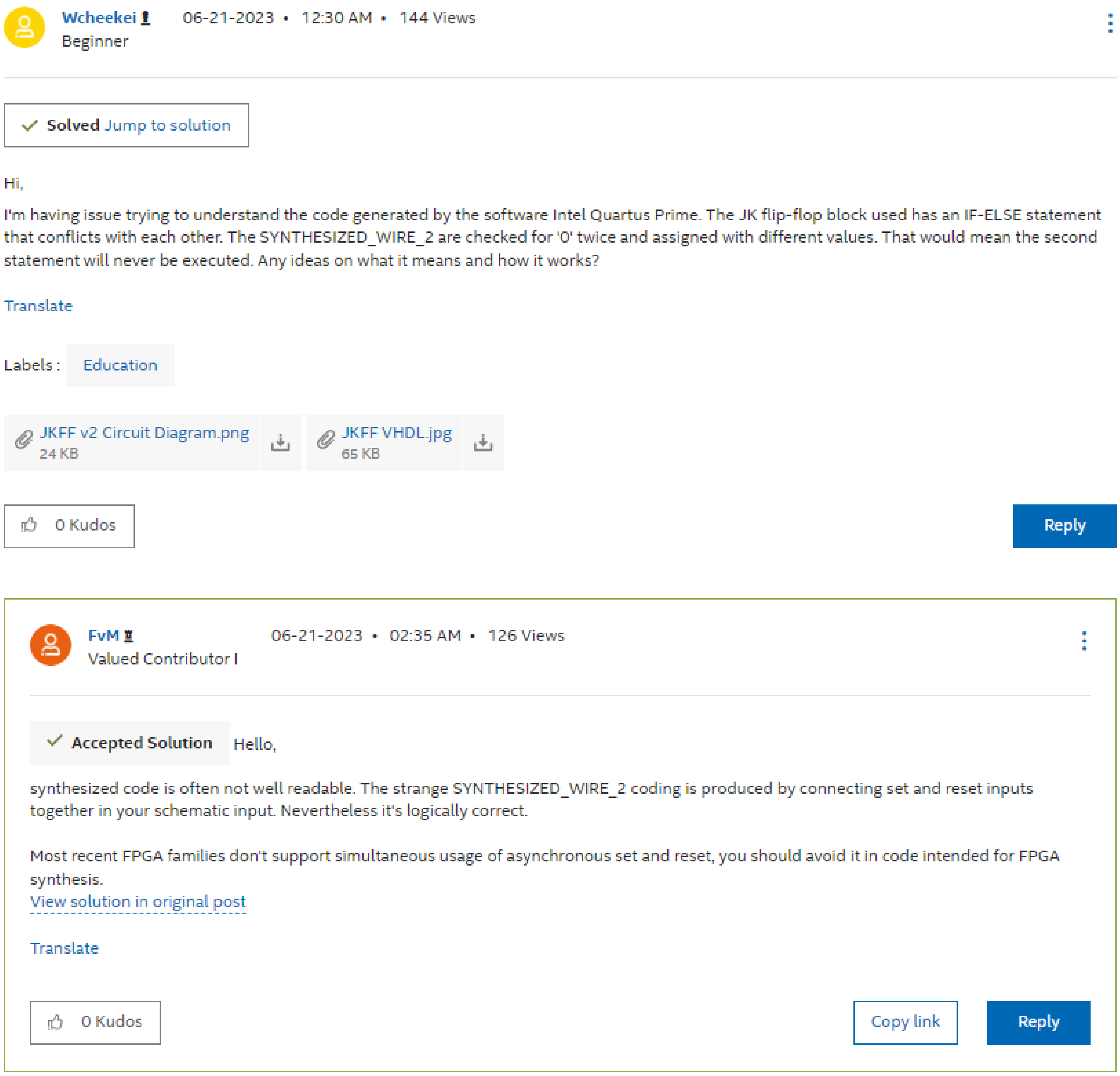
Geeks For Geeks. (2023). *Full Subtractor in Digital Logic.* https://www.geeksforgeeks.org/full-subtractor-in-digital-logic/

Geeks For Geeks. (2023). *SR Flip Flop.* https://www.geeksforgeeks.org/sr-flip-flop/

Geeks For Geeks. (2020). *JK Flip Flop and SR Flip Flop.* https://www.geeksforgeeks.org/jk-flip-flop-and-sr-flip-flop/

Tutorials Point. (2020). *What is Gray Code?* https://www.tutorialspoint.com/what-is-gray-code

Appendix



Wcheekei
Beginner
06-21-2023 • 12:30 AM • 144 Views

Hi,

I'm having issue trying to understand the code generated by the software Intel Quartus Prime. The JK flip-flop block used has an IF-ELSE statement that conflicts with each other. The SYNTHESIZED_WIRE_2 are checked for '0' twice and assigned with different values. That would mean the second statement will never be executed. Any ideas on what it means and how it works?

FvM
Valued Contributor I
06-21-2023 • 02:35 AM • 126 Views

Hello,

synthesized code is often not well readable. The strange SYNTHESIZED_WIRE_2 coding is produced by connecting set and reset inputs together in your schematic input. Nevertheless it's logically correct.

Most recent FPGA families don't support simultaneous usage of asynchronous set and reset, you should avoid it in code intended for FPGA synthesis.

Figure 95: Appendix 1