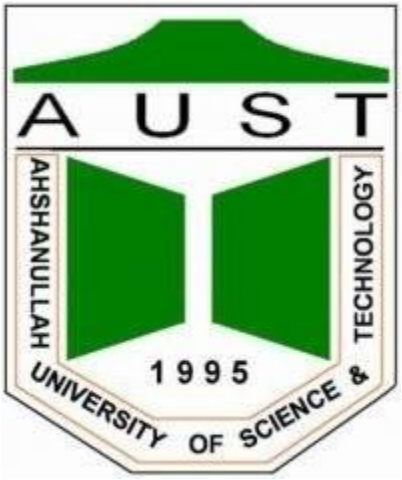


EEE – 2104
Digital Circuit Design Lab

AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT
OF
ELECTRICAL AND ELECTRONIC ENGINEERING

EEE- 2104

Digital Circuit Design Lab

Edited & Revised By
Sumit Narayan Saurov
EEE, AUST

Edition 2026

Table of Contents

Table of Contents 3

Experiment: 1 4

Experiment: 2 13

Experiment: 3 17

Experiment: 4 50

Experiment: 5 63

Experiment: 6 75

Experiment: 7 86

Experiment: 8 98

Experiment: 9 112

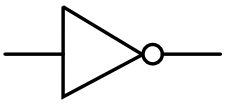
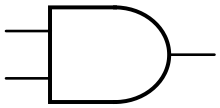
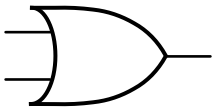
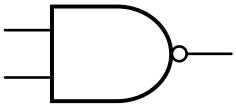
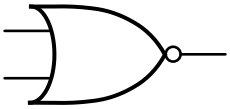
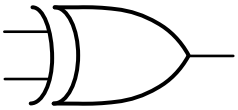
Experiment: 1

Experiment name: *Introduction to different digital ICs.*

Introduction:

In this experiment you will be introduced to different digital ICs that will be used in this digital lab to perform different functions and also the function of each IC. You are asked to memorize the following associated with each IC.

1. IC number
2. IC name
3. Total number of pins
4. V_{cc} pin number
5. Ground (GND) pin number

IC number	IC Name	Gate Name	Schematic
7404	Hex Inverter	NOT	
7408	Quad 2-input AND	AND	
7432	Quad 2-input OR	OR	
7400	Quad 2-input NAND	NAND	
7402	Quad 2-input NOR	NOR	
7486	Quad 2-input XOR	XOR	

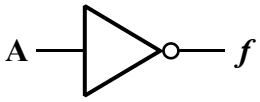
Logic Levels

In digital electronics, there are two distinct logic levels: **Logic 0** (also called Logic LOW or Logic FALSE) and **Logic 1** (also called Logic HIGH or Logic TRUE), which represent the two states used to process and store information. Logic 0 usually corresponds to a lower voltage level, while Logic 1 corresponds to a higher voltage level, with the exact values depending on the technology. In our system, Logic 0 corresponds to 0V and Logic 1 corresponds to 5V.

NOT Gate

A NOT gate has only one input and one output. Its job is to produce the exact opposite logic level of the input signal. If the input is logic 1 (high), the output becomes logic 0 (low), and if the input is logic 0, the output becomes logic 1. This gate is also known as INVERTER. This inversion is useful for generating complementary signals, controlling logic conditions, and creating more complex digital functions. It is essential in generating complementary signals needed in flip-flops, multiplexers, and memory circuits. This gate is also used in control logic to activate devices when an input is low and in address decoders for memory selection. Additionally, they form the basis of other logic gates like NAND and NOR and can be used in oscillator circuits or noise-filtering circuits such as Schmitt triggers, making them versatile components in both simple and complex digital systems.

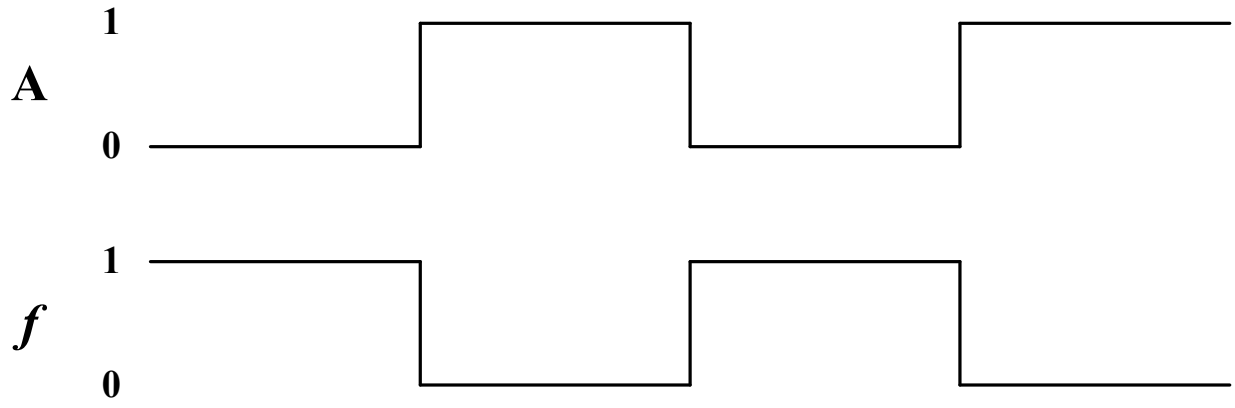
Its Symbol, Truth Table and Boolean Expression are shown as follows –



A	f
0	1
1	0

$$f = \overline{A}$$

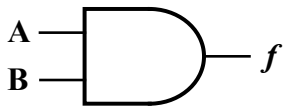
Waveform for the NOT gate is shown as follows –



From the waveform we can easily see output signal ***f*** is always inverse of the input signal **A**.

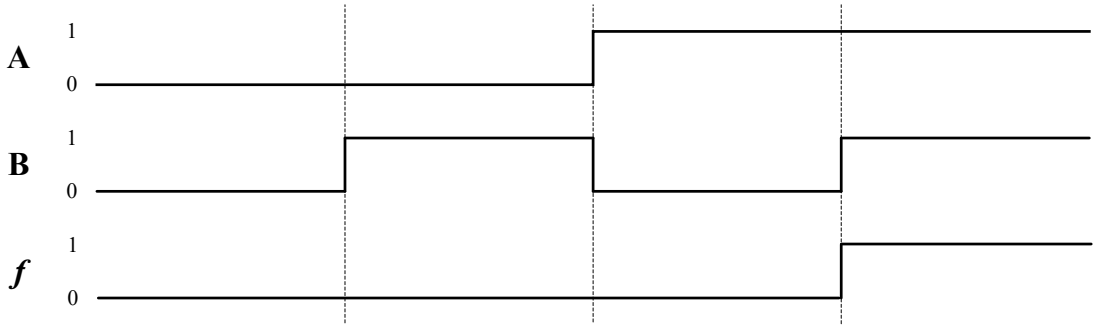
AND Gate

An AND gate outputs logic 1 only when all of its inputs are logic 1. If even one input is 0, the output becomes 0. This gate represents the logical AND operation used in mathematics and programming. It is often used in circuits where multiple conditions must be true at the same time – for example, enabling a device only when all required safety checks output a logic 1. AND gates can have two or more inputs depending on the circuit design.



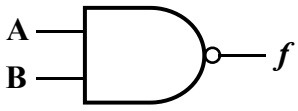
A	B	f
0	0	0
0	1	0
1	0	0
1	1	1

$$f = AB$$



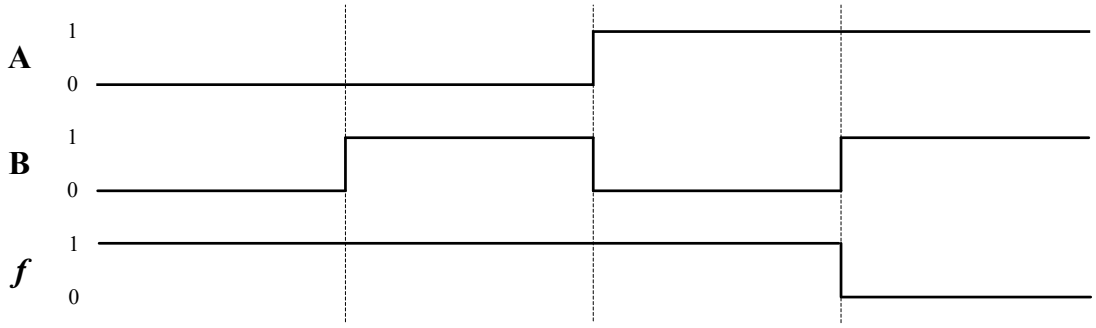
NAND Gate

The NAND gate is the inverted form of the AND gate. It produces logic 0 only when all of its inputs are 1; in every other case, it outputs 1. This behavior makes NAND extremely important – it is known as a universal gate, meaning you can build any digital circuit (including AND, OR, NOT, and even complex systems) using only NAND gates. NAND gates are widely used because they are easy to implement in CMOS technology and are efficient in real circuits.



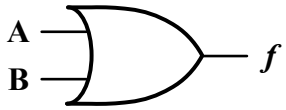
A	B	f
0	0	1
0	1	1
1	0	1
1	1	0

$$f = \overline{AB}$$



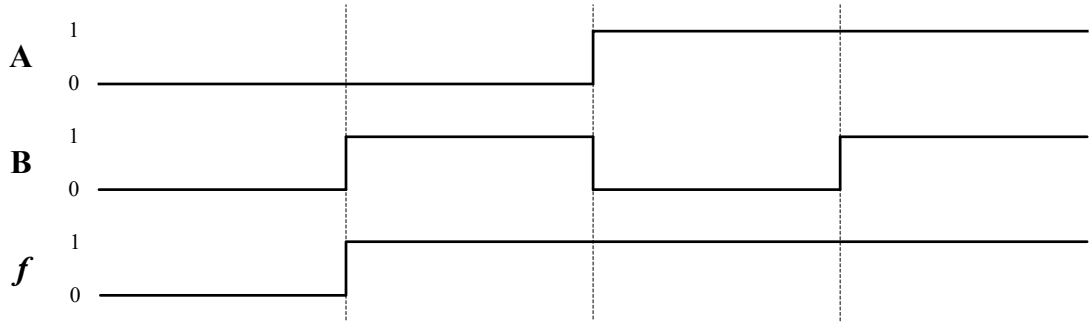
OR Gate

An OR gate outputs logic 1 if at least one of its inputs is 1. It produces logic 0 only when all inputs are 0. This gate reflects logical addition, meaning multiple possible conditions can trigger the output. OR gates are commonly used when a device should activate if any one of several signals is present, such as alarm systems where any triggered sensor should produce a high output.



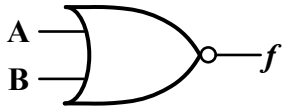
A	B	f
0	0	0
0	1	1
1	0	1
1	1	1

$$f = A + B$$



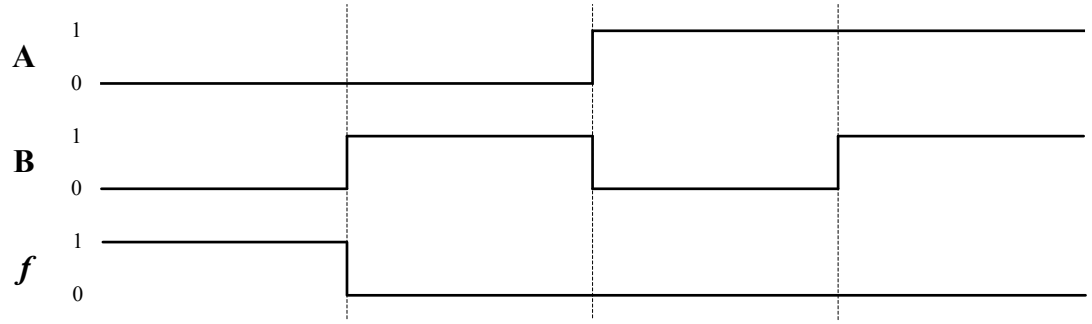
NOR Gate

A NOR gate is the inverse of the OR gate. It outputs logic 1 only when all inputs are 0, and gives 0 if any input is 1. NOR gates, like NAND gates, are universal gates, capable of forming any digital circuit when combined in certain ways. They are widely used in memory elements, control logic, and simple state machines because their inverted OR behavior makes them useful in decision-making circuits.



A	B	f
0	0	1
0	1	0
1	0	0
1	1	0

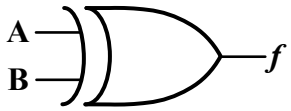
$$f = \overline{A + B}$$



XOR Gate

The XOR (Exclusive OR) gate is a digital logic gate that produces a high output based on the concept of **Odd Parity**. For a standard 2-input XOR gate, the output is 1 only when the two inputs are different – meaning one is 0 and the other is 1. However, for XOR gates with more than two inputs, the output becomes 1 when an ***odd number*** of inputs are 1, not simply when they are ***different***. This unique behavior makes XOR especially important in arithmetic circuits such as half adders and full adders, where it generates the sum bit. It is also widely used in comparators, parity generators, error detection systems, and encryption because XOR is effective at identifying differences and producing predictable transformations. Due to its ability to highlight differences, the XOR gate plays a crucial role in many signal-processing and decision-making circuits.

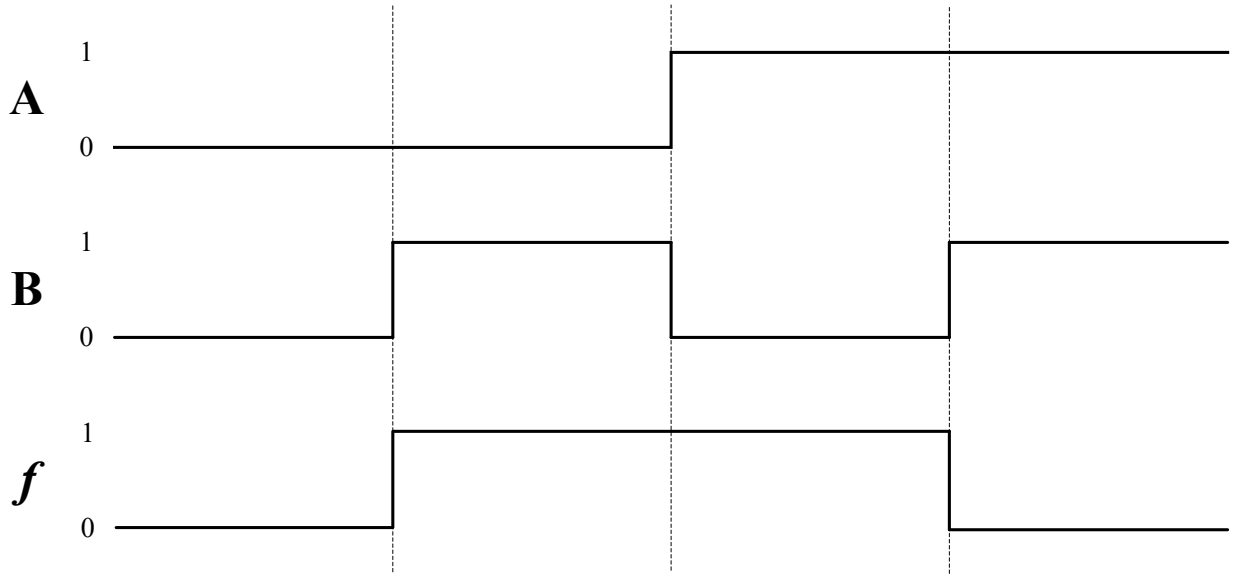
Its Symbol, Truth Table and Boolean Expression are shown as follows –



A	B	*f*
0	0	0
0	1	1
1	0	1
1	1	0

$$f = A \oplus B$$

Waveform for the XOR gate is shown as follows –



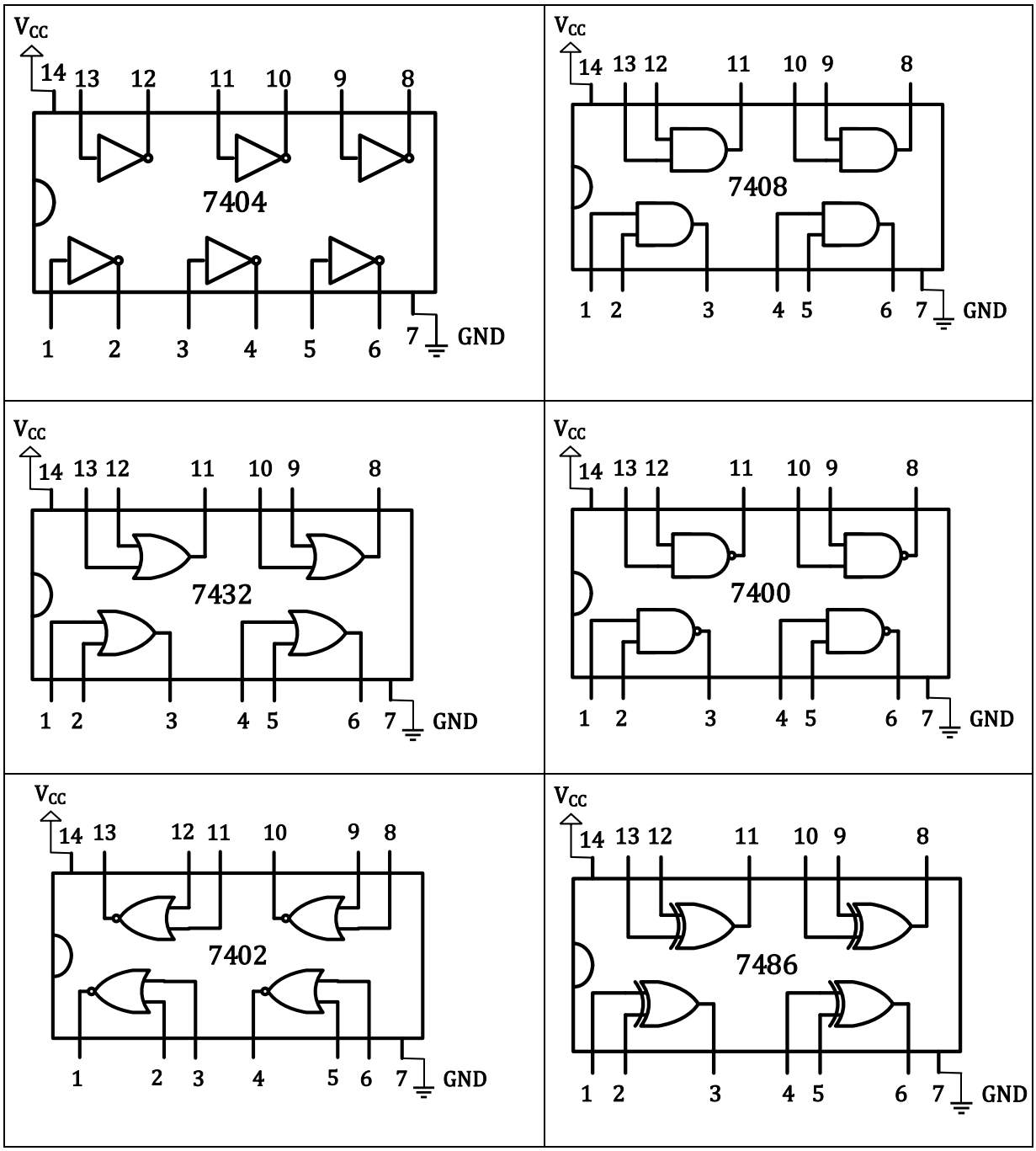
Equipment:

1. Trainer Board
2. IC 7400,7402,7404,7408,7432,7486
3. Microprocessor Data handbook

Procedure:

1. Take any of the following ICs. From microprocessor data handbook find the name of the IC, total number of pins that it has, V_{cc} pin and ground pin.

IC Number	IC name	Total pin	V_{cc} pin	GND pin
7400	NAND	14	14	7
7402	NOR	14	14	7
7404	NOT	14	14	7
7408	AND	14	14	7
7432	OR	14	14	7
7486	XOR	14	14	7

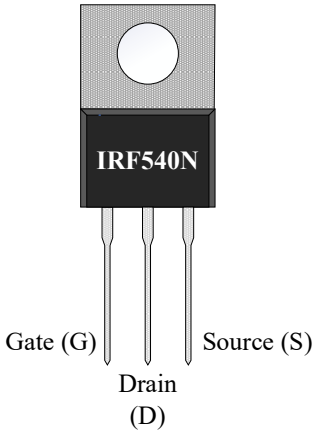


2. Now fill up the following table:

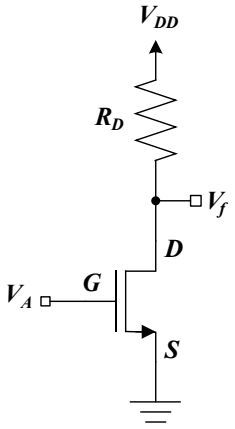
Input A	Input B	7404 NOT	7408 AND	7400 NAND	7432 OR	7402 NOR	7486 XOR
0	0						
0	1						
1	0						
1	1						

Implementation Technology:

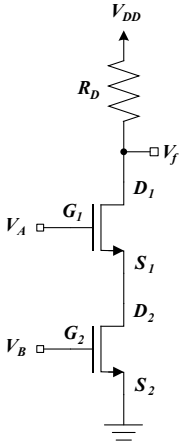
The earliest methods of constructing logic gates with MOSFETs gained popularity in the 1970s and were based on either PMOS or NMOS technology. In this section, we focus on NMOS circuits, which are built using only NMOS transistors. By applying the principle of transistor switching, we will explore how basic logic gates can be implemented using NMOS devices.



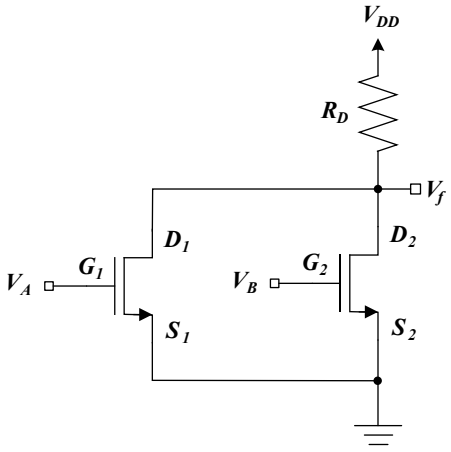
NMOS IC



NOT Gate



NAND Gate



NOR Gate

MOSFET as a Switch

An N-channel enhancement-type MOSFET acts as a voltage-controlled switch that operates by forming or removing a conductive channel between the drain and source terminals based on the gate-to-source voltage (V_{GS}). In its natural state, with $V_{GS} = 0$, there is no physical conducting channel between the drain and source because the substrate is p-type, and the two n+ regions (drain and source) are separated by this p-type material. As a result, the MOSFET remains OFF.

When a positive voltage is applied to the gate, it creates an electric field through the thin insulating oxide layer. This field repels holes (the majority carriers in the p-type substrate) away from the region beneath the gate and attracts electrons (minority carriers) toward it. As V_{GS} increases and exceeds the threshold voltage (V_{th}), enough electrons accumulate at the surface to form a thin n-type inversion layer, or channel, connecting the source and drain. This channel allows current to flow from drain to source when a drain-to-source voltage (V_{DS}) is applied.

NOT Gate

In the inverter circuit, a single NMOS transistor is connected with its drain to the supply voltage V_{DD} through a resistor R_D, the source to ground, and the input applied to its gate. When the input is LOW (0V), NMOS transistor is OFF, and the output is pulled HIGH (5V) through R_D. When the input is HIGH (5V), NMOS transistor turns ON, pulling the output LOW (0V), thus producing an inverted output.

NAND Gate

For a NAND gate, two NMOS transistors (MOS1 and MOS2) are connected in series between the output node and ground, with both their gates acting as the two inputs, A and B. The output node is connected to V_{DD} through a resistor. The circuit outputs HIGH when either A or B is LOW, because at least one transistor is OFF, preventing a path to ground. The output goes LOW only when both A and B are HIGH, turning both MOS1 and MOS2 ON and creating a conduction path to ground.

NOR Gate

In contrast, for a NOR gate, two NMOS transistors (MOS1 and MOS2) are connected in parallel between the output and ground, again with their gates as inputs A and B, and a resistor pulling up the output to V_{DD}. The output remains HIGH only when both A and B are LOW, since both transistors are OFF, breaking the path to ground. If either input is HIGH, the corresponding MOSFET turns ON, connecting the output to ground and producing a LOW output.

Report:

1. How can you make a three input AND/OR/XOR gate with a two input AND/OR/XOR gate?
2. Is it possible to make a three input NAND/NOR gate with two input NAND/NOR gate? Justify your answer.
3. Design AND & OR gate using N-MOSFET & Resistance.

Experiment: 2

Experiment name: *Introduction to Combinational Logic and K-map Minimization.*

Introduction:

Logic design basically means the construction of appropriate function, presented in Boolean algebraic form, then edification of the logic diagram, and finally choosing of available ICs and implementing the IC connection so that the logic intended is achieved. The efficiency in simplifying the algebra leads to less complicated logic diagram, which in the end leads to easier IC selection and easier circuit implementation.

Caution:

1. Remember to properly identify the pin numbers so that no accidents occur.
2. Properly bias the ICs appropriate voltages to appropriate pins.

Equipment:

1. Trainer Board
2. IC 7404,7408,7432

Job:

Implement of function –

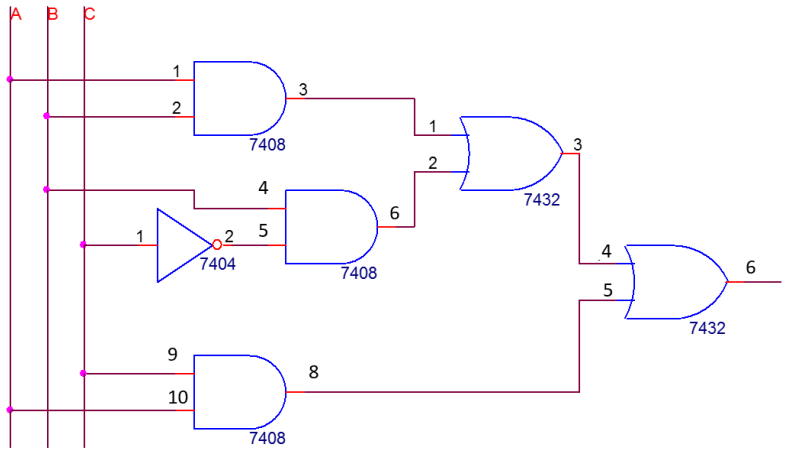
$$x.1 = x$$
$$x + x' = 1$$
$$x + x = x$$

$$\begin{aligned} f &= AB + BC' + CA \\ &= AB.1 + BC'.1 + CA.1 \\ &= AB(C + C') + BC'(A + A') + CA(B + B') \\ &= ABC + ABC' + ABC' + A'BC' + ABC + AB'C \\ &= ABC + ABC' + A'BC' + AB'C \\ &= A'BC' + AB'C + ABC' + ABC \\ &= m_2 + m_5 + m_6 + m_7 \\ &= \sum m(2,5,6,7) \end{aligned}$$

Truth Table:

Row	Minterm	Input			Output
		A	B	C	*f*
0	$m_0 = A' B' C'$	0	0	0	0
1	$m_1 = A' B' C$	0	0	1	0
2	$m_2 = A' B C'$	0	1	0	**1**
3	$m_3 = A' B C$	0	1	1	0
4	$m_4 = A B' C'$	1	0	0	0
5	$m_5 = A B' C$	1	0	1	**1**
6	$m_6 = A B C'$	1	1	0	**1**
7	$m_7 = A B C$	1	1	1	**1**

Circuit Diagram:



$$f = AB + BC' + CA$$

k-Map:

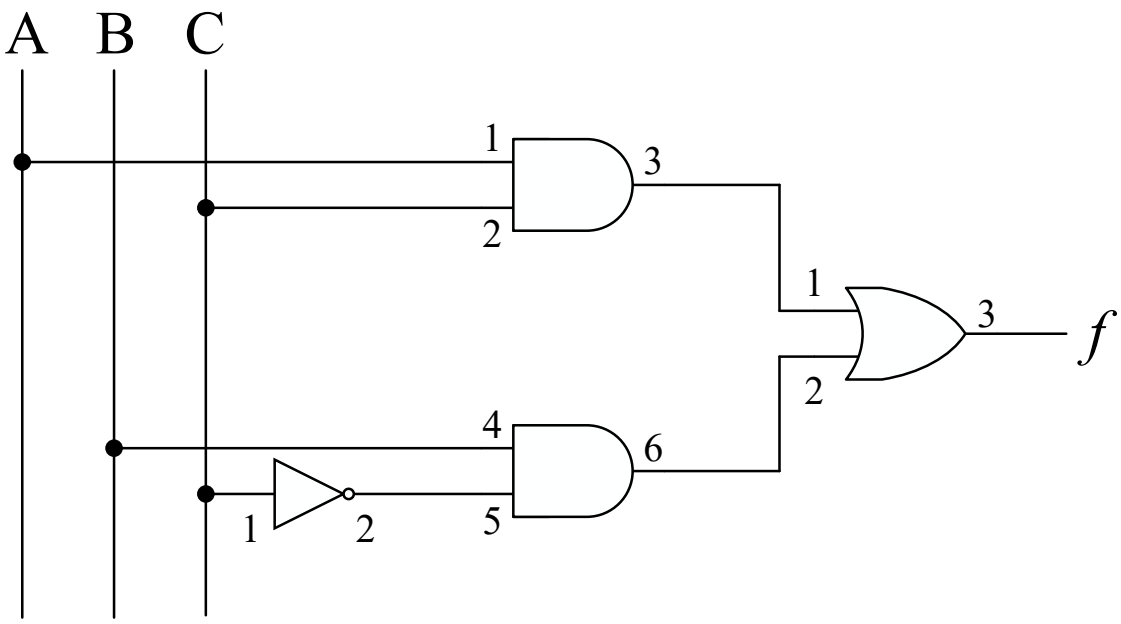
A \ BC	00	01	11	10
0	0	0	0	1
1	0	1	1	1

AC

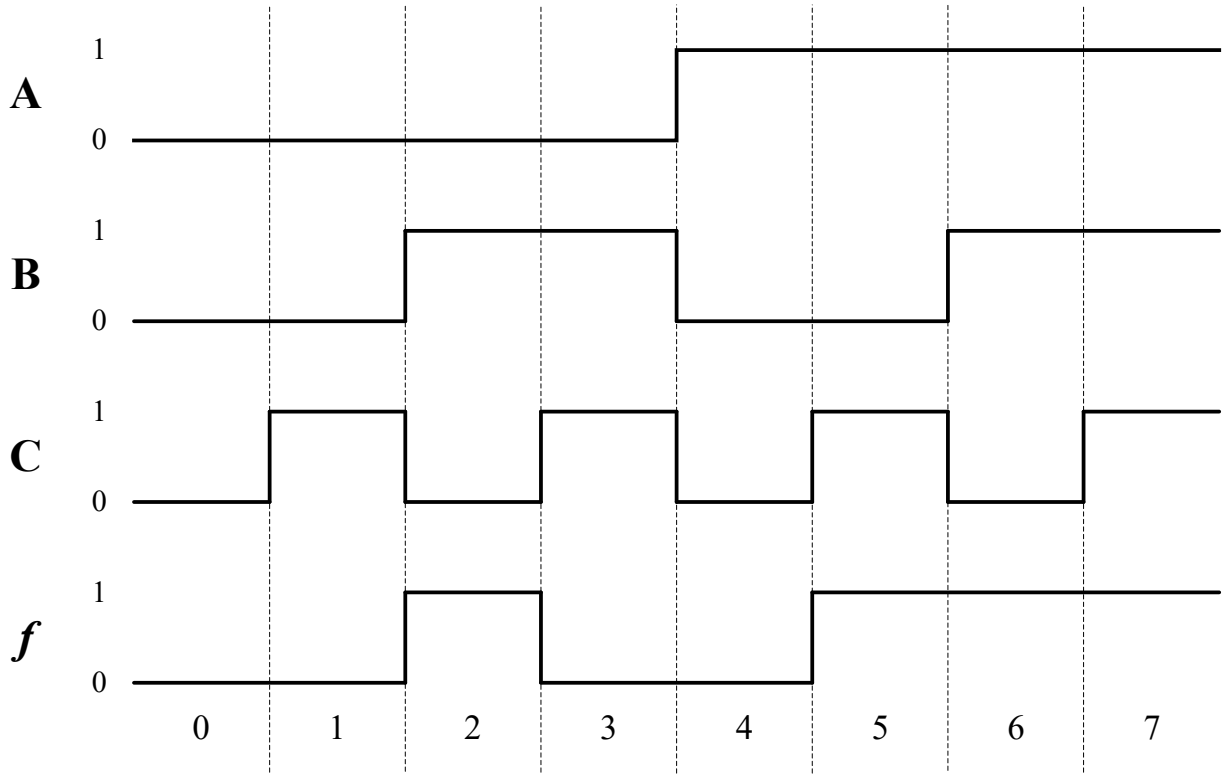
BC′

$$f = AC + BC'$$

Circuit Diagram:



Timing Diagram:



Procedure:

1. Draw logic diagram to implement the function.
2. Select ICs from the equipment list.
3. Note the output logic for all combinations of inputs.
4. Now fill out the truth table for that function.
5. Simplify the function in POS and in SOP form using K-map.
6. Repeat step - 1, 2 and 3.

Experiment: 3

Experiment name: *Introduction to FPGA, Quartus Software & Verilog HDL*

System Synthesis:

- CAD Tool: Quartus
- Hardware Description Language (HDL): Verilog
- Development Board: Altera FPGA Board

Procedure:

We have two development boards currently available in Lab – DE2 & DE1-SoC. We will use Quartus II and Quartus Prime for these two boards respectively.

Design Flow:

1. Start:	Project	
2. Design:	Schematic	Verilog
3. Verification:	Simulation	FPGA Board

Function Implementation:

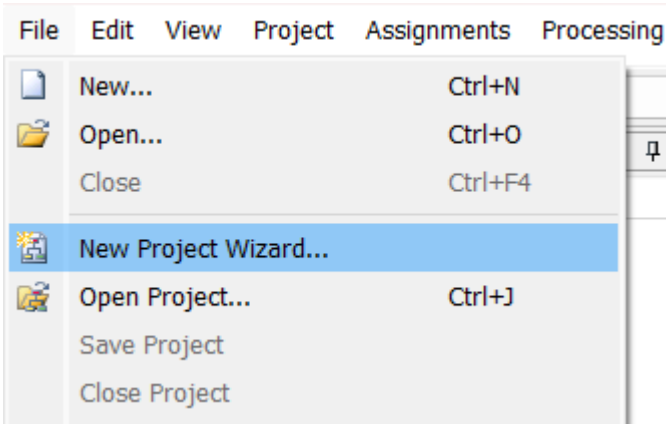
Here, we are going to implement the following function –

$$f = AC + BC'$$

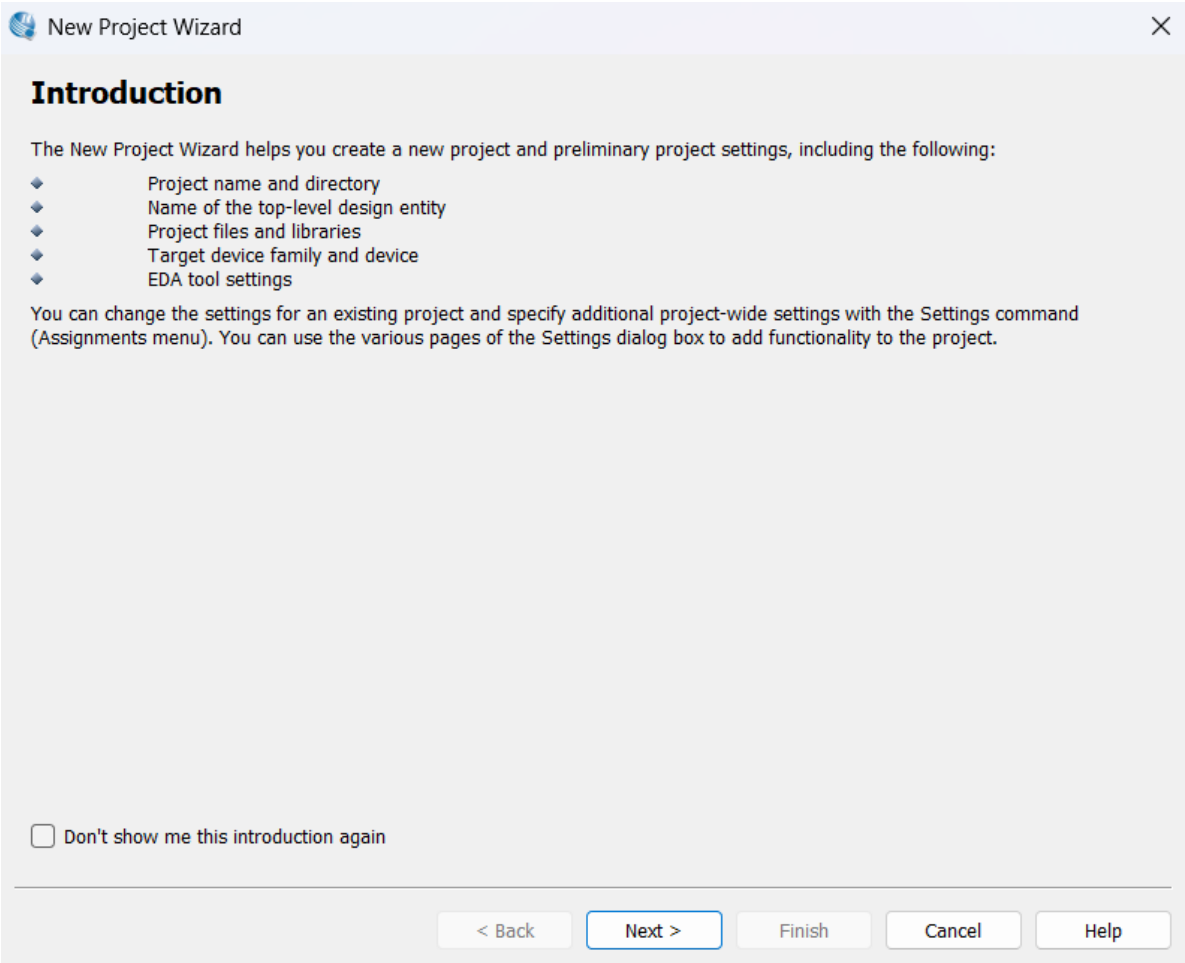
Project in Quartus:

P1. Open **Quartus** software.

P2. Go to ***File → New Project Wizard.***



P3. The following window will open up. Click **Next**.

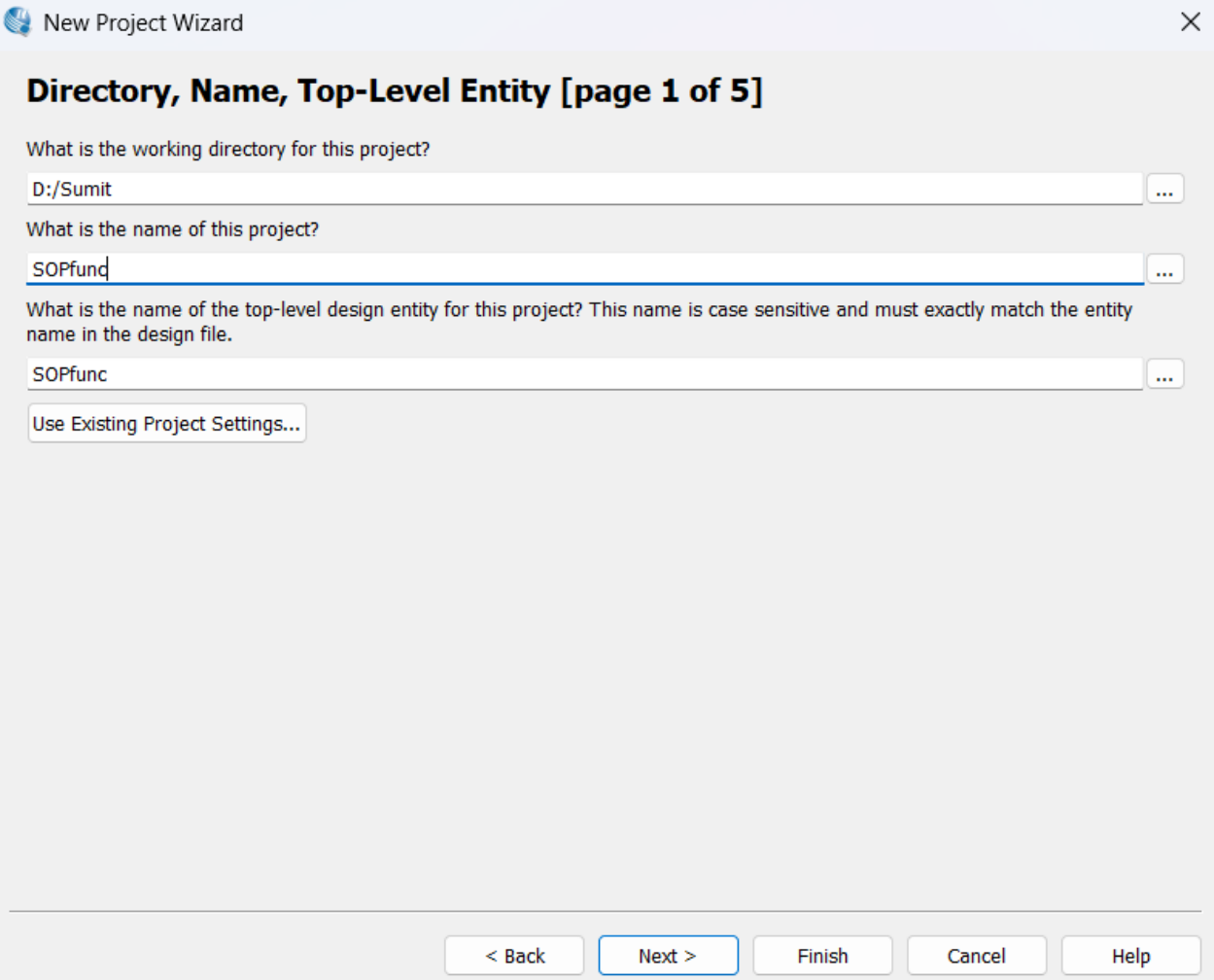


P4. In the next window that appears, change the default working directory to your working directory (e.g. **E:\Sumit\DLD**) and give a name to this project.

When giving name of project and top-level design entity, keep in mind these two important points –

** **Space in the name is not allowed**.

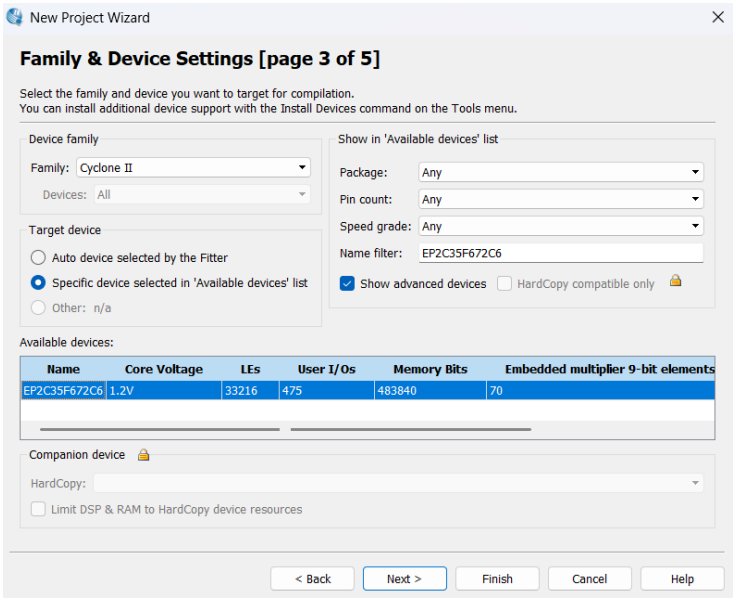
** **It is recommended that top-level design entity file should have the same name as the name of the project.**

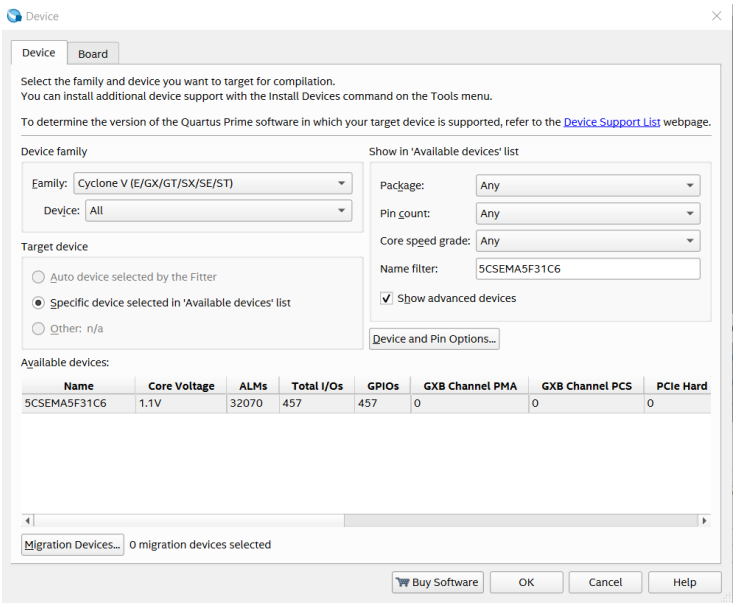


P5. In the next window, you may include files to your project, which we will demonstrate later, for now, click **Next**.

P6. In the following window, select **Device Family** and type your device name in **Name filter**. Double click on it under **Available devices** so that **Specific device selected in 'Available devices' list** is selected under **Target Device**.

Family	Name Filter
Cyclone II	EP2C35F672C6
Cyclone V	5CSEMA5F31C6

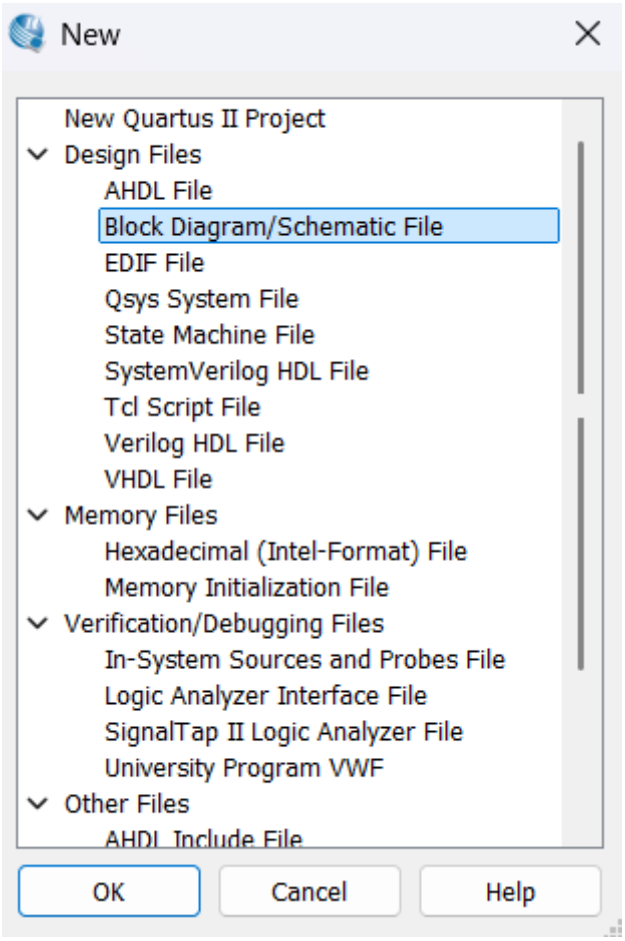




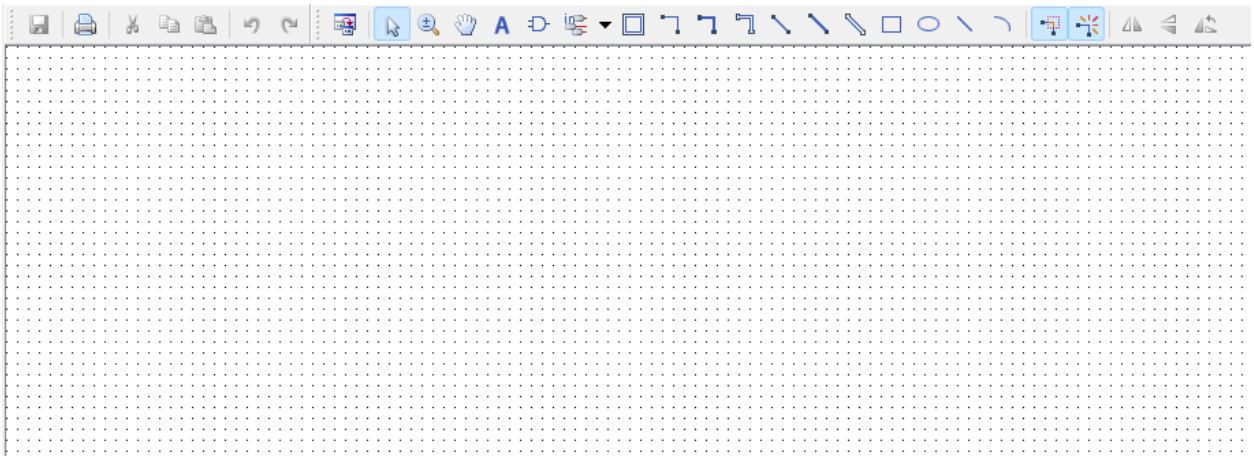
P7. Click **Finish**. This completes the steps for creating a project file.

Block Diagram/Schematic Design:

B1. Go to ***File*** → ***New***. Select **Block Diagram/Schematic File** and click **OK**.



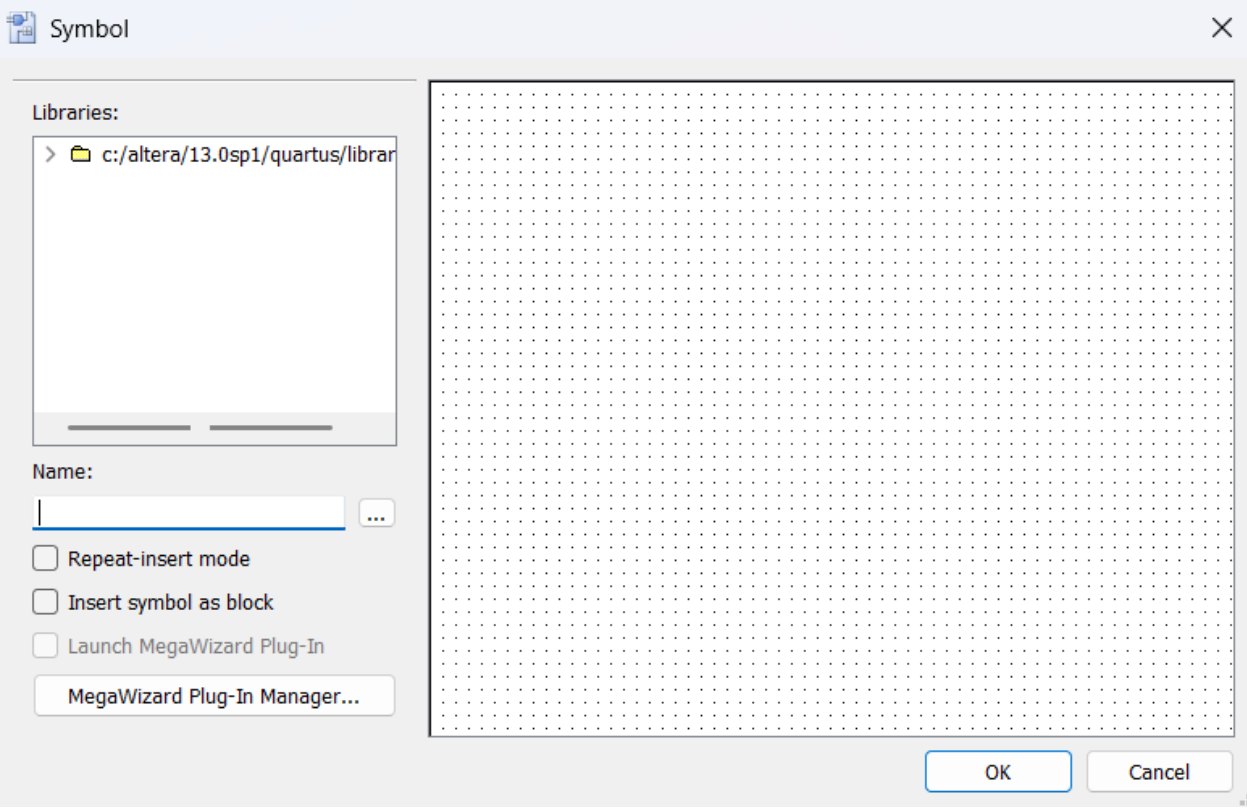
A blank block diagram window will appear.



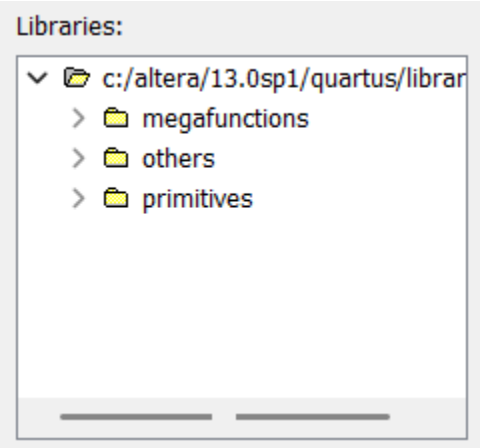
B2. To implement the circuit, we will need an AND, OR & NOT gate. From the menu-bar, click on icon for **Symbol Tool**, or alternatively double clock on the blank schematic window.



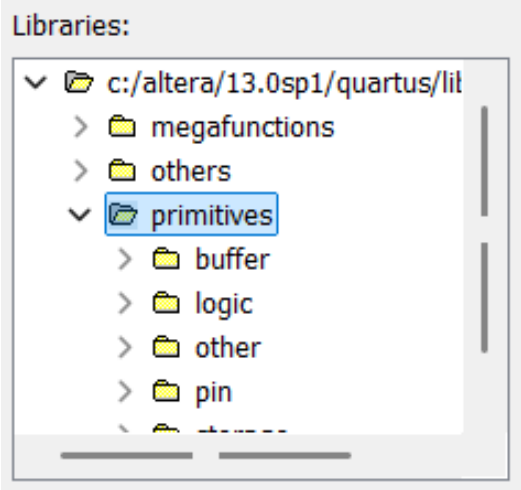
B3. The following window will appear. Under **Libraries**, click on the plus icon beside **c:/altera/…**



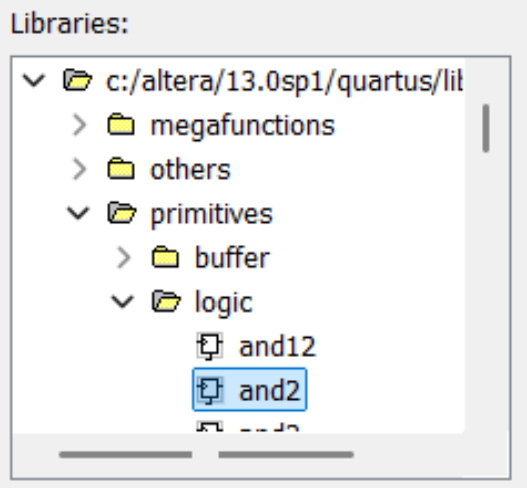
B4. After expanding the plus icon, you will see the following library directories:



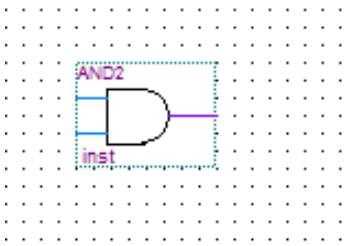
B5. Click on the plus sign beside **primitives**.



B6. Different Logic Gates (AND, OR, NOT etc.) are under **logic** directory, input and output pins are under **pin** directory, and flip-flops are under **storage** directory. Go to **logic** directory and select **and2** from the list for a 2-input AND gate and click **OK**.

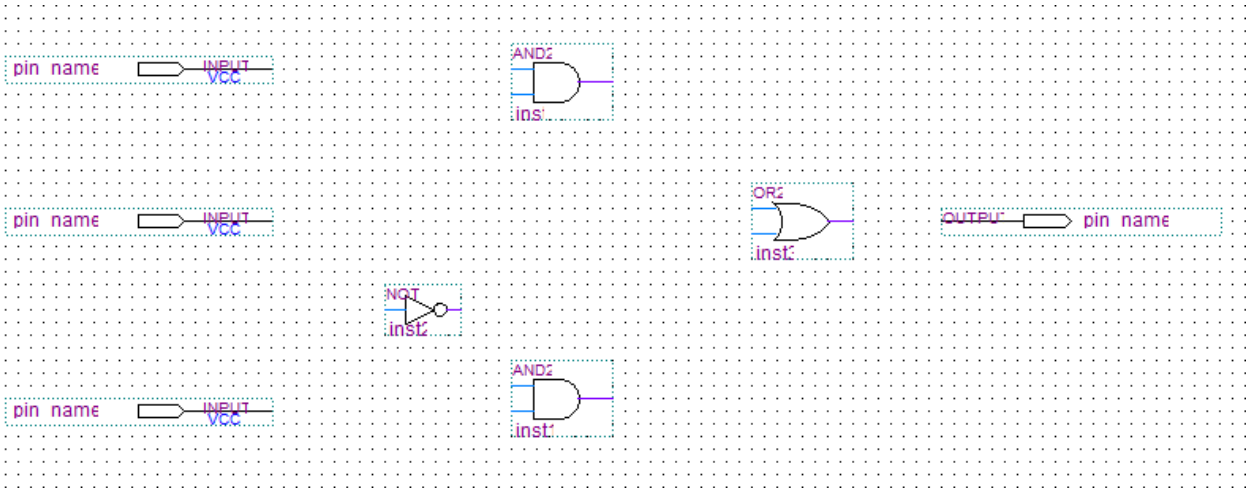


B7. Then go to block diagram window and place the symbol on it.

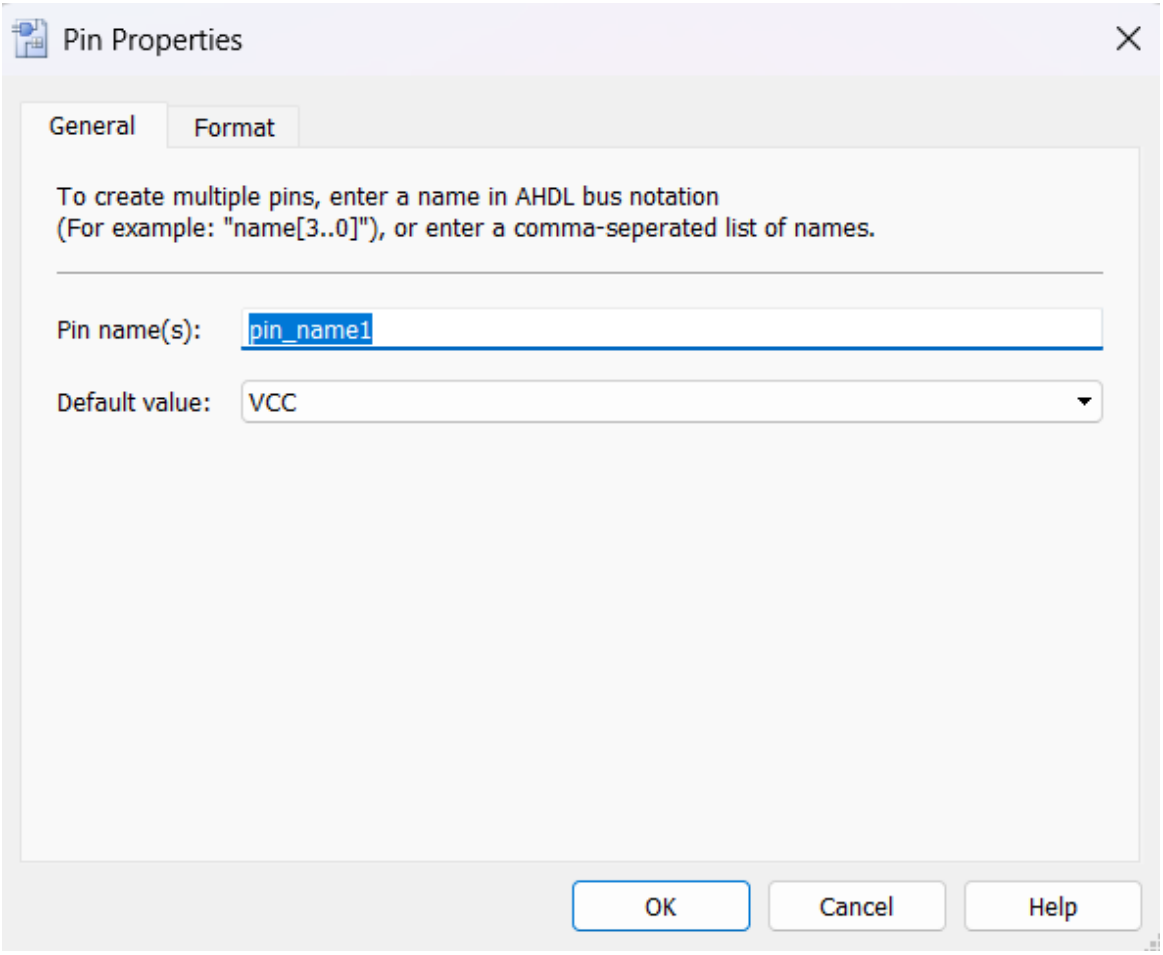


B8. Do the same for the OR & NOT gate.

B9. Now, click on the drop-down menu on **Pin Tool** at toolbar and select **Input/Output** pins.



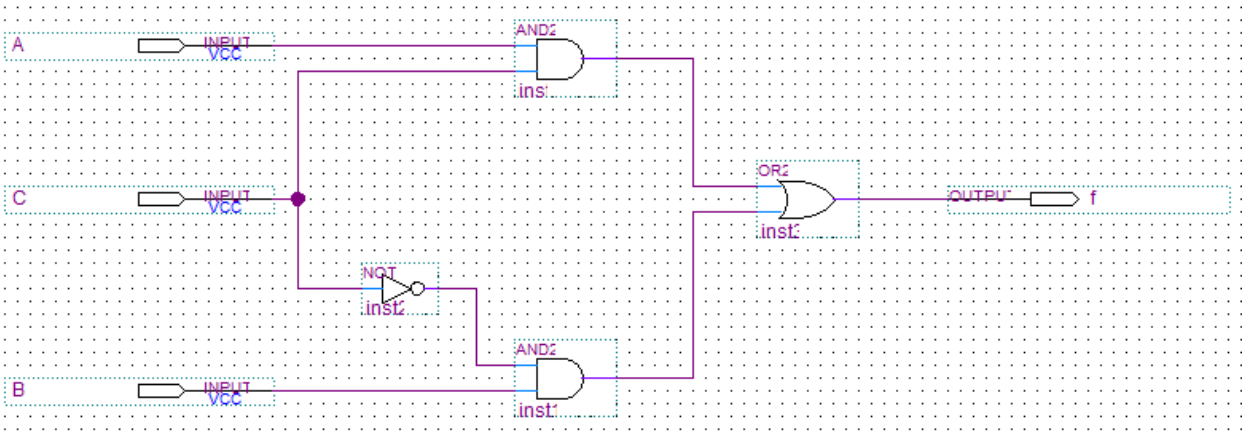
B10. You can click on the pin names and rename them.



B11. Now, in the block diagram window, move cursor to input/output pins on gates and you will see wiring icon showing up, or you can select **Orthogonal Node Tool** from the left menu bar.



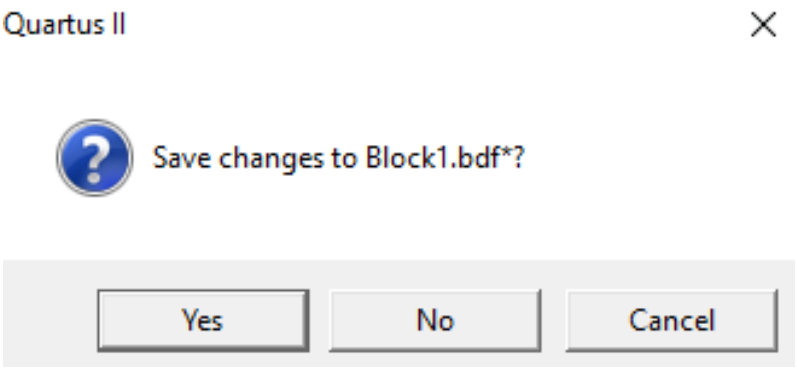
B12. Now, wire the gates and pins to construct the circuit. When completed, it should look like the following –



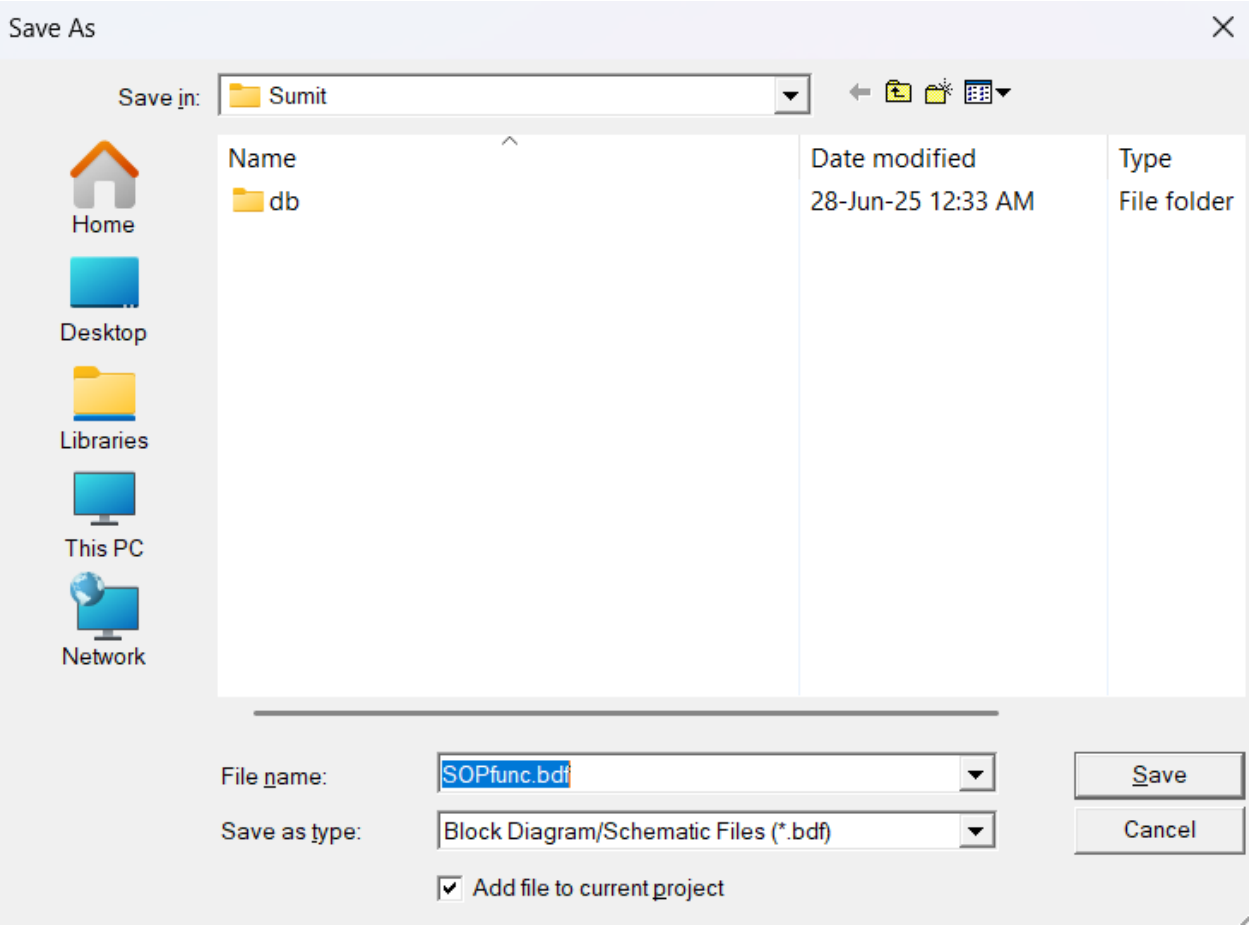
B13. Click on **start compilation** button on the top menu bar.



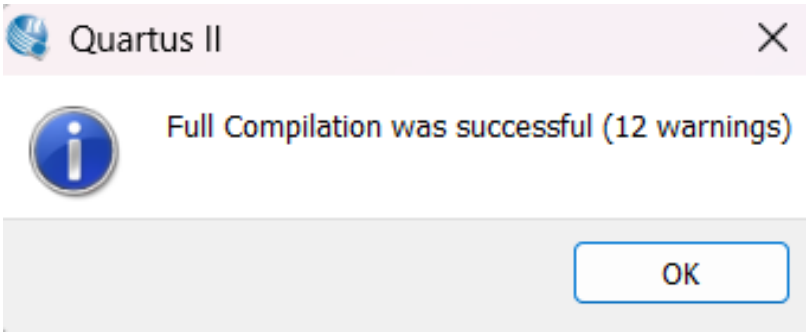
B14. Click **Yes,** if you are prompted to save the block diagram.



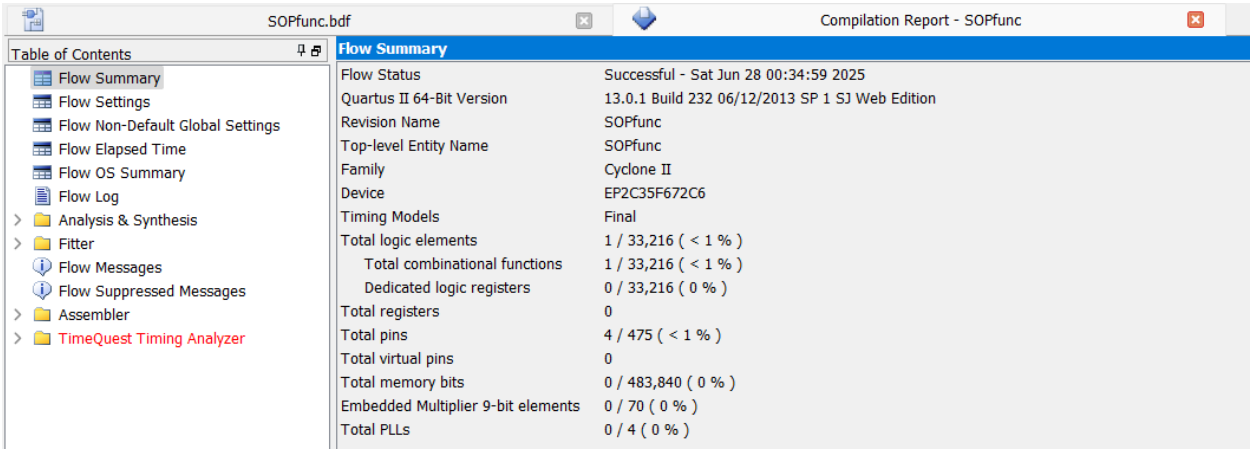
B15. File name of **top-level design entity** is recommended be the same as that of the project name (e.g. SOPfunc). Click **Save**.



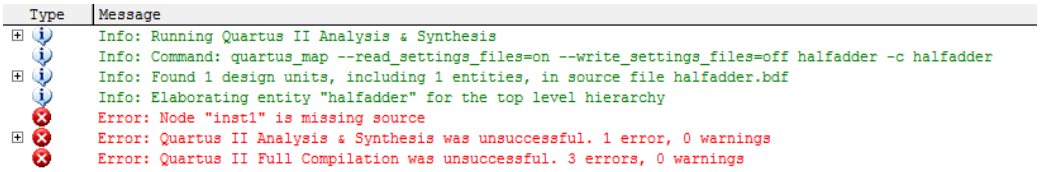
B16. If compilation is successful, you will get a message like the following. Click **OK**.



B17. Ignore warnings for now. **Compilation report-flow summary** will present you with the details:



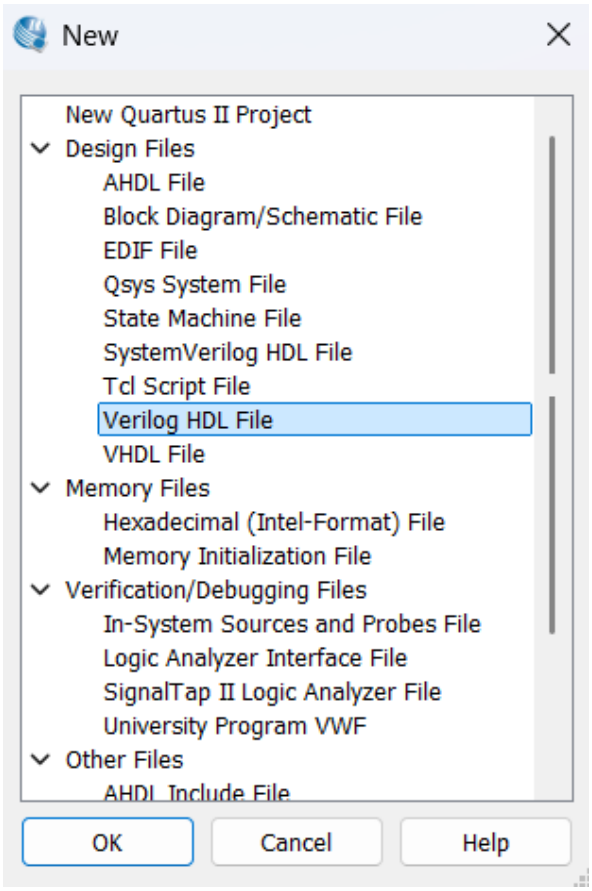
If any error occurs, you will find them at the bottom window.



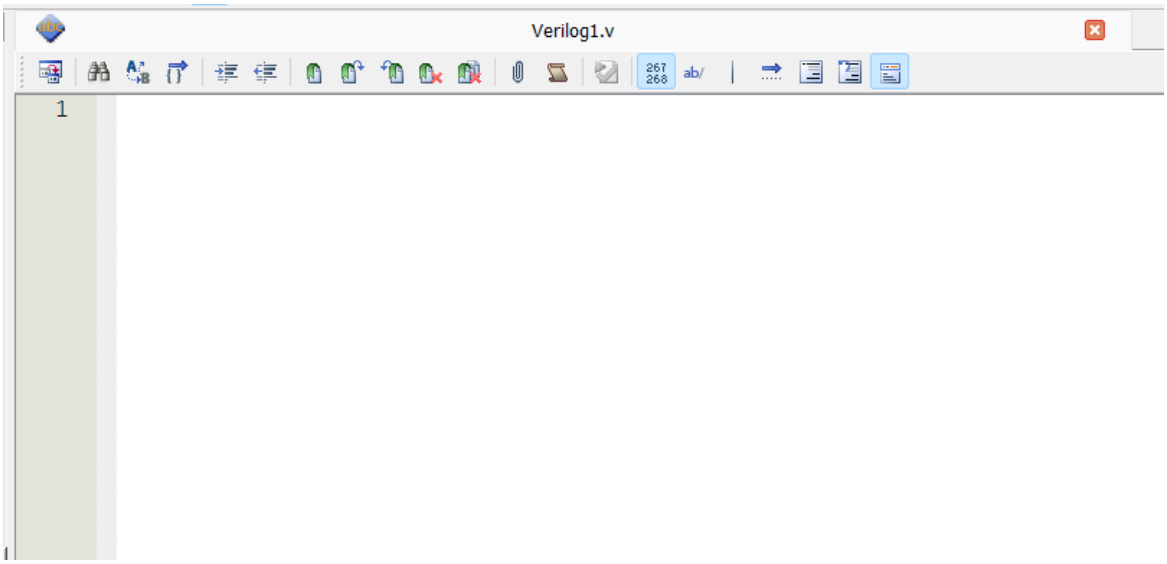
In case an error occurs, find the error on the block diagram and rerun compilation.

Verilog HDL File:

V1. Click on File → New… & a **New** Window will pop-up. Select **Verilog HDL File** under **Design Files** and clink **OK**.

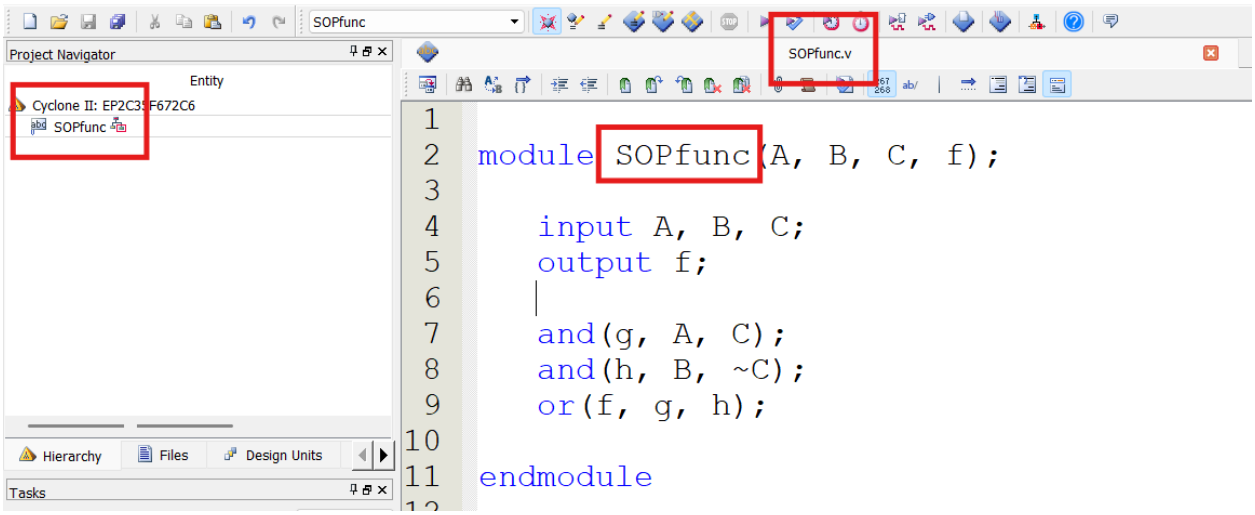


V2. A blank window will appear where we will write our Verilog HDL code.



V3. Complete the Code and Save the File.

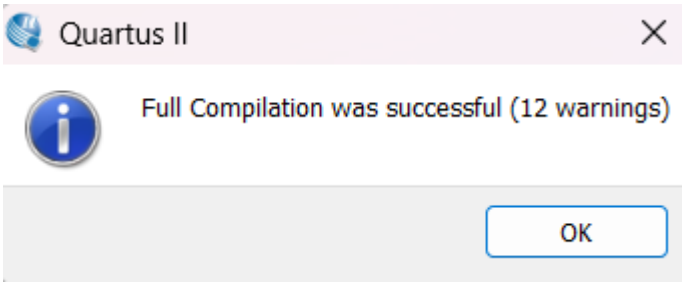
*** **Make sure Module Name, Verilog File Name, Top-level Entity (Project) Name is eactly same.** (i.e. SOPfunc in these 3 cases)



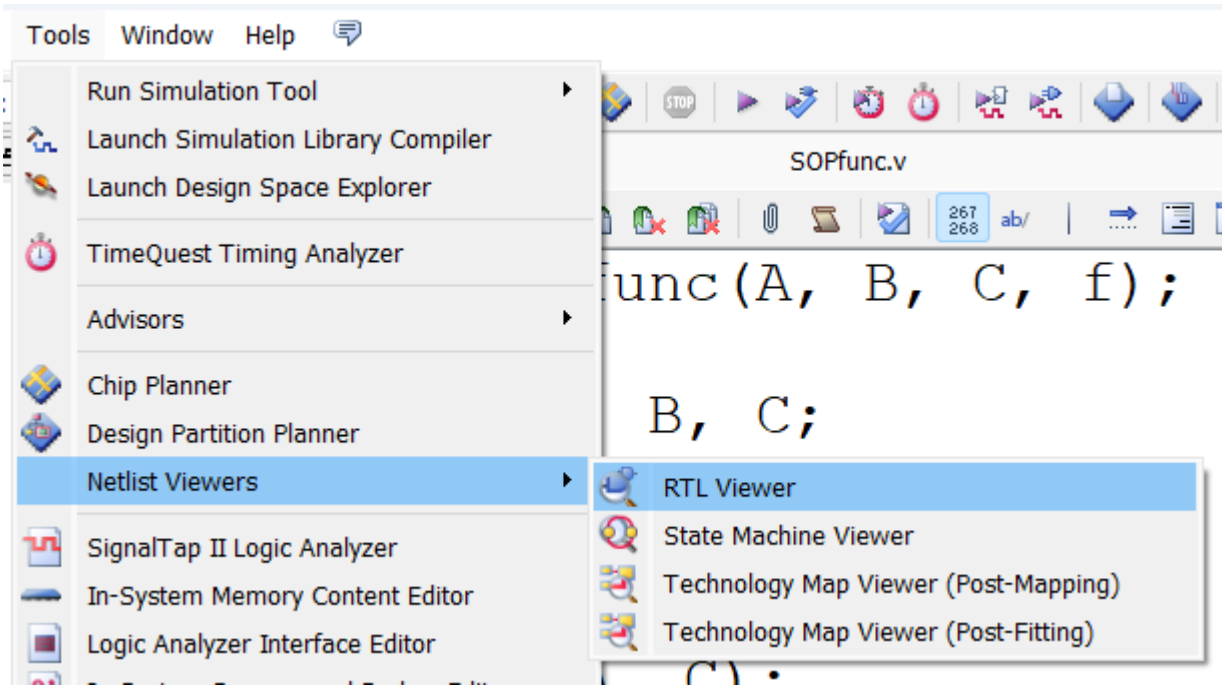
V4. Click on **start compilation** button on the top menu bar.



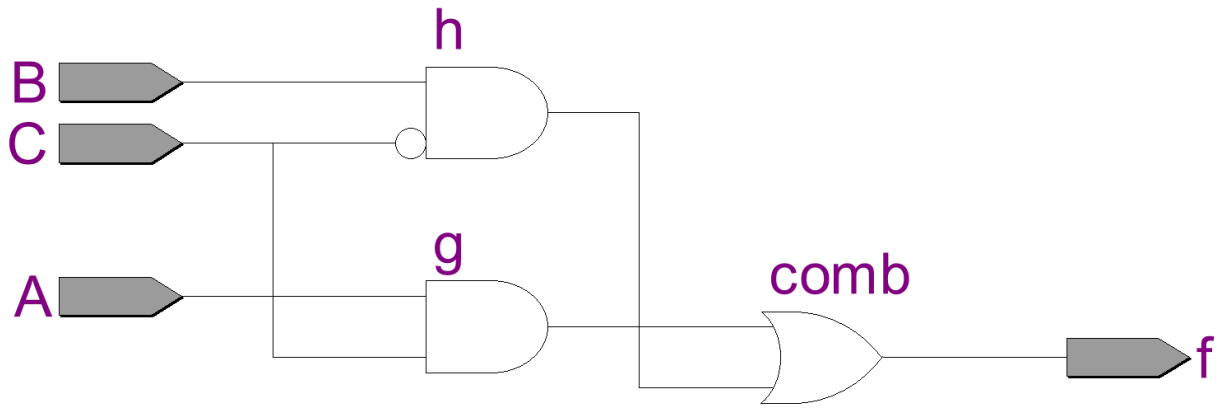
V5. If compilation is successful, you will get a message like the following. Click **OK**.



V6. Click On **Tools → Netlist Viewers → RTL Viewer.**

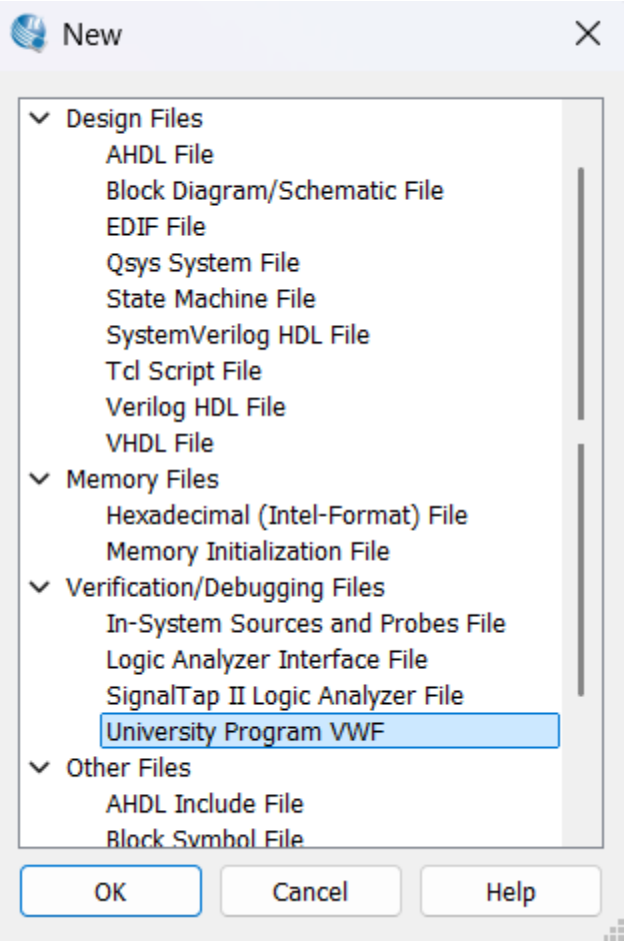


V7. You can see the RTL (Register Transfer Level) view of your Verilog Code.

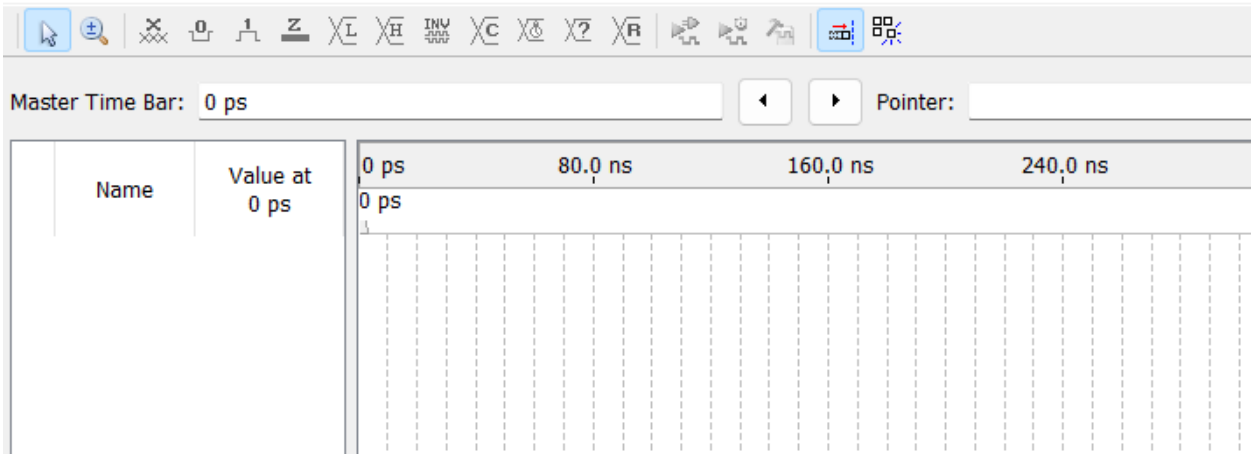


Simulation:

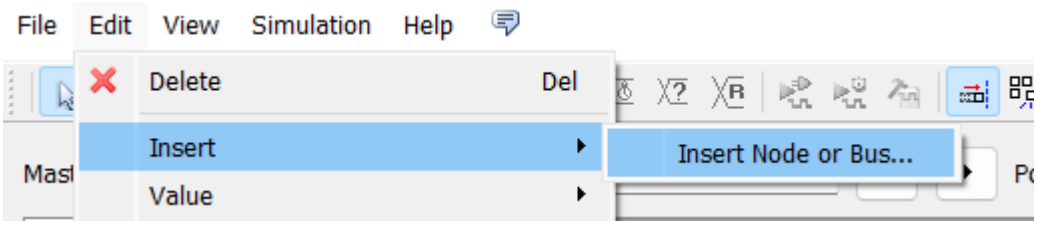
S1. Go to ***File → New*** to create a Vector Waveform File (VWF) which is required for simulating inputs and outputs. Select **University Program VWF** under **Verification/Debugging Files** and click **OK**.



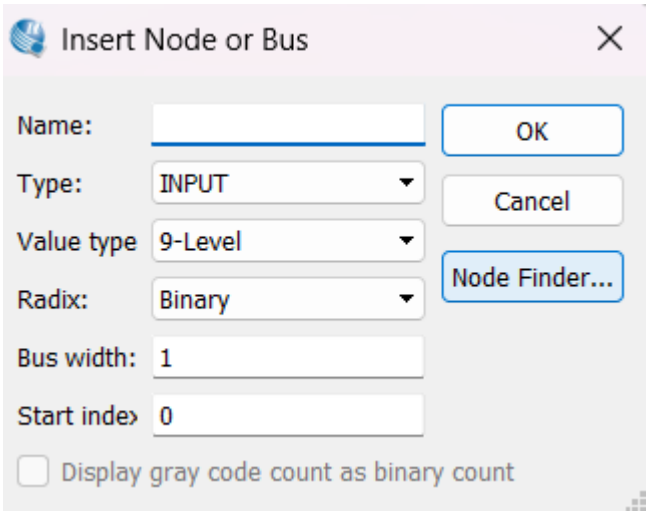
S2. The following window will open up.



S3. Click on **Edit → Insert → Insert Node or Bus..**

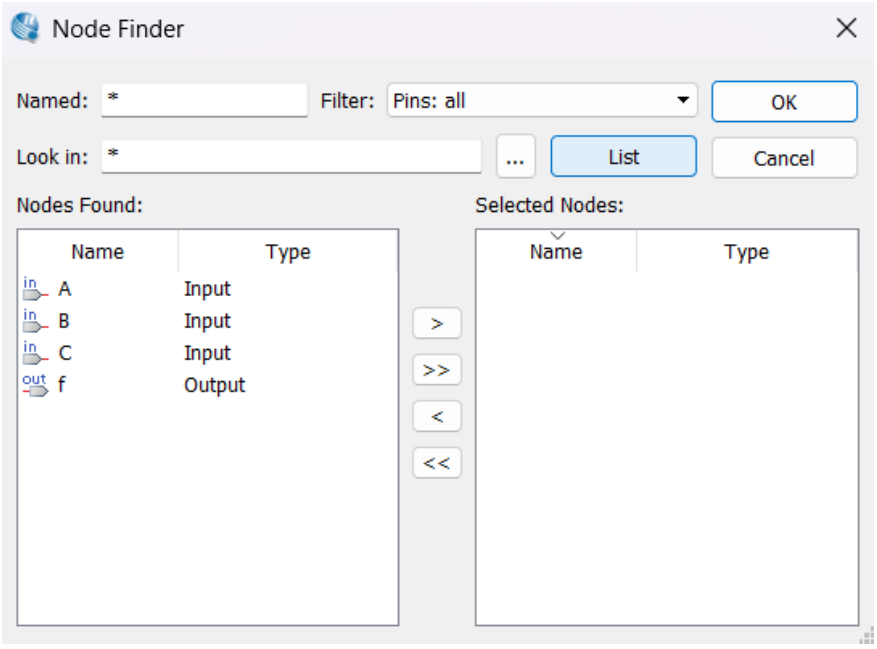


S4. In the **Insert Node or Bus** window, click **Node Finder**.

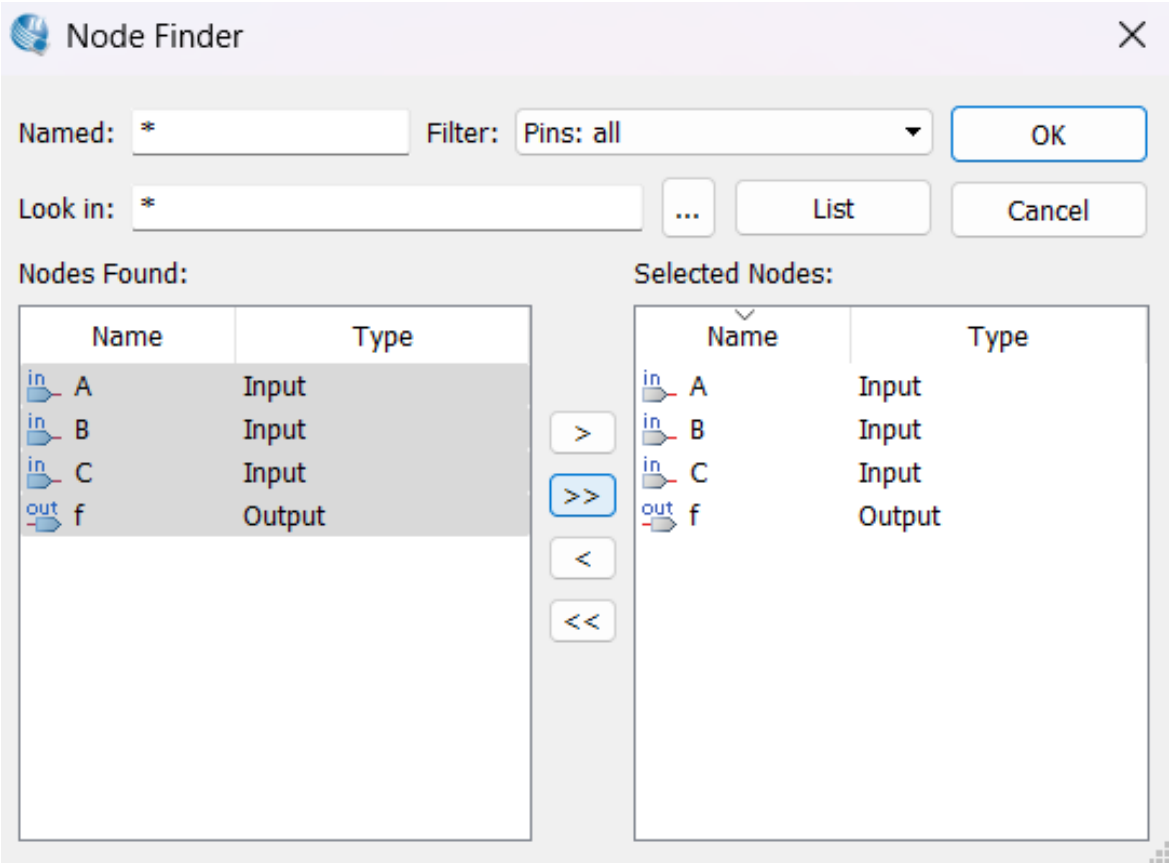


S5. In the **Node Finder** window, Click **List**. Make sure **Pins:all** is selected under **Filter**.

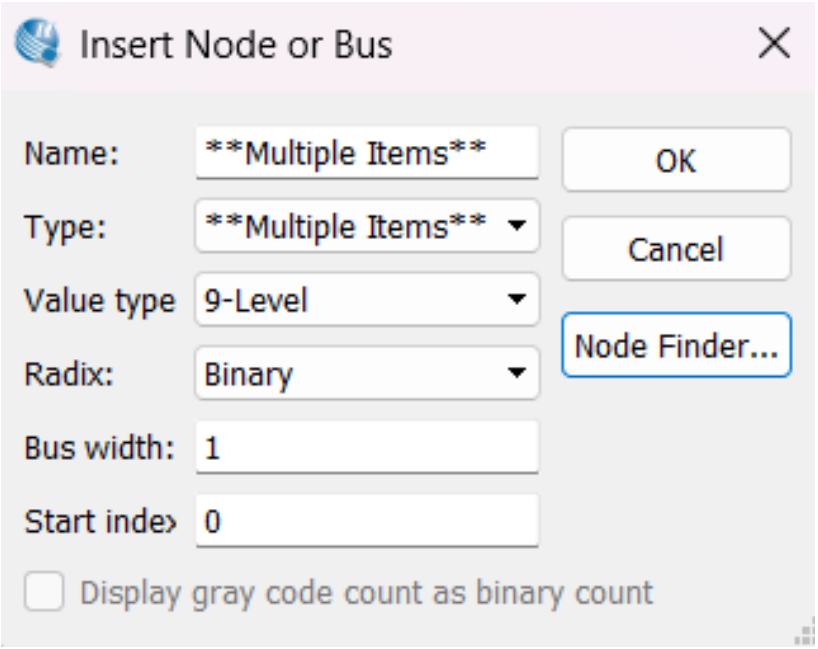
S6. Now the window will look like the following. Click on the *'>>'* button.



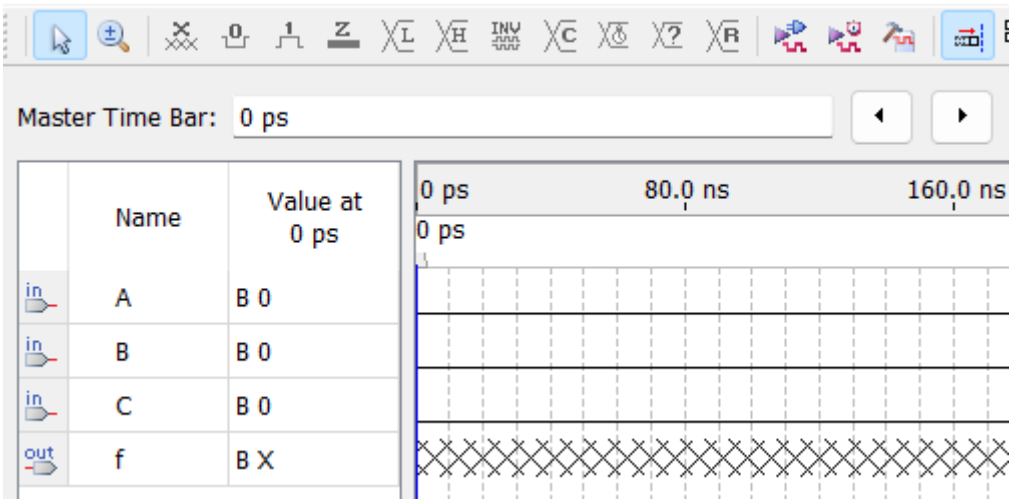
S7. Now, it should appear like the following. Click **OK**.



S8. In the following window, click **OK**.

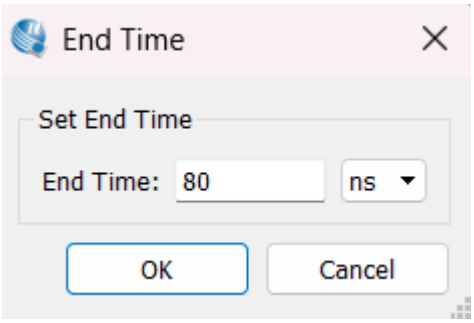


S9. The vector waveform file will now look like the following:

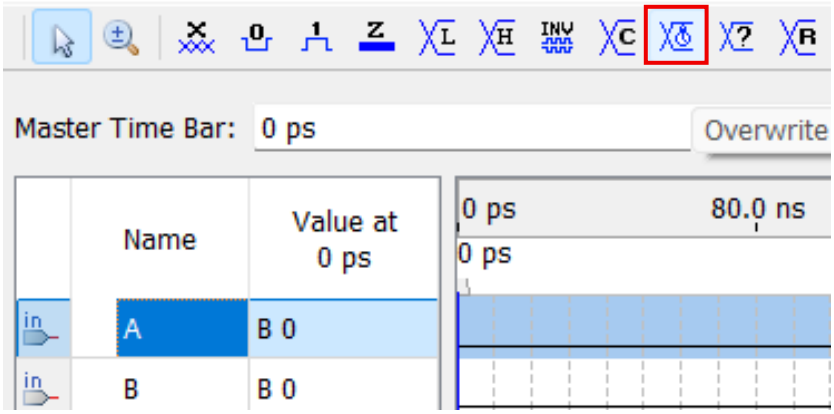


Now, you can clearly see the inputs and outputs.

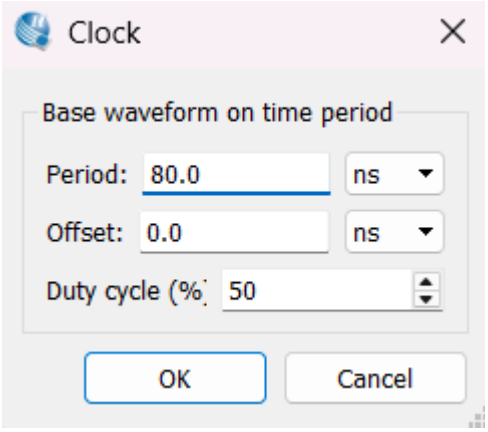
S10. Click on **Edit → Set End Time…** and set the time to **80ns**.



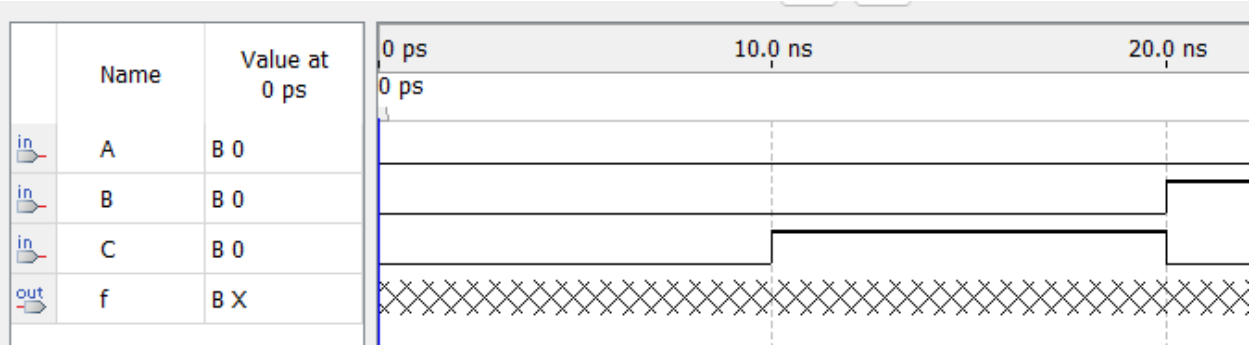
S11. Select an input and from tool bar, click on the **Overwrite Clock** icon.



S12. In the **Clock** window, set parameters of the clock. Only change the **Period** and keep everything else the same as before. Double clock period as you move from one input (**LSB**) to another as this will enable you to simulate the circuit for all possible input signal conditions.



S13. Now, after setting all the input clocks, **Vector Waveform File** will look like the following:

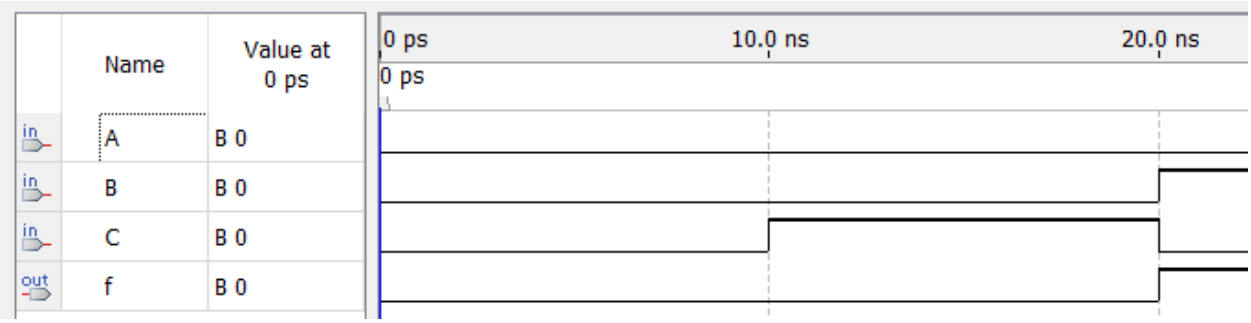


S14. Save the **Vector Waveform File**.

S15. Click on **Run Functional Simulation** icon.



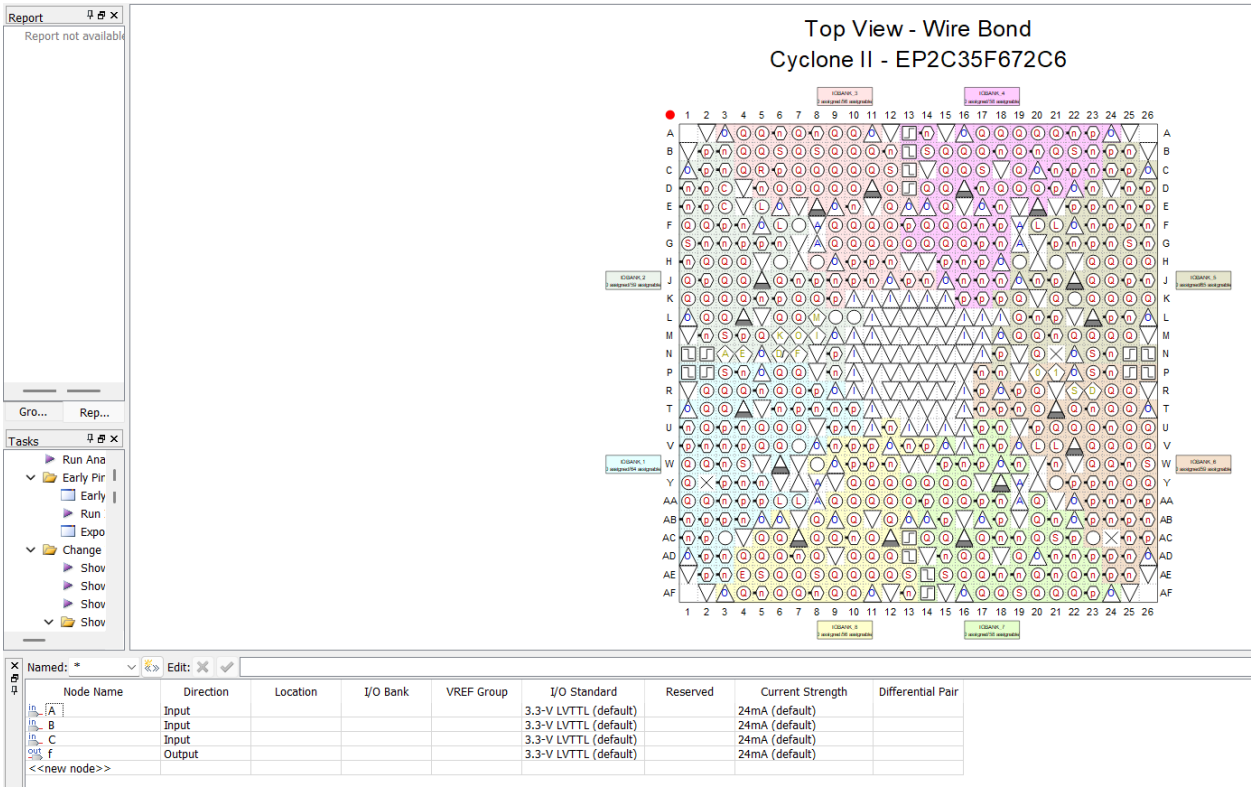
S16. Now observe the simulated waveforms..



FPGA Programming:

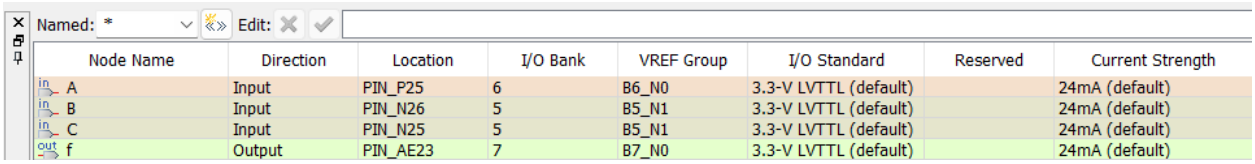
Cyclone II (DE2 Board)

FA1. To load the program on to the FPGA board, go to ***Assignments → Pins Planner***



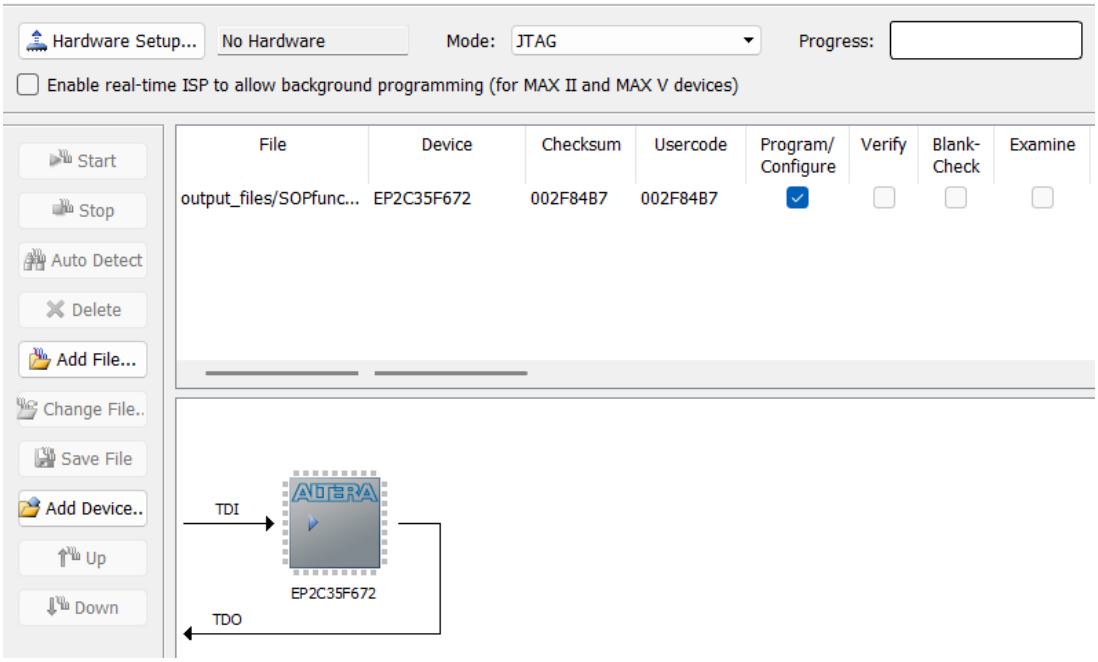
FA2. We will be using toggle switches for inputs and Red LED for output. Assign the pins as follows. Click on **Location** and type in the pin name.

INPUT			OUTPUT		
Signal	**Switch No.**	**Pin No.**	**Signal**	**LED No.**	**Pin No.**
A	SW2	PIN_P25	*f*	LEDR0	PIN_AE23
B	SW1	PIN_N26			
C	SW0	PIN_N25			

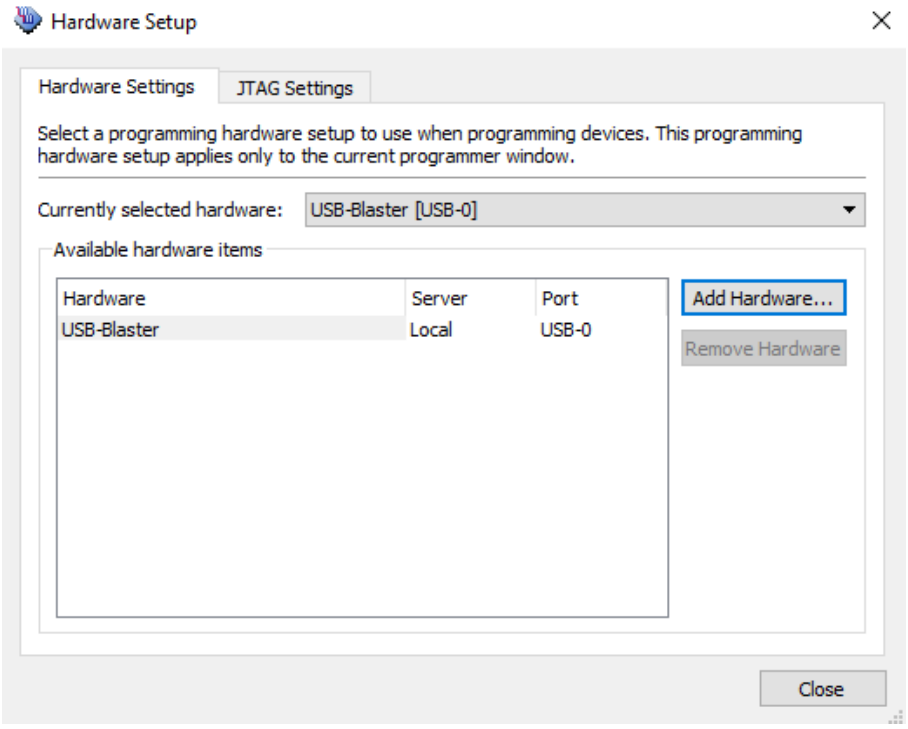


FA3. Close the window. Note that now you will have to compile the code again. So go to **Start Compilation**.

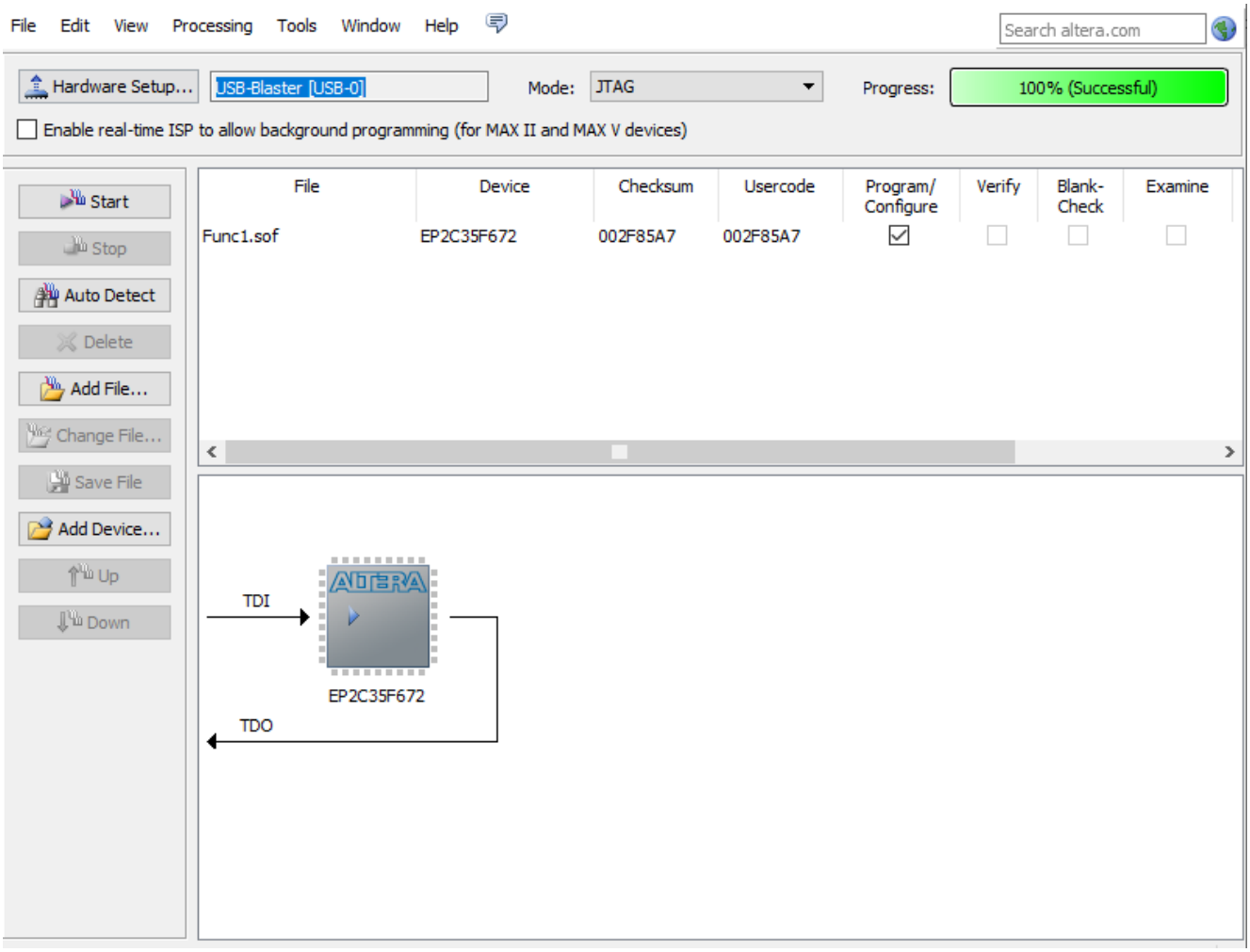
FA4. After successful compilation, go to ***Tools → Programmer***.



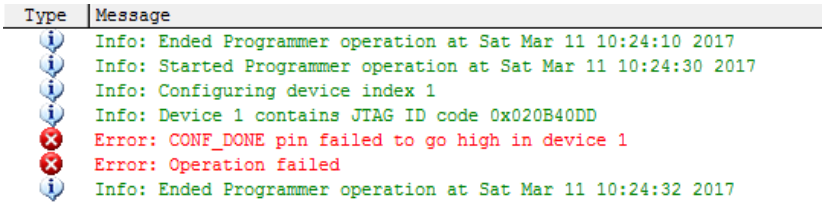
FA5. **Turn ON** the FPGA Board. Select **Hardware Setup**. In the **Hardware Setup** window, select **USB-Blaster (USB-0)** and click **Close**.



FA6. Click **Start** on the Left-Hand Side toolbar. You should see the progress bar moving and going to 100% if loading is successful.



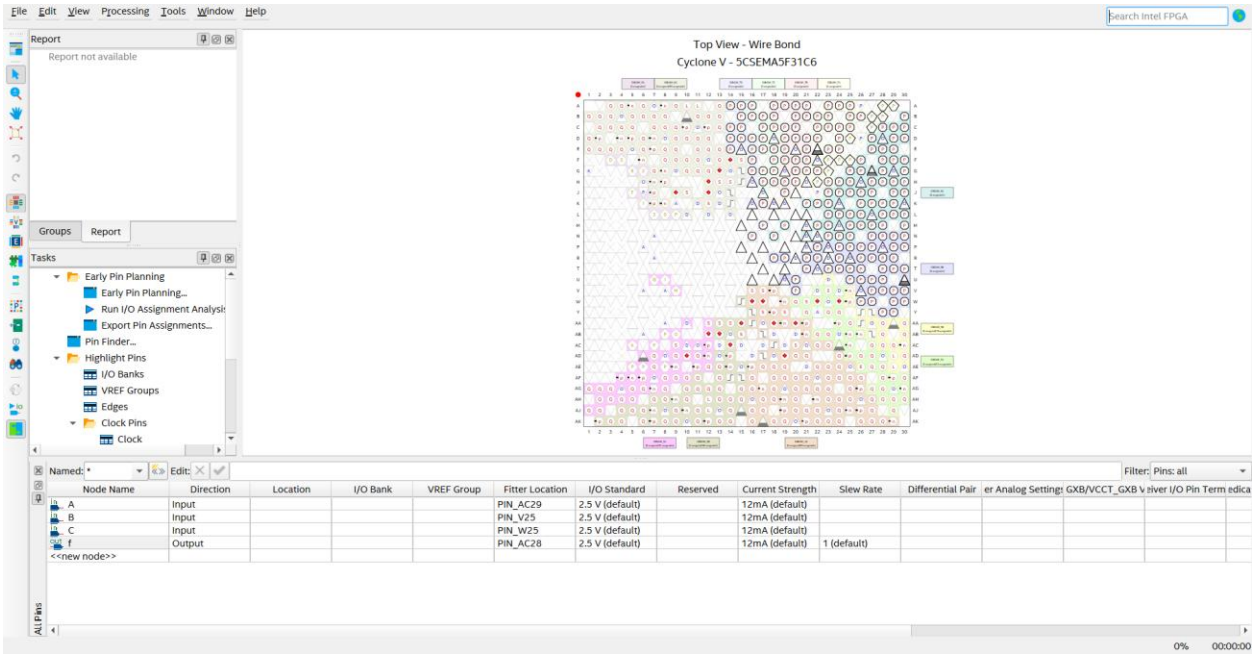
Also note that, if an error occurs, it can be found in the message window:



FA7. Verify the outputs in FPGA board.

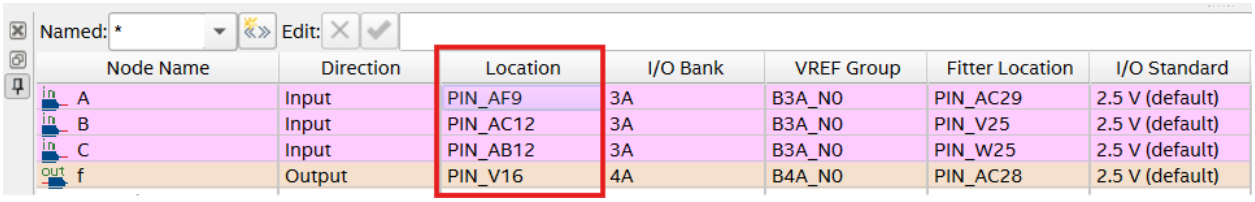
Cyclone V (DE1 SoC Board)

FB1. To load the program on to the FPGA board, go to ***Assignments → Pin Planner***



FB2. We will be using Slide Switches for inputs and Red LED for output. Assign the pins as follows. Click on **Location** and type in the pin name.

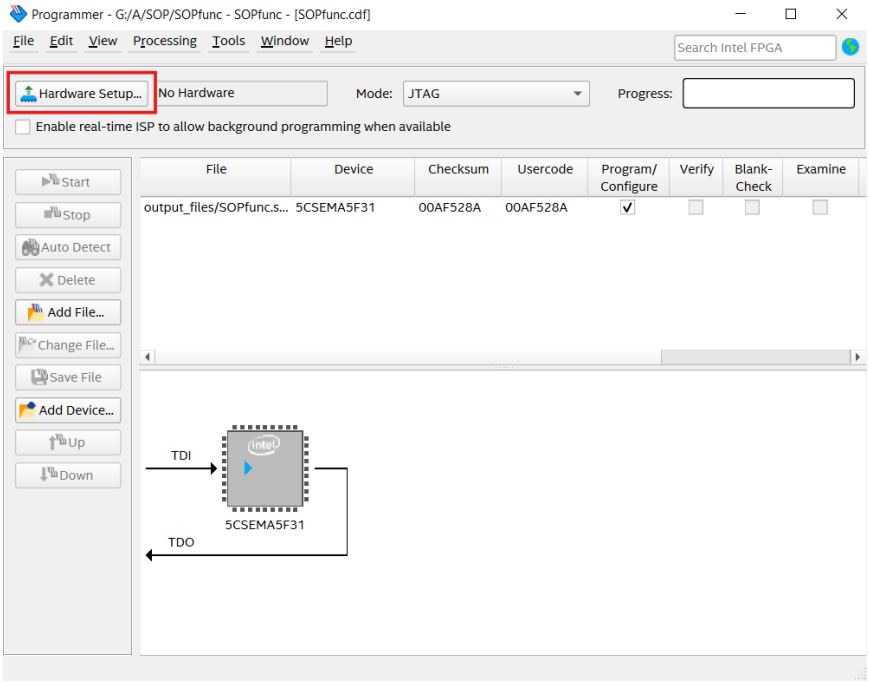
INPUT			OUTPUT		
Signal	**Switch No.**	**Pin No.**	**Signal**	**LED No.**	**Pin No.**
A	SW2	PIN_AF9	*f*	LEDR0	PIN_V16
B	SW1	PIN_AC12			
C	SW0	PIN_AB12			



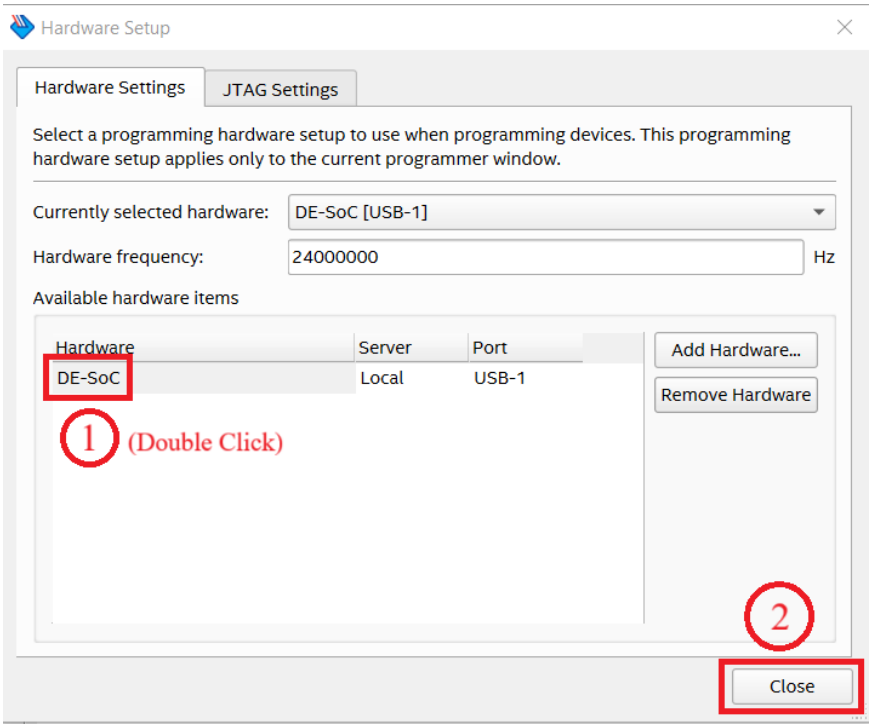
FB3. Close the window. Note that now you will have to compile the code again. So go **to start compilation**.

FB4. After successful compilation, go to ***Tools → Programmer***.

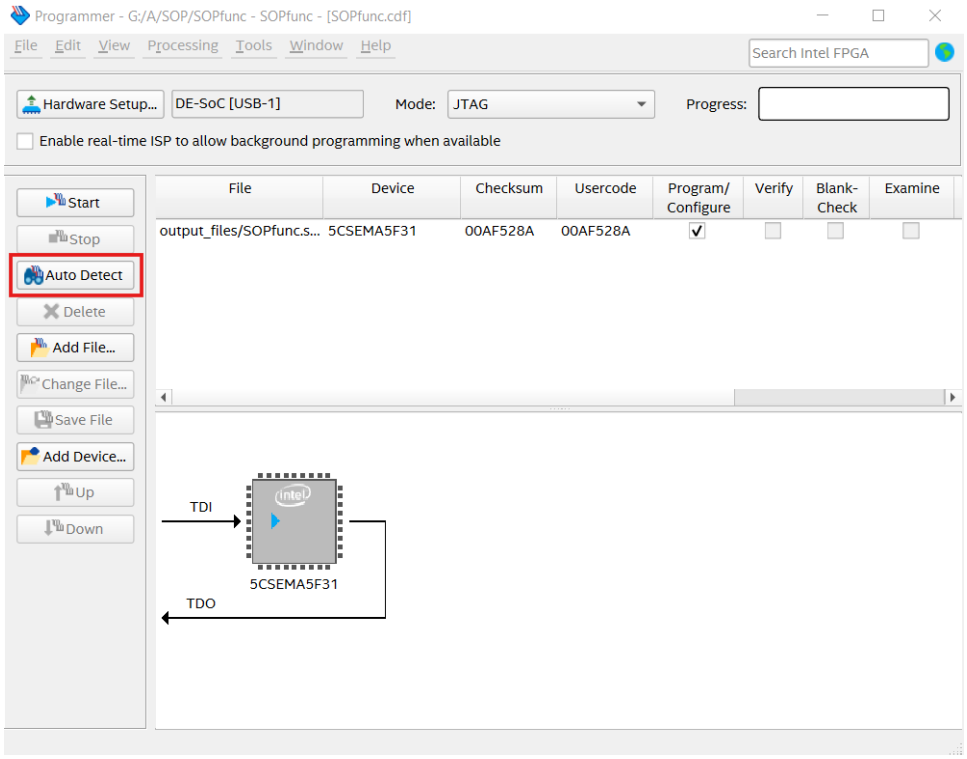
FB5. **Turn ON** the FPGA Board. In the Programmer window, select **Hardware Setup**.



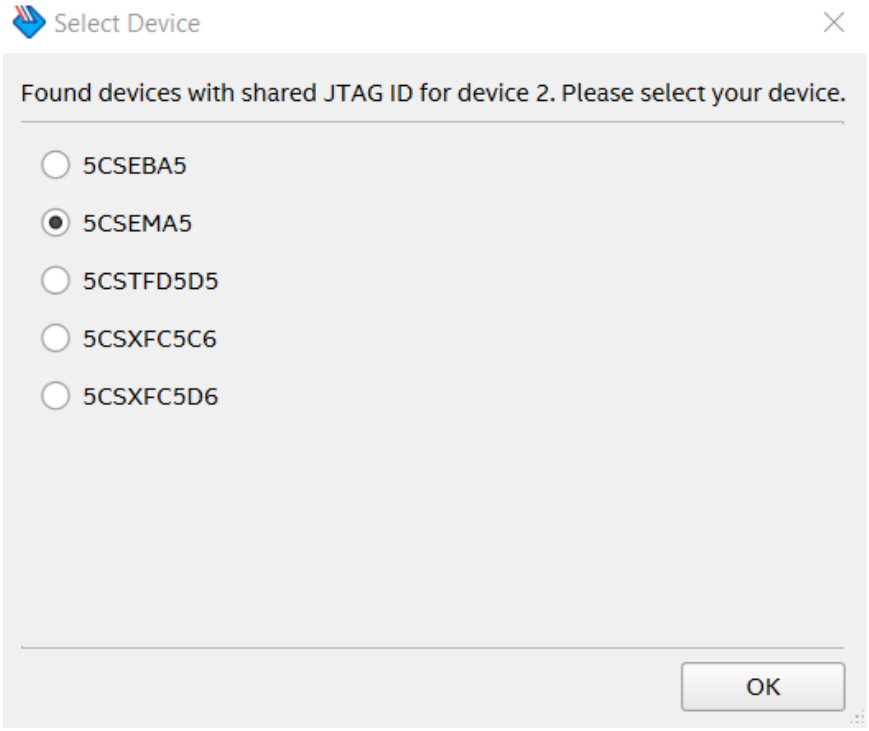
FB6. In the **Hardware Setup** window, first **Double Click** on **DE-SoC** and then Click **Close**.



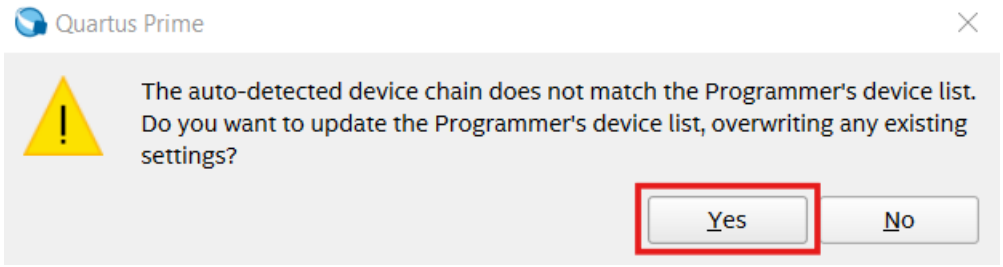
FB7. Open the Programmer and click **Auto Detect.**



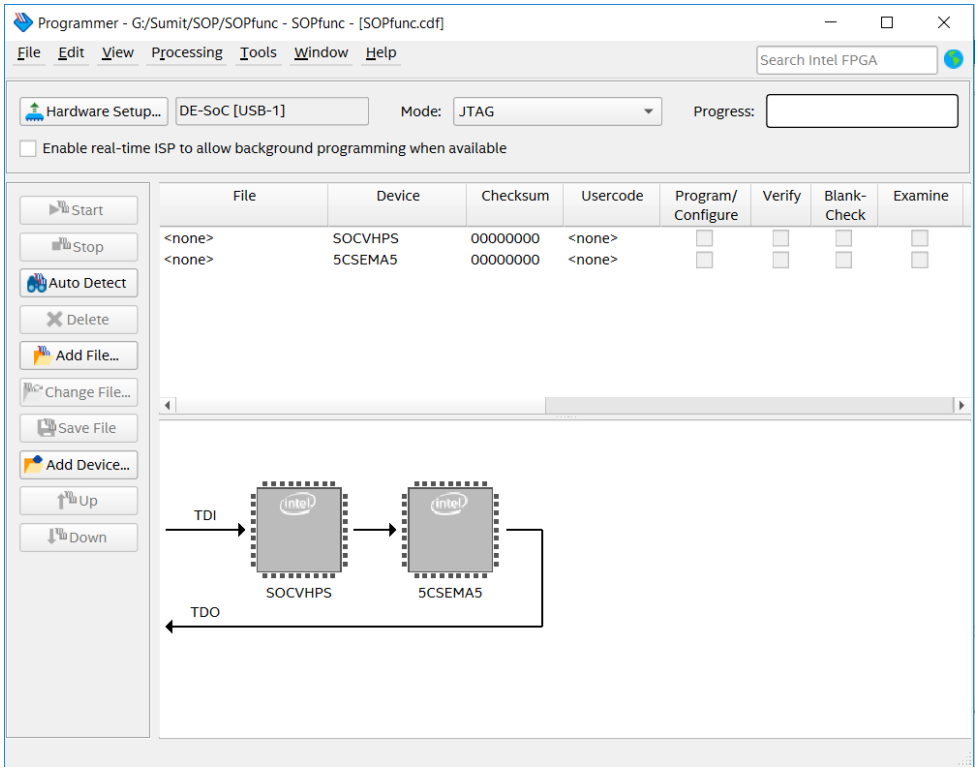
FB8. Select detected device **5CSEMA5.**



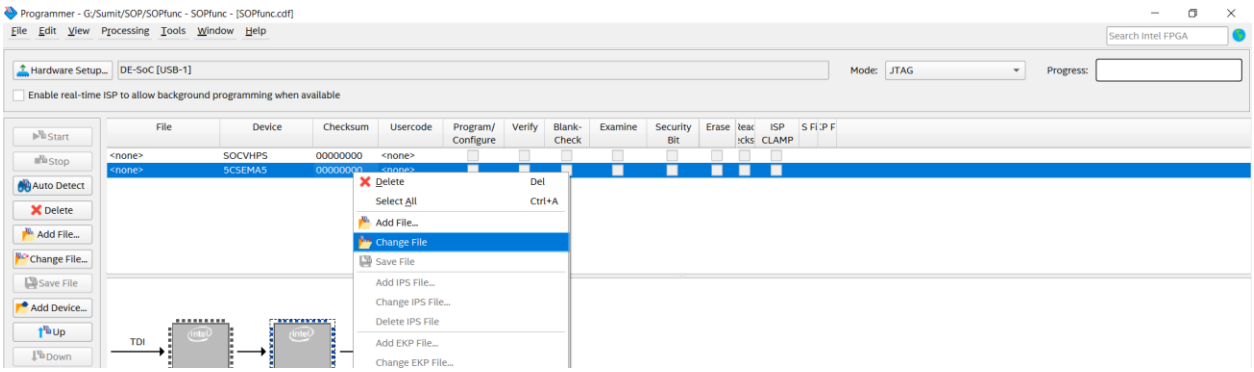
FB9. If the following message windows appears click **Yes.**



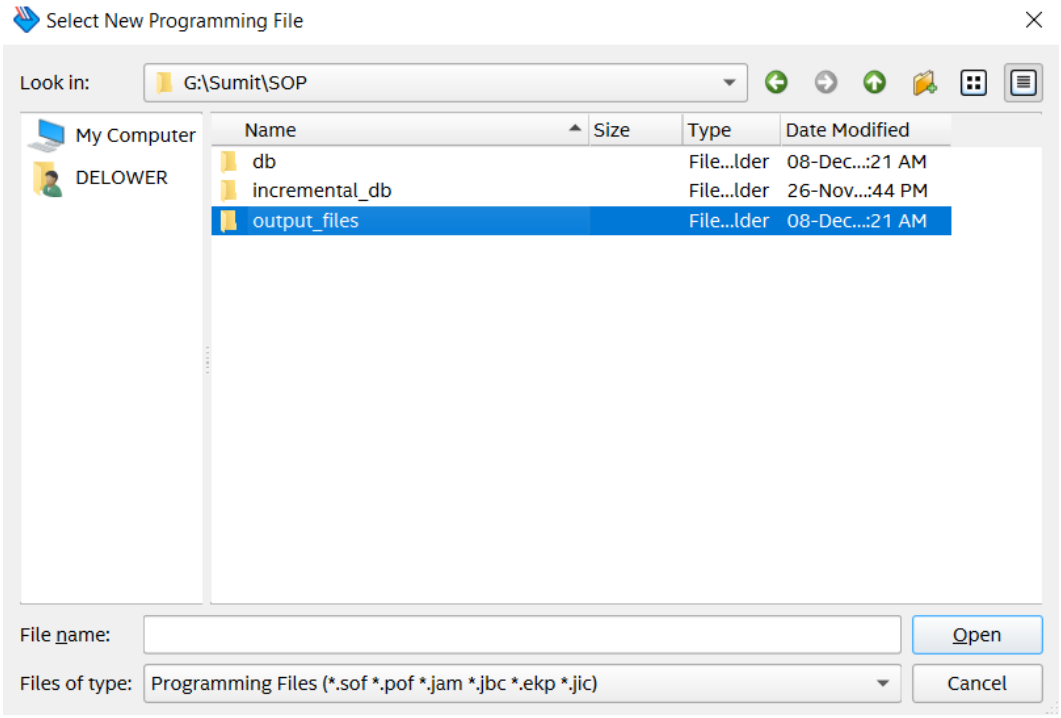
FB10. Both FPGA and HPS are detected.



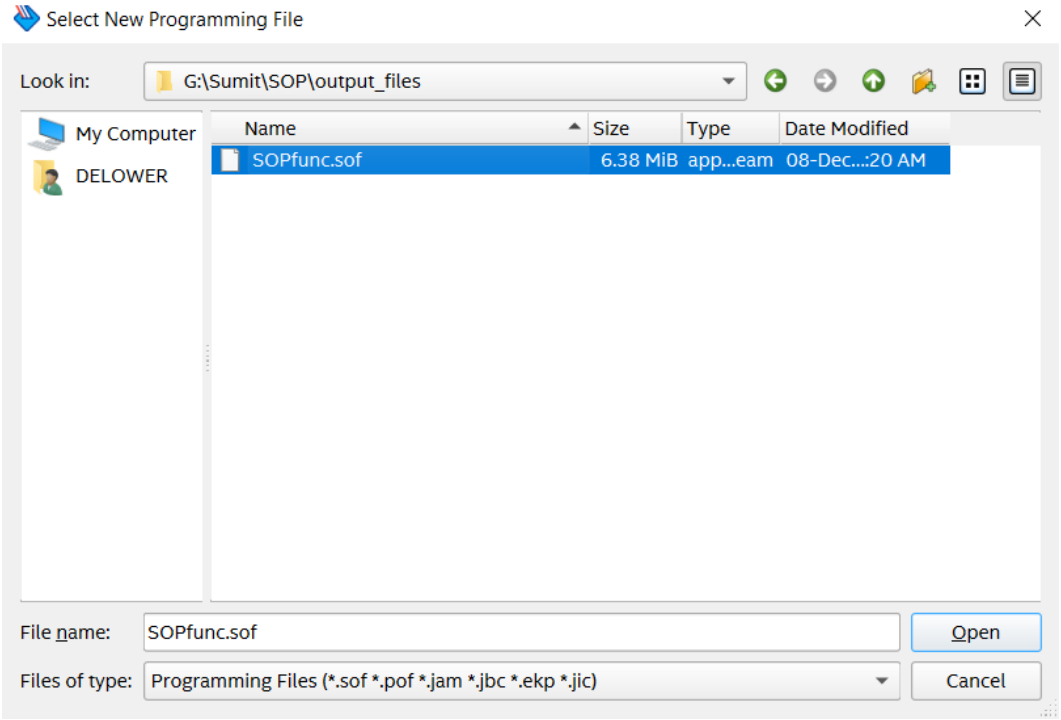
FB11. Right click on the FPGA device (5CSEMA5) and select **Change File**.



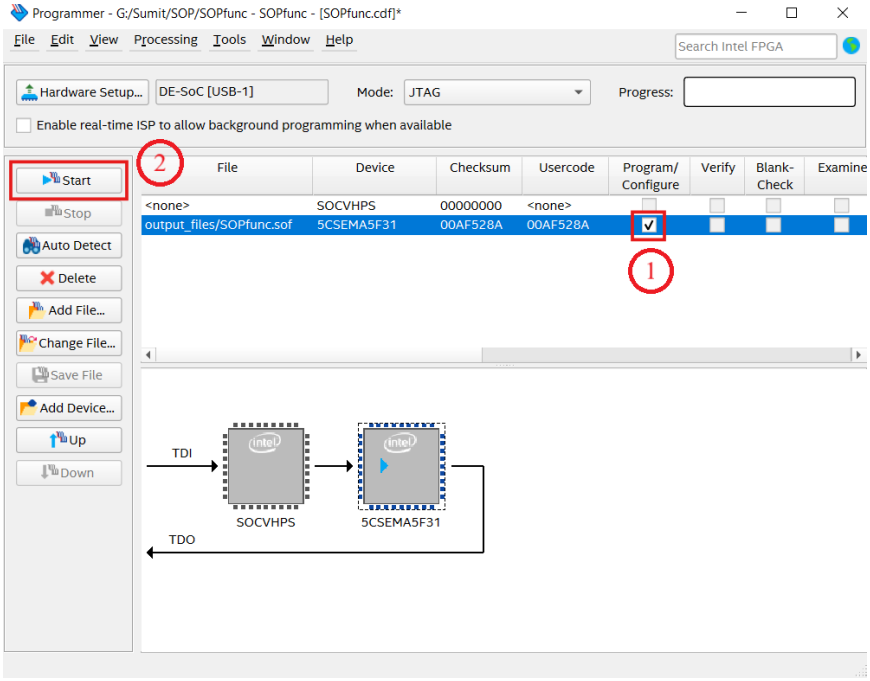
FB12. Open the folder named **output_files**



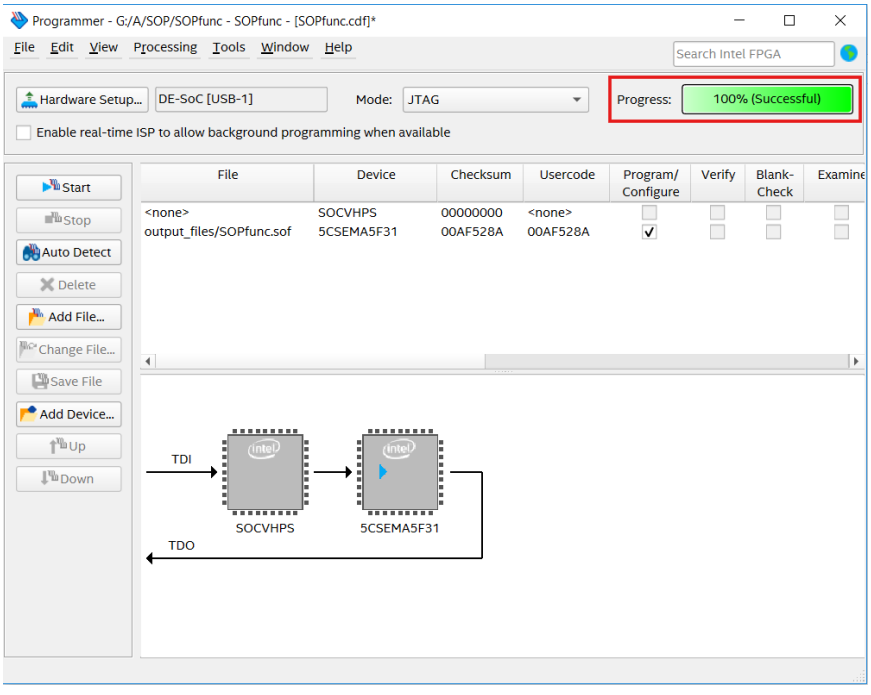
FB13. Select the ***.sof** file to be programmed



FB14. First Click **Program/Configure** check box and then click **Start** button to download the .sof file into the FPGA device.



FB15. If your upload is successful you can see in the Progress window 100% (Successful) message.



Family, Name & Pin Planner List

DE2 FPGA Board

Family	Cyclone II
Name	EP2C35F672C6

Switch	
SW0	PIN_N25
SW1	PIN_N26
SW2	PIN_P25
SW3	PIN_AE14
SW4	PIN_AF14
SW5	PIN_AD13
SW6	PIN_AC13
SW7	PIN_C13

LED	
LEDR0	PIN_AE23
LEDR1	PIN_AF23
LEDR2	PIN_AB21
LEDR3	PIN_AC22
LEDR4	PIN_AD22

Button	
KEY[0]	PIN_G26
KEY[1]	PIN_N23
KEY[2]	PIN_P23
KEY[3]	PIN_W26

7 Segment Display	
HEX0[0]	PIN_AF10
HEX0[1]	PIN_AB12
HEX0[2]	PIN_AC12
HEX0[3]	PIN_AD11
HEX0[4]	PIN_AE11
HEX0[5]	PIN_V14
HEX0[6]	PIN_V13

DE1 SoC FPGA Board

Family	Cyclone V
Name	5CSEMA5F31C6

Slide Switch	
SW0	PIN_AB12
SW1	PIN_AC12
SW2	PIN_AF9
SW3	PIN_AF10
SW4	PIN_AD11
SW5	PIN_AD12
SW6	PIN_AE11
SW7	PIN_AC9

LED	
LEDR0	PIN_V16
LEDR1	PIN_W16
LEDR2	PIN_V17
LEDR3	PIN_V18
LEDR4	PIN_W17

Push Button	
KEY[0]	PIN_AA14
KEY[1]	PIN_AA15
KEY[2]	PIN_W15
KEY[3]	PIN_Y16

7 Segment Display	
HEX0[0]	PIN_AE26
HEX0[1]	PIN_AE27
HEX0[2]	PIN_AE28
HEX0[3]	PIN_AG27
HEX0[4]	PIN_AF28
HEX0[5]	PIN_AG28
HEX0[6]	PIN_AH28

Hardware Description Language (HDL)

A hardware description language (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit.

There are two major Hardware Description languages –

- VHDL
- Verilog

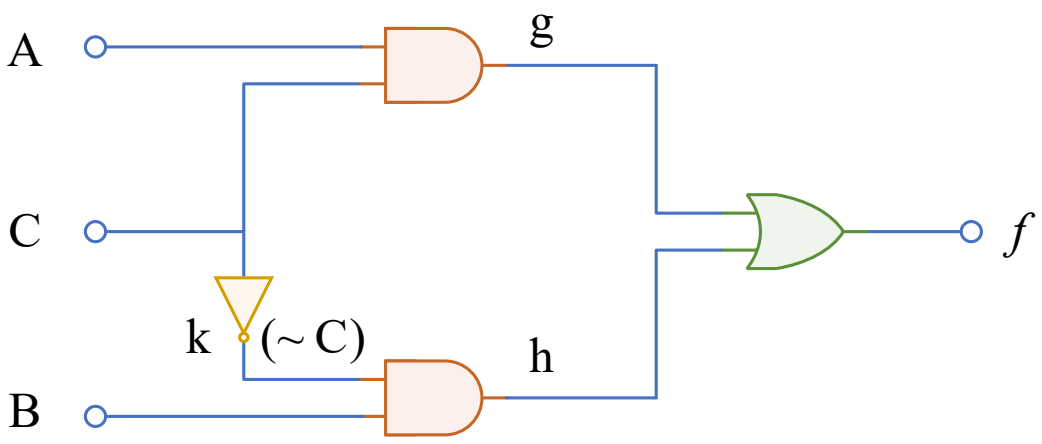
We will focus on Verilog HDL for our Lab. In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design Systems.

There are three distinct ways we can write Verilog Codes.

- Structural
- Data Flow
- Behavioral

In this Lab, we will learn these different ways of writing Verilog Code by implementing the following function:

$$f = AC + BC'$$



Structural

Basic

```
/*
        This Verilog Code implements the Sum of Product (SOP) function using
        Structural representation which is also known as Gate level modeling.
*/
module SOPfunc (A, B, C, f);
        input A, B, C;
        output f;
        wire g, h, k; // Optional Declaration

        and (g, A, C);
        not (k, C);
        and (h, B, k);
        or (f, g, h);

endmodule
```

Simplified

```
module SOPfunc (input A, B, C, output f);

        and (g, A, C);
        and (h, B, ~C);
        or (f, g, h);

endmodule
```

Data Flow

Basic

```
module SOPfunc (A, B, C, f);
    input A, B, C;
    output f;
    wire g, h, k;

    assign g = A & C;
    assign k = ~C;
    assign h = B & k;
    assign f = g | h;

endmodule
```

Simplified

```
module SOPfunc (A, B, C, f);
    input A, B, C;
    output f;
    wire g, h;

    assign g = A & C;
    assign h = B & ~C;
    assign f = g | h;

endmodule
```

Further Simplified

```
module SOPfunc (A, B, C, f);
    input A, B, C;
    output f;

    assign f = (A & C) | (B & ~C);

endmodule
```

Experiment: 4

Experiment name: *Introduction to Adder Circuit.*

Introduction:

Adders are the basic operational units of simple digital arithmetic operations. In this experiment, the students will construct the basic adder circuit with common logic gates and test their operability. Then in the last job, they will cascade adder ICs to get higher bit adders.

Binary Adder

Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry.* When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half-adder.* One that performs the addition of three bits (two significant bits and a previous carry) is *full adder.*

Half Adder

From the basic understanding of a half-adder, we find that the circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. It is necessary to specify two output variables because the result may consist of two binary digits. We arbitrarily assign symbols A and B to the two inputs and S (for sum) and C (for carry) to the outputs. So, we have established the number and names of the input and output variables, we are ready to formulate a truth table to identify exactly the function of the half-adder. This truth table is –

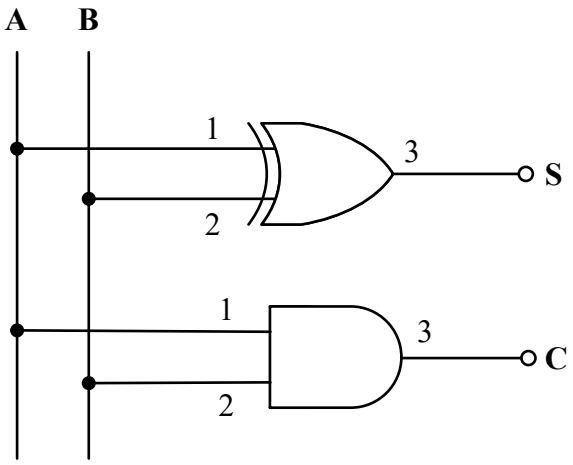
Input		**Output**	
A	**B**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum. The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are –

$$S = A'B + AB' = A \oplus B$$

$$C = AB$$

The logic diagram for this implementation is shown below –



Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by A and B, represent the two significant bits to be added. The third input, C_{in}, represents the carry-in from the previously significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S for sum and C_{out} for carry-out. The binary variable S gives the value of the least significant bit of the sum. The binary variable C_{out} gives the output carry. The truth table of the full adder is

Input			**Output**	
A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

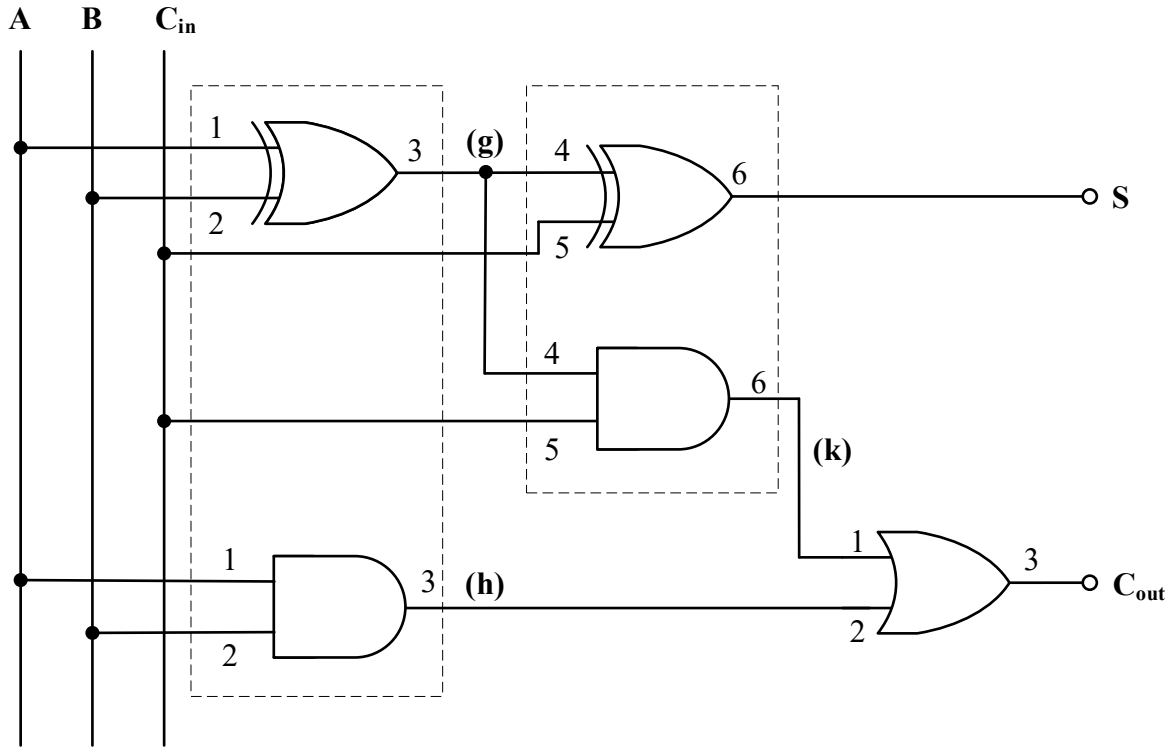
The eight rows under the input variables designate all possible combinations of 1's and 0's that these variables may have. The 1's and 0's for the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0's, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1. Physically, the binary signals of the input wires are considered binary digits added arithmetically to form a two-digit sum at the output wires. On

the other hand, the same binary values are considered variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. It is important to realize that two different interpretations are given to the values of the bits encountered in this circuit. The input-output logical relationship of the full-adder circuit may be expressed in two Boolean functions, one for each output variable. This implementation uses the following Boolean expressions:

$$\begin{aligned} S &= A'B'C_{in} + A'BC_{in}{}' + AB'C_{in}{}' + ABC_{in} \\ &= (A'B + AB')C_{in}{}' + (A'B' + AB)C_{in} \\ &= (A \oplus B)C_{in}{}' + (A \oplus B)'C_{in} \\ &= A \oplus B \oplus C_{in} \end{aligned}$$

$$\begin{aligned} C_{out} &= A'BC_{in} + AB'C_{in} + ABC_{in}{}' + ABC_{in} \\ &= (A'B + AB')C_{in} + AB(C_{in}{}' + C_{in}) \\ &= (A \oplus B)C_{in} + AB \end{aligned}$$

The logic diagram for the full adder implemented in sum of products is shown here.



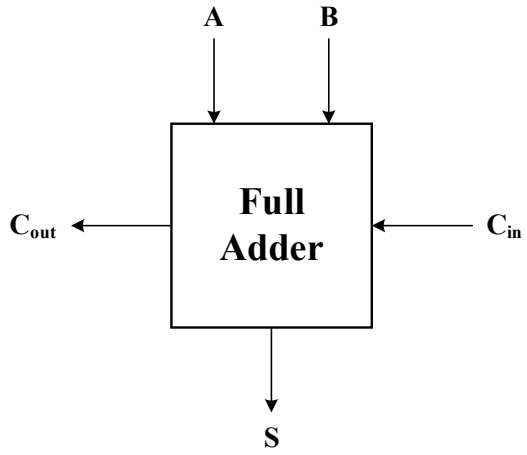
Ripple Carry Adder

A Ripple Carry Adder (RCA) is the simplest form of a binary adder used in digital electronics to add two binary numbers. It has a very simple and intuitive structure. An RCA is built using a chain of full adders (FAs). Each full adder adds two bits (A & B) along with a carry-in (Cin) and produces: a sum (S) bit a carry-out (Cout). The carry-out of one stage **ripples** into the carry-in of the next stage, hence the name. The advantages of RCA is that it is very simple to design and implement as it requires only full adders connected in series. In this experiment, we will design an RCA which can add two 2-bit binary numbers. Two 2-bit numbers A (A2 A1) & B (B2 B1) will be added resulting 3-bit sum result (Cout S2 S1). For example, $(3)_{10} + (1)_{10} = (11)_2 + (01)_2$ will generate $(4)_{10} = (100)_2$. We will design an RCA which perform this type of operations.

	A		3			A2	A1		1	1
+	B	+	1		+	B2	B1	+	0	1
Sum			4		Cout	S2	S1	1	0	0

A Full Adder is a combinational circuit designed to perform the arithmetic addition of three input bits: A, B, and Cin (carry-in). It generates two outputs, where the first output is the sum bit (S), which represents the result of adding the three inputs in binary form, and the second output is the carry-out bit (Cout), which indicates whether a carry is produced from the addition that must be passed on to the next higher stage in multi-bit addition. This makes the Full Adder an essential building block for constructing adders capable of handling numbers larger than a single bit.

	A
	B
+	C_{in}
C_{out}	S

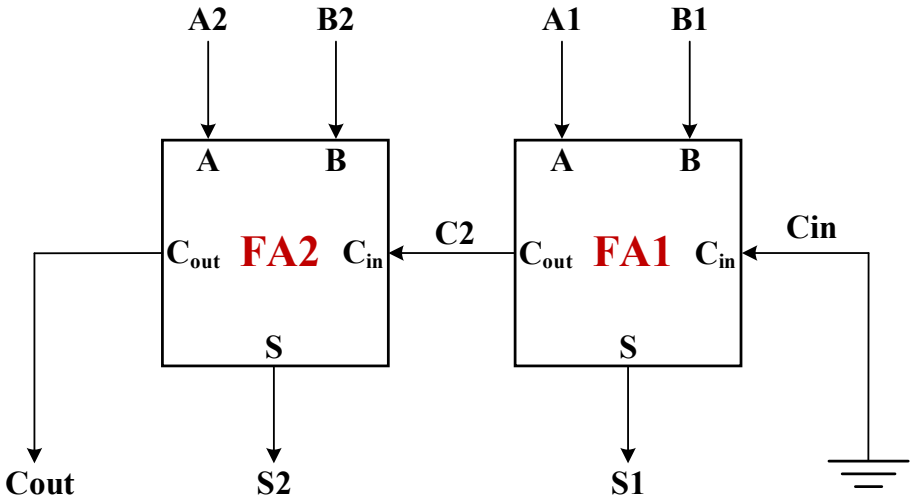


A 2-bit Ripple Carry Adder (RCA) is constructed using two Full Adders (FA). In the first stage, the Full Adder (FA1) adds the least significant bits A1 and B1 along with an initial carry-in (Cin) of 0, producing a sum bit S1 and a carry-out C2. This carry-out then **ripples** into the second stage Full Adder (FA2) as its carry-in, where the most significant bits A2 and B2 are added together with C2, producing the sum bit S2 and the final carry-out Cout. The complete result of the addition is represented by the three output bits (Cout, S2, S1), which together form the binary sum of the two 2-bit inputs.

	Stage 2 (FA2)	Stage 1 (FA1)
	A2	A1
	B2	B1
+	C2	*Cin*
Cout	**S2**	**S1**

	Stage 2 (FA2)	Stage1 (FA1)
	1	1
	0	1
+	1	0
1	**0**	**0**

Here is the block diagram of a 2-bit Ripple Carry Adder (RCA) used to add two 2-bit numbers, where the first stage consists of a Full Adder (FA1) that adds the least significant bits A1 and B1 with the carry-in grounded (Cin = 0) since no carry is present at the beginning, and its carry-out (C2) is then passed as the carry-in to the second stage Full Adder (FA2), which performs the addition of the most significant bits A2 and B2, thereby completing the overall addition process.



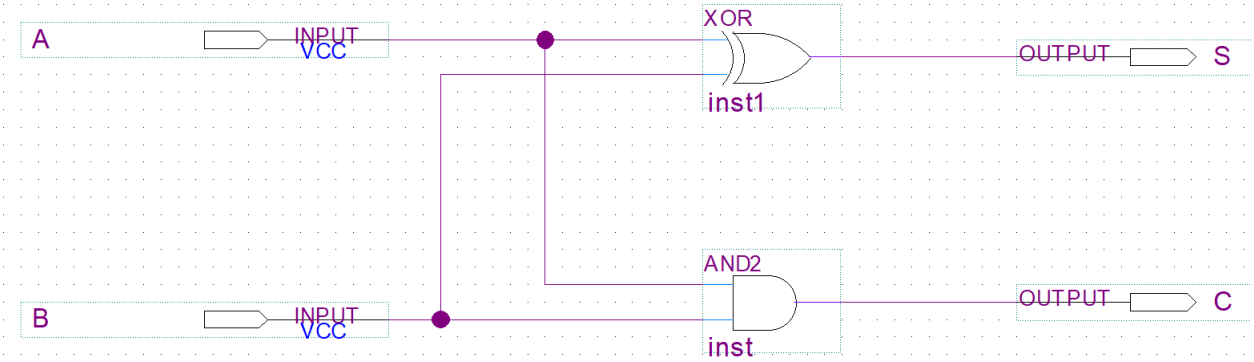
❖ **Here is the summary of the operation of 2-bit Ripple Carry Adder –**

FA1 (Least Significant Bit → First Stage Full Adder) • Inputs → A1, B1, Cin • Outputs → C2, S1 • A1 + B1 + Cin → produces C2 & S1	**Stage 1: FA1 (LSB)** • A1 = 1, B1 = 1, Cin = 0 • C2 = 1, S1 = 0
FA2 (Most Significant Bit → Second Stage Full Adder) • Inputs → A2, B2, C2 • Outputs → Cout, S2 • A2 + B2 + C2 → produces Cout & S2	**Stage 2: FA2 (MSB)** • A2 = 1, B2 = 0, C2 = 1 • Cout = 1, S2 = 0

Procedure for FPGA

Half Adder:

1. Create a new project and create a new block diagram/schematic or verilog file.



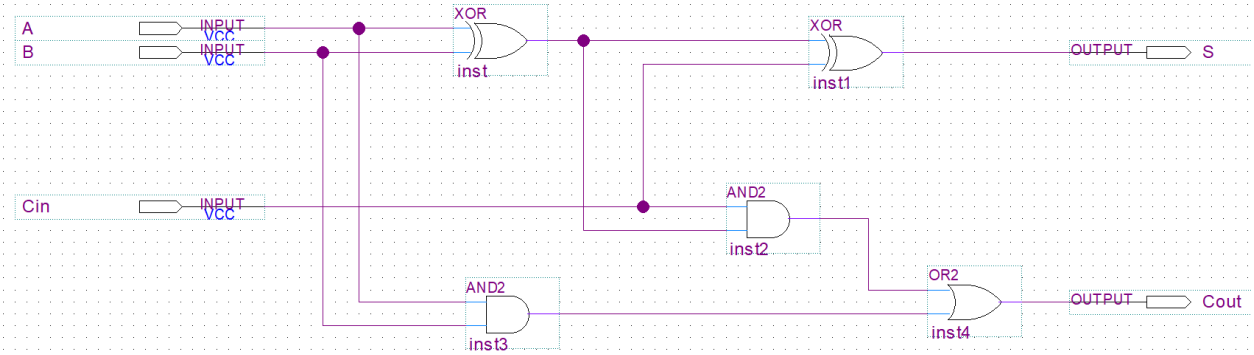
2. Compile and simulate the schematic. If everything is ok, assign pins as follows –

Signal	Name	Cyclone II	Cyclone V
A	SW1	PIN_N26	PIN_AC12
B	SW0	PIN_N25	PIN_AB12
C	LEDR1	PIN_AF23	PIN_W16
S	LEDR0	PIN_AE23	PIN_V16

3. Verify the outputs in FPGA board.

Full Adder:

1. Create a new project and create a new block diagram/schematic or verilog file.



2. Compile and simulate the schematic. If everything is ok, assign pins as follows –

Signal	Name	Cyclone II	Cyclone V
A	SW2	PIN_P25	PIN_AF9
B	SW1	PIN_N26	PIN_AC12
Cin	SW0	PIN_N25	PIN_AB12
Cout	LEDR1	PIN_AF23	PIN_W16
S	LEDR0	PIN_AE23	PIN_V16

3. Verify the outputs in FPGA board.

Creating a Symbol from Design File & Hierarchical Design

Today we will learn how we can create module for our design file and then use that file for hierarchical design for another project. We will use Full Adder Design file to create Ripple Carry Adder.

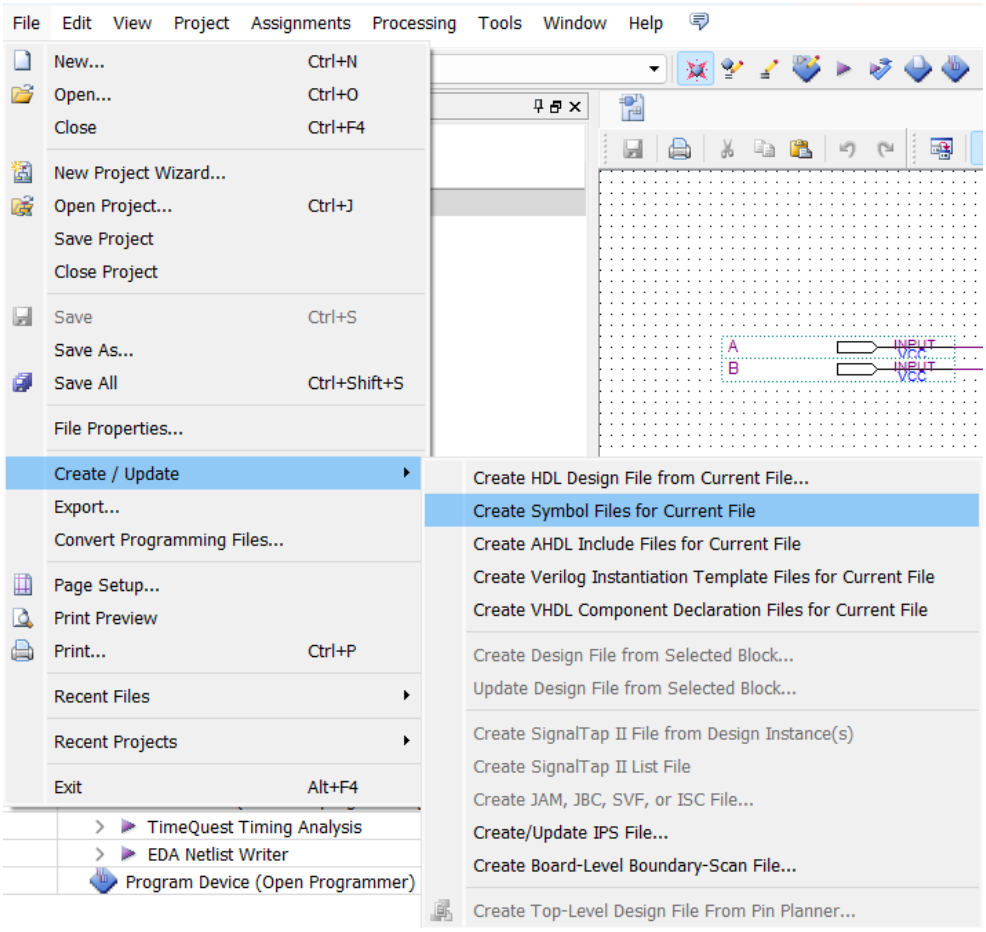
❖ Full Adder Project

1. First, we will create a Block Symbol File (*.bsf) for Full Adder. We can do it from any design file –

   (a) Block Diagram (*.bdf) File

   (b) Verilog (*.v) File

2. For that, go to ***File → Create/Update → Create Symbol Files for Current File***.



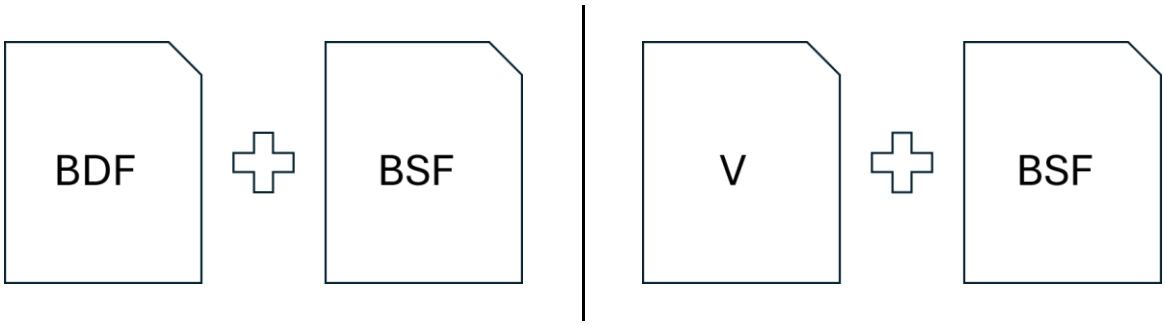
3. Block Symbol File (*.bsf) will be created and saved in the project directory for Full Adder.

❖ Ripple Carry Adder Project

4. Open a new empty folder for Ripple Carry Adder Project. Copy the design file (**Block Diagram** or **Verilog**) and the **Block Symbol** file from *Full Adder* project and paste it in this newly created empty folder for *Ripple Carry Adder* project.

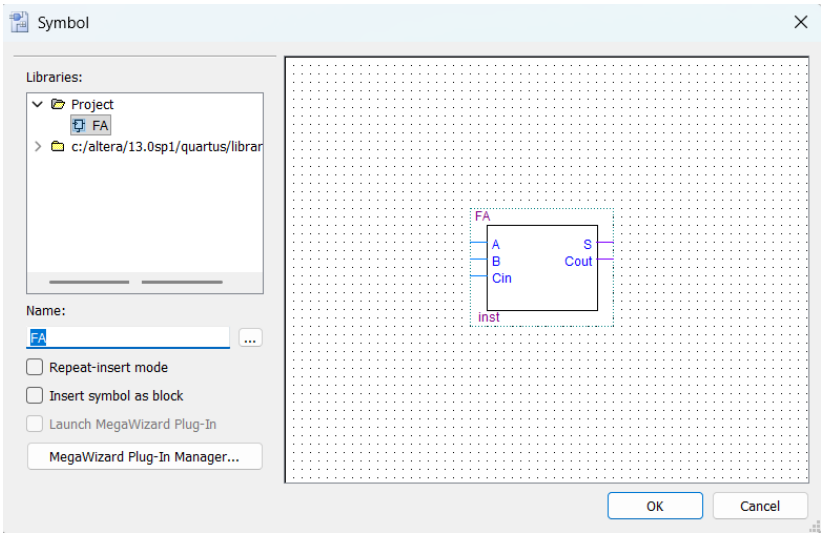
   (a) Block Diagram File (*.bdf) & Block Symbol File (*.bsf)

   (b) Verilog File (*.v) & Block Symbol File (*.bsf)

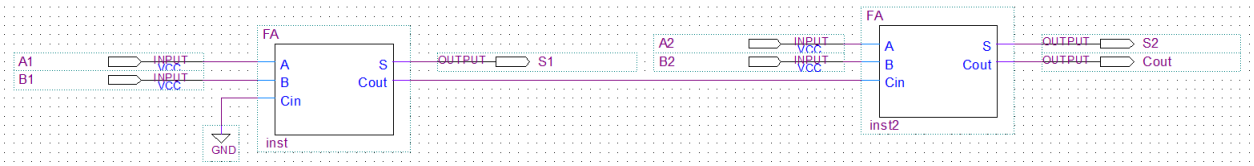


5. Open a Quartus project in this folder for ripple carry adder. After creating the project, go to **File → New** and select Block Diagram/Schematic file.

6. Open Symbol Tool. There will be a new Folder name **Project** under this you will see the Block Symbol File for your Full Adder.



7. Complete the schematic for Ripple Carry Adder.



8. Compile and if no error occurs then assign pins according to the following table –

Signal	Name	Cyclone II	Cyclone V
A1	SW2	PIN_P25	PIN_AF9
A2	SW3	PIN_AE14	PIN_AF10
B1	SW0	PIN_N25	PIN_AB12
B2	SW1	PIN_N26	PIN_AC12
Cout	LEDR2	PIN_AB21	PIN_V17
S1	LEDR0	PIN_AE23	PIN_V16
S2	LEDR1	PIN_AF23	PIN_W16

9. Verify the following table –

Decimal Values		
A	**B**	**A + B**
0	0	0
0	1	1
0	2	2
0	3	3
1	1	2
1	2	3
1	3	4
2	2	4
2	3	5
3	3	6

Input (A)	
A2	**A1**
0	0
0	0
0	0
0	0
0	1
0	1
0	1
1	0
1	0
1	1

Input (B)	
B2	**B1**
0	0
0	1
1	0
1	1
0	1
1	0
1	1
1	0
1	1
1	1

Output Sum		
Cout	**S2**	**S1**
0	0	0
0	0	1
0	1	0
0	1	1
0	1	0
0	1	1
1	0	0
1	0	0
1	0	1
1	1	0

Verilog Code

Half Adder (Structural)

```
module HalfAdder (A, B, S, C);
        input A, B;
        output S, C;

        xor (S, A, B);
        and (C, A, B);
endmodule
```

Half Adder (Data Flow)

```
module HalfAdder (A, B, S, C);
        input A, B;
        output S, C;

        assign S = A ^ B;
        assign C = A & B;
endmodule
```

Half Adder (Behavioral)

```
module HalfAdder (A, B, S, C);
        input A, B;
        output reg S, C;

        always @ (A, B) // always @ *
                {C, S} = A + B;
endmodule
```

Full Adder (Structural)

```
module FullAdder (A, B, Cin, S, Cout);
        input A, B, Cin;
        output S, Cout;
        wire g, h, k;

        // Sum (S)
        xor (g, A, B);
        xor (S, g, Cin);

        // Carryout(Cout)
        and (h, A, B);
        and (k, g, Cin);
        or (Cout, h, k);
endmodule
```

Full Adder (Data Flow)

```
module FullAdder (A, B, Cin, S, Cout);
        input A, B, Cin;
        output S, Cout;

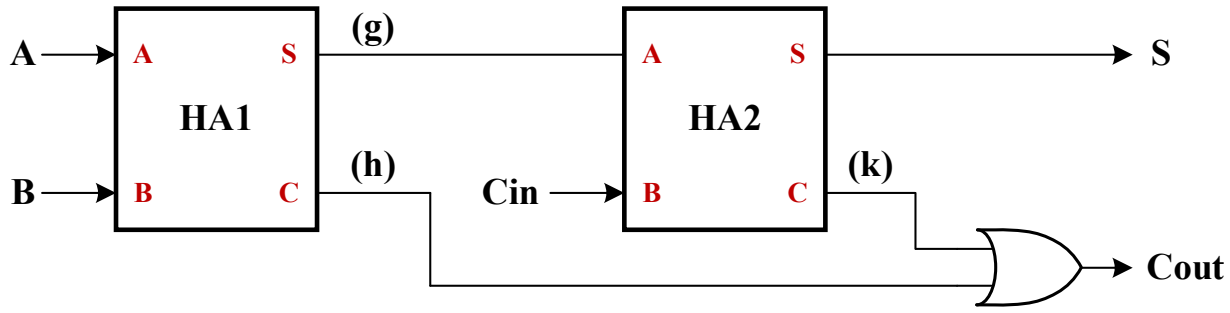
        assign S = A ^ B ^ Cin;
        assign Cout = ((A ^ B) & Cin) | (A & B);
endmodule
```

Full Adder (Behavioral)

```
module FullAdder (A, B, Cin, S, Cout);
        input A, B, Cin;
        output reg S, Cout;

        always @ (A, B, Cin)
                {Cout, S} = A + B + Cin;
endmodule
```

Full Adder from Half Adder




```
// Full Adder (Top Level Hierarchy)

module FA (A, B, Cin, S, Cout);
        input A, B, Cin;
        output S, Cout;
        wire g, h, k;

        HA HA1 (A, B, g, h);
        HA HA2 (g, Cin, S, k);
        or (Cout, h, k);  // assign Cout = h | k;
endmodule


// Half Adder

module HA (A, B, S, C);
        input A, B;
        output reg S, C;

        always @ (A, B)
                {C, S} = A + B;
endmodule
```

Experiment: 5

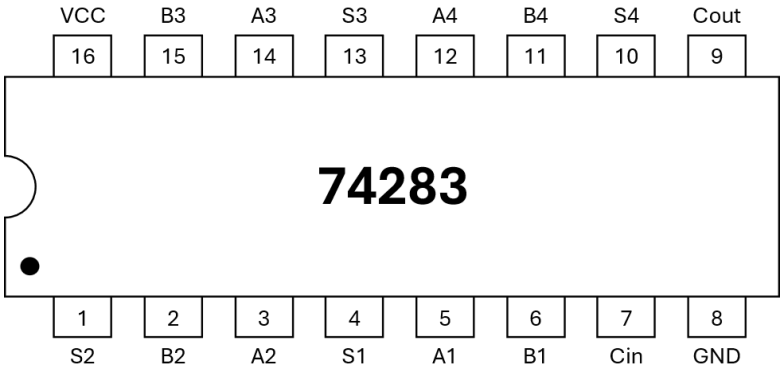
Experiment name: *Introduction to Binary & BCD Adder (4-bit Version)*

4-bit Binary Adder:

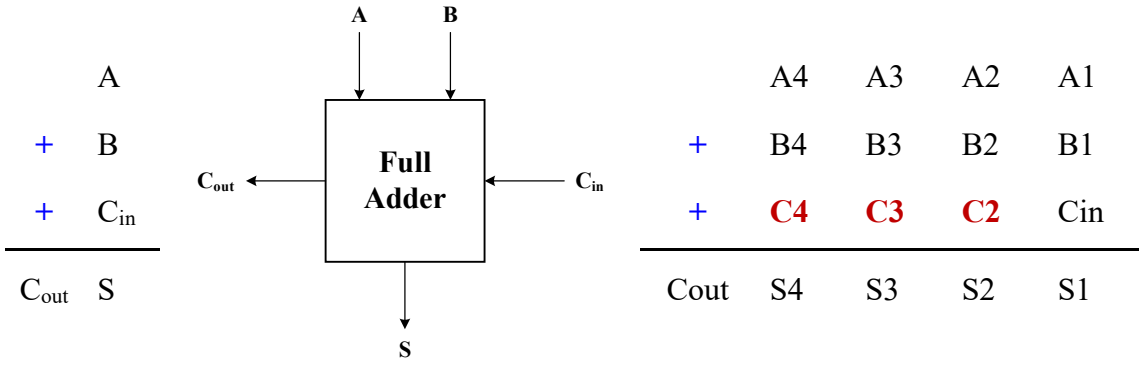
We will implement a 4-bit Banary Adder using **IC 74283**. The 74283 is a 4-bit binary full adder from the 74xx TTL logic IC family. It performs 4-bit binary addition of two numbers (A and B) with a carry-in input (C_{in}) and produces a 4-bit sum (S) and a carry-out (C_{out}). It's basically four full adders cascaded inside one chip. It performs as ripple carry adder for 4-bit operation. The reason we need 4-bit operation because Decimal digits (0 – 9) needs maximum 4-bits. So, having a 4-bit adder directly supports BCD arithmetic, which is very common.

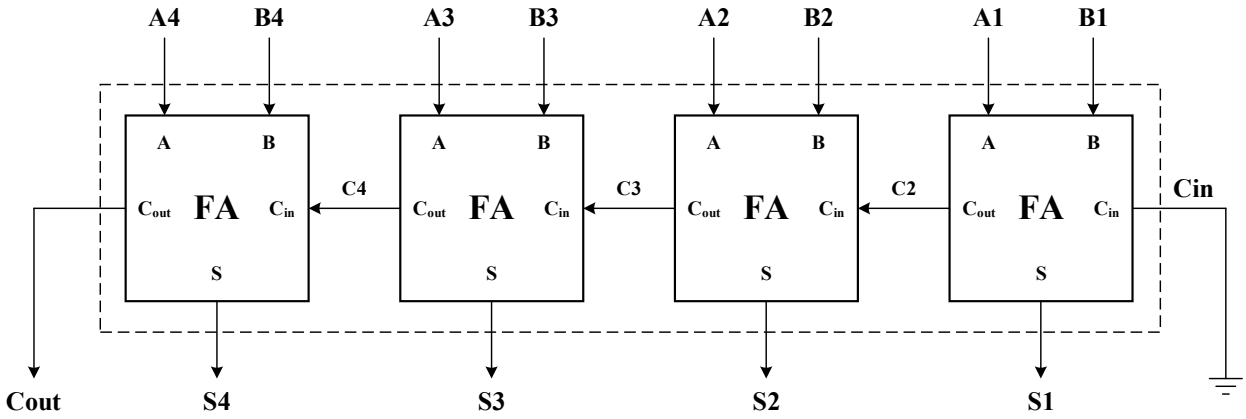
	A		A4	A3	A2	A1
+	B	+	B4	B3	B2	B1
Cout	S	Cout	S4	S3	S2	S1

We are going to add two four-bit Numbers **A** & **B**. Here, **A** = $A_4\ A_3\ A_2\ A_1$ & **B** = $B_4\ B_3\ B_2\ B_1$ (where, 4 → MSB, 1→ LSB) and summation would yield 5 bits output. $\mathbf{C_{out}}$ (Carryout) bit would be the MSB followed by 4 sum bits $S_4\ S_3\ S_2\ S_1$ in that order (**Sum** = $C_{out}\ S_4\ S_3\ S_2\ S_1$). Now, if necessary, an extra $\mathbf{C_{in}}$ (Carry-in) bit can be added to sum of **A** & **B** resulting in Summation of **A** + **B** + $\mathbf{C_{in}}$. However, for first stage $\mathbf{C_{in}}$ would be zero so it would be grounded. However, if we want to design an 8-bit adder than $\mathbf{C_{out}}$ from previous stage would be connected to $\mathbf{C_{in}}$ of next stage. So, this carry-out ripples from one IC to next IC as carry-in input.



IC 74283 incorporates all these inputs and outputs along **VCC** & **GND** resulting in a 16 pin IC. There are 9 input pins – 4 pins of input **A**, 4 pins for input **B** and one pin for $\mathbf{C_{in}}$. There are 5 output pins – one pin for $\mathbf{C_{out}}$ and 4 pins for **S**.





Check the outputs for following Table –

Input		Output
A	*B*	*A + B*
2	3	05
5	7	12
8	9	17
12	15	27
10	0	10
4	14	18
11	13	24
6	1	07
15	15	30

Input (A)			
A4	*A3*	*A2*	*A1*
0	0	1	0
0	1	0	1
1	0	0	0
1	1	0	0
1	0	1	0
0	1	0	0
1	0	1	1
0	1	1	0
1	1	1	1

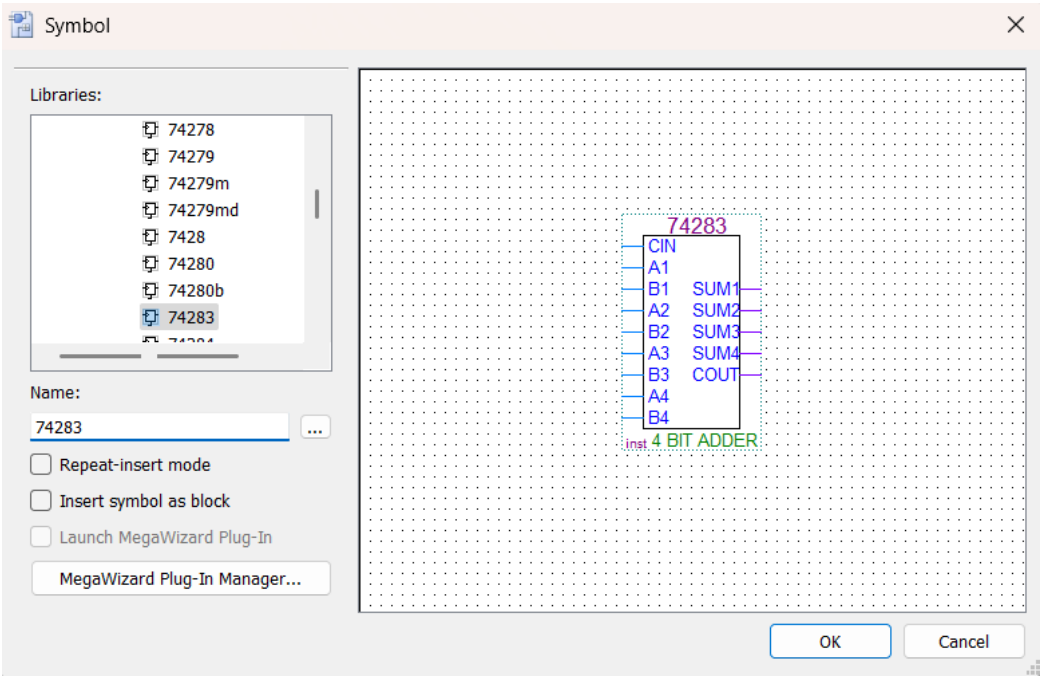
Input (B)			
B4	*B3*	*B2*	*B1*
0	0	1	1
0	1	1	1
1	0	0	1
1	1	1	1
0	0	0	0
1	1	1	0
1	1	0	1
0	0	0	1
1	1	1	1

Output Sum (A + B)				
Cout	*S4*	*S3*	*S2*	*S1*
0	0	1	0	1
0	1	1	0	0
1	0	0	0	1
1	1	0	1	1
0	1	0	1	0
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
1	1	1	1	0

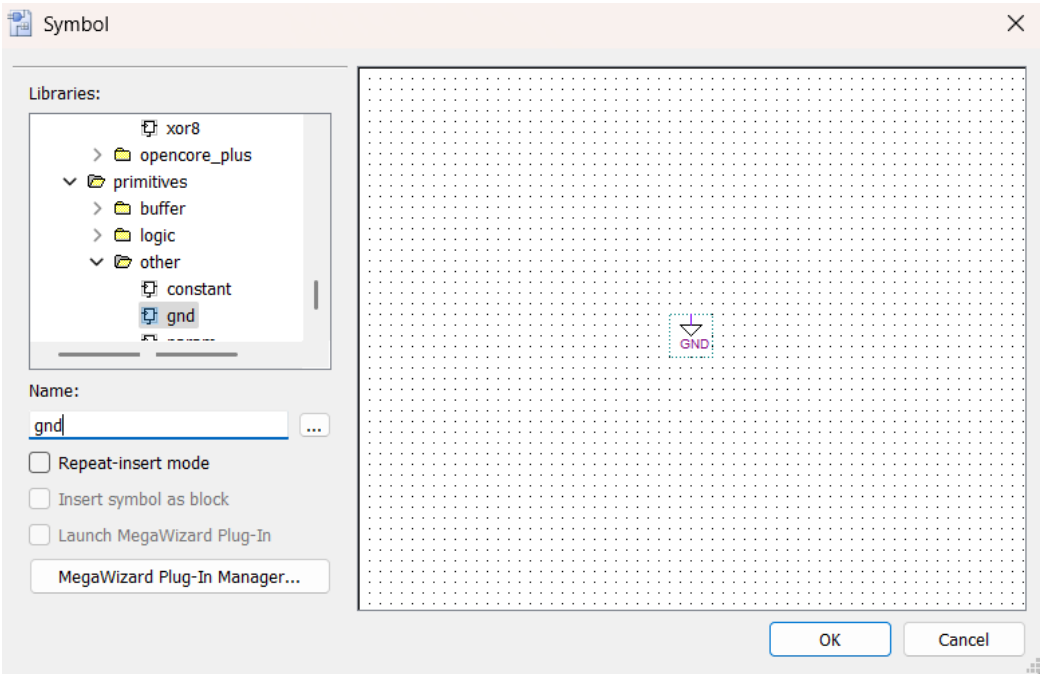
Procedure for FPGA

1. Open a new Empty Folder and create a project inside this. Open a new Block Diagram/Schematic file.

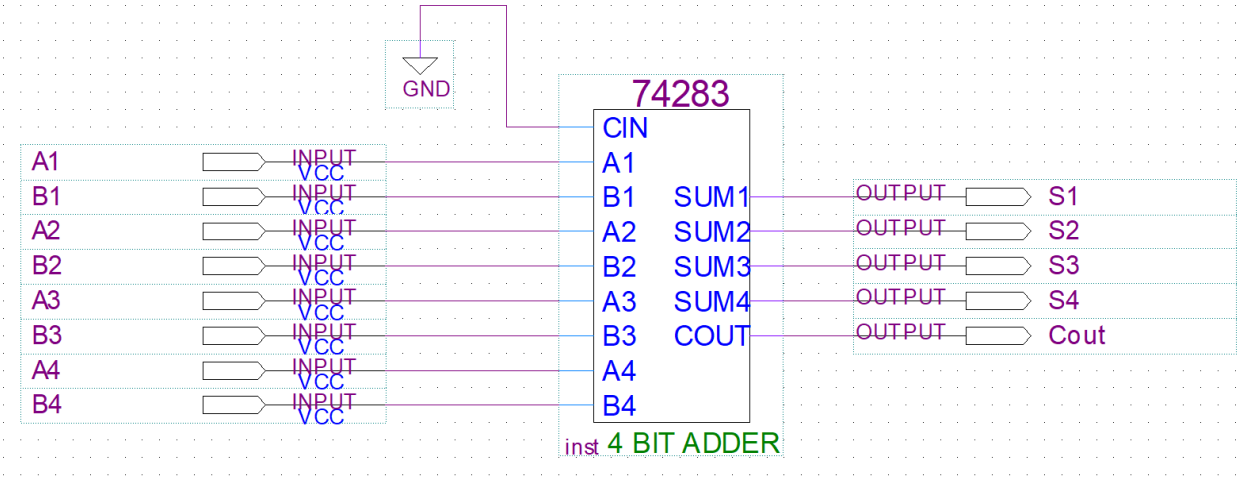
2. Under library directories, type **74283** under ***Name:*** and IC 74283 will appear.



3. Under library directories, type **gnd** under ***Name:*** and ground will appear.



4. Place all other components in the same manner and connect those using wires and add input/output pins.



5. Assign pins as follows and then upload the program onto the FPGA board.

Signal	Switch No.	Cyclone II	Cyclone V
A1	SW4	PIN_AF14	PIN_AD11
A2	SW5	PIN_AD13	PIN_AD12
A3	SW6	PIN_AC13	PIN_AE11
A4	SW7	PIN_C13	PIN_AC9
B1	SW0	PIN_N25	PIN_AB12
B2	SW1	PIN_N26	PIN_AC12
B3	SW2	PIN_P25	PIN_AF9
B4	SW3	PIN_AE14	PIN_AF10
S1	LEDR0	PIN_AE23	PIN_V16
S2	LEDR1	PIN_AF23	PIN_W16
S3	LEDR2	PIN_AB21	PIN_V17
S4	LEDR3	PIN_AC22	PIN_V18
Cout	LEDR4	PIN_AD22	PIN_W17

6. Verify that the observed output at FPGA is the same as the one observed using discrete ICs.

BCD Adder:

Before discussing BCD Adder circuitry first, we can review the basic concepts of BCD no system and BCD addition technique.

BCD

Binary coded decimal (BCD) is a weighted code that is commonly used in many computers and calculators to represent decimal numbers. This code takes each decimal digit and represents it by a four-bit code ranging from 0000 to 1001.

Decimal	Binary	BCD
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010	00010000
11	1011	00010001
12	1100	00010010
13	1101	00010011
14	1110	00010100
15	1111	00010101

The table illustrates the difference between straight binary and BCD. BCD represents each decimal digit with a 4-bit code. Notice that the codes 1010 through 1111 are not used in BCD.

BCD Addition

The addition of decimal numbers that are in BCD form can be best understood by considering the two cases that can occur when two decimal digits are added.

Sum Equals 9 or Less

Consider adding 45 and 33 using BCD to represent each digit:

```
  45        0100 0101   ← BCD for 45
+ 33      + 0011 0011   ← BCD for 33
  78        0111 1000   ← BCD for 78
```

In the examples above, none of the sums of the pairs of decimal digits exceeded 9; therefore, *no decimal carries were produced.* For these cases, the BCD addition process is straightforward and is actually the same as binary addition.

Sum Greater than 9

Consider the addition of 6 and 7 in BCD:

```
 6      0110   ← BCD for 6
+7    + 0111   ← BCD for 7
+13    1101   ← invalid code group for BCD
```

The sum 1101 does not exist in the BCD code; it is one of the six forbidden or invalid four-bit code groups. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs, the sum must be corrected by the addition of six (0110) to take into account the skipping of the six invalid code groups:

```
        0110   ← BCD for 6
      + 0111   ← BCD for 7
        1101   ← invalid sum
        0110   ← add 6 for correction
      ------
 0001   0011   ← BCD for 13
   1      3
```

As shown above, 0110 is added to the invalid sum and produces the correct BCD result. Note that with the addition of 0110, a carry is produced in the second decimal position. This addition must be performed whenever the sum of the two decimal digits is greater than 9.

Consider the addition of 59 and 38 in BCD:

```
          1
 59    0101   1001   ← BCD for 59
+38  + 0011   1000   ← BCD for 38
 97    1001   0001   ← perform addition
               0110   ← add 6 to correct
      ------------
      1001   0111     BCD for 97
        9      7
```

Here, the addition of the least significant digits (LSDs) produces a sum of 17 = 10001**.** This generates a carry into the next digit position to be added to the codes for 5 and 3. Since 17 > 9, a correction factor of 6 must be added to the LSD sum. Addition of this correction does not generate a carry; the carry was already generated in the original addition.

To summarize the BCD addition procedure:

1. Using ordinary binary addition, add the BCD code groups for each digit position.

2. For those positions where the sum is 9 or less, no correction is needed. The sum is in proper BCD form.

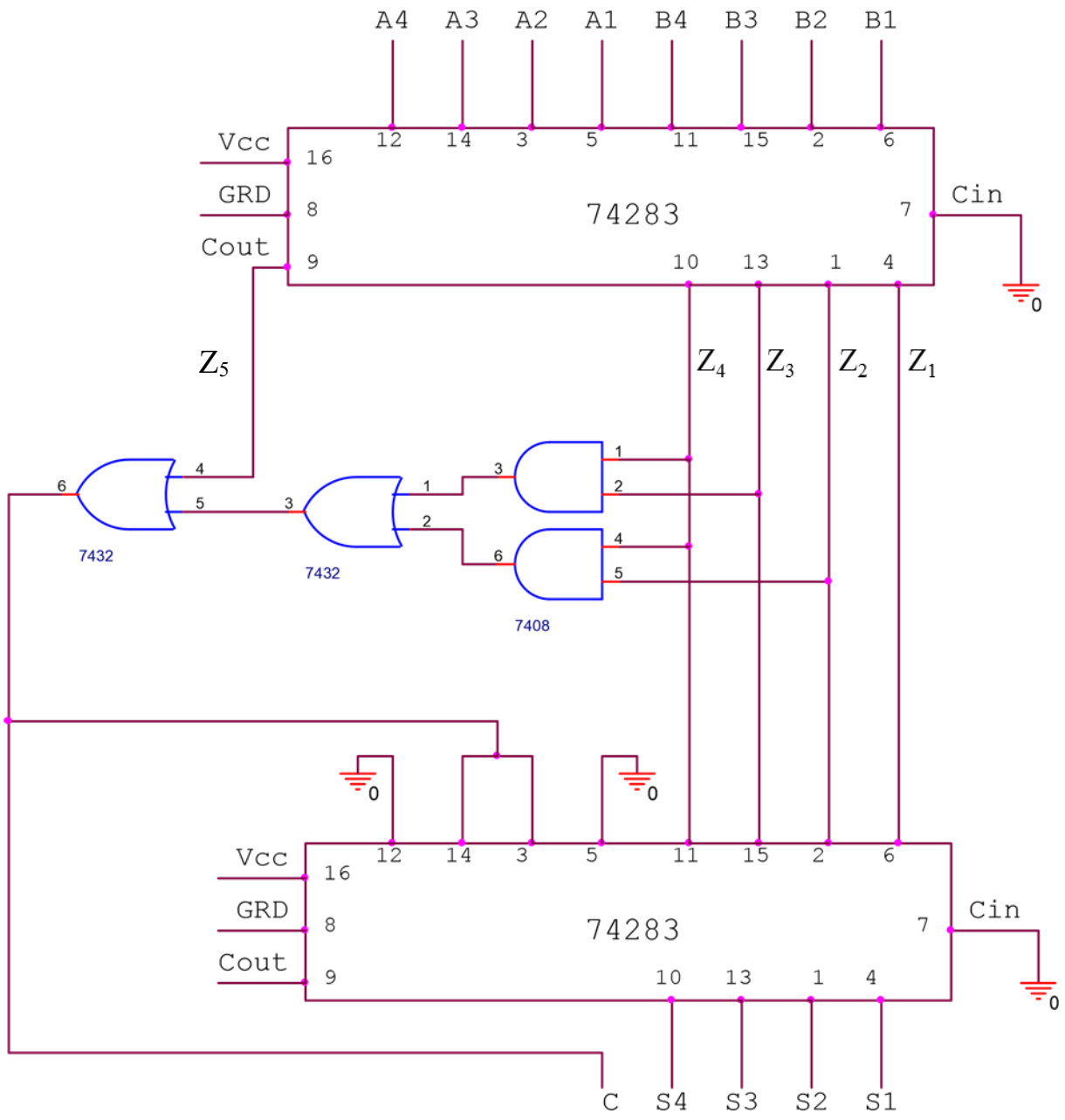
3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum to get the proper BCD result. This case always produces a carry into the next digit position, either from the original addition (step 1) or from the correction addition.

BCD ADDER

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table and are labeled by symbols Z_5, Z_4, Z_3, Z_2, and Z_1. Z_5 is the carry, and the subscripts under the letter Z represent the weights 4, 3, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal digits* must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number, in the first column can be converted to the correct BCD-digit representation of the number in the second column. In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required. The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $Z_5 = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_4. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_4 we specify further that, either Z_3 or Z_2 must have a 1 along with Z_4. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = Z_5 + Z_4Z_3 + Z_4Z_2$$

When C = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage. A *BCD adder* is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder, as shown in Figure. The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal. The BCD adder can be constructed with three IC packages.

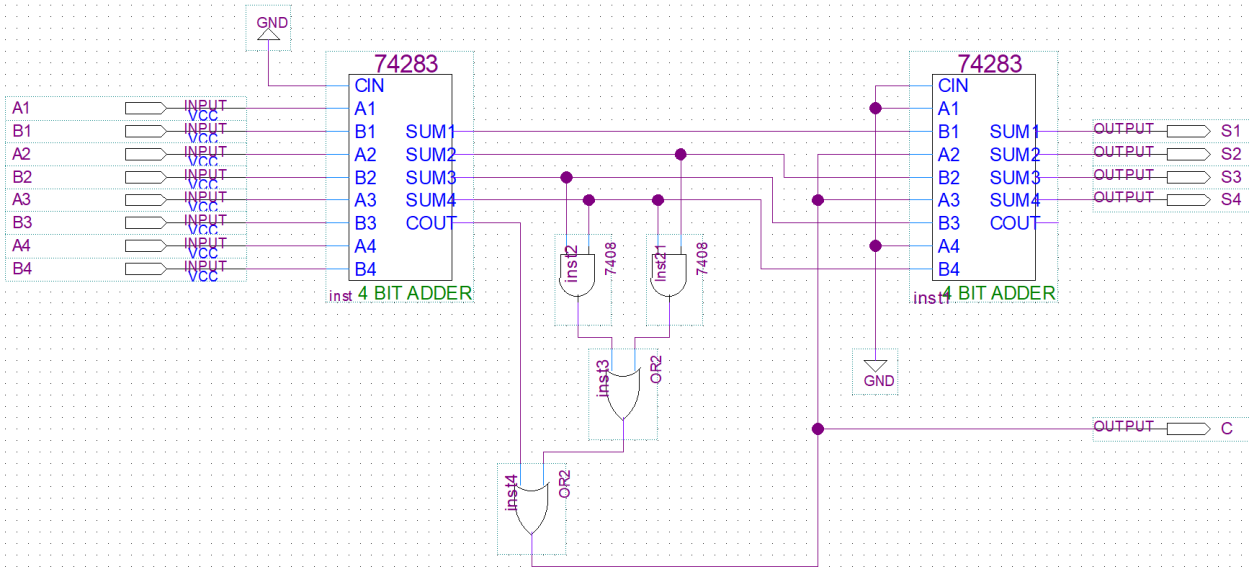

A4
A3
A2
A1
B4
B3
B2
B1
Vcc
GRD
Cout
74283
Cin
Z_5
Z_4
Z_3
Z_2
Z_1
7432
7432
7408
C
S4
S3
S2
S1

	Binary Sum						BCD Sum				
Decimal	Z_5	Z_4	Z_3	Z_2	Z_1		C	S_4	S_3	S_2	S_1
0	0	0	0	0	0		0	0	0	0	0
1	0	0	0	0	1		0	0	0	0	1
2	0	0	0	1	0		0	0	0	1	0
3	0	0	0	1	1		0	0	0	1	1
4	0	0	1	0	0		0	0	1	0	0
5	0	0	1	0	1		0	0	1	0	1
6	0	0	1	1	0		0	0	1	1	0
7	0	0	1	1	1		0	0	1	1	1
8	0	1	0	0	0		0	1	0	0	0
9	0	1	0	0	1		0	1	0	0	1
10	0	1	0	1	0		**1**	**0**	**0**	**0**	**0**
11	0	1	0	1	1		**1**	**0**	**0**	**0**	**1**
12	0	1	1	0	0		**1**	**0**	**0**	**1**	**0**
13	0	1	1	0	1		**1**	**0**	**0**	**1**	**1**
14	0	1	1	1	0		**1**	**0**	**1**	**0**	**0**
15	0	1	1	1	1		**1**	**0**	**1**	**0**	**1**
16	1	0	0	0	0		**1**	**0**	**1**	**1**	**0**
17	1	0	0	0	1		**1**	**0**	**1**	**1**	**1**
18	1	0	0	1	0		**1**	**1**	**0**	**0**	**0**
19	1	0	0	1	1		**1**	**1**	**0**	**0**	**1**

Procedure for FPGA

1. Create a new project and create a new block diagram/schematic file.

2. Compile and simulate the schematic.



3. If everything is ok, assign pins as follows:

Signal	Switch No.	Cyclone II	Cyclone V
A1	SW4	PIN_AF14	PIN_AD11
A2	SW5	PIN_AD13	PIN_AD12
A3	SW6	PIN_AC13	PIN_AE11
A4	SW7	PIN_C13	PIN_AC9
B1	SW0	PIN_N25	PIN_AB12
B2	SW1	PIN_N26	PIN_AC12
B3	SW2	PIN_P25	PIN_AF9
B4	SW3	PIN_AE14	PIN_AF10
C	LEDR4	PIN_AD22	PIN_W17
S1	LEDR0	PIN_AE23	PIN_V16
S2	LEDR1	PIN_AF23	PIN_W16
S3	LEDR2	PIN_AB21	PIN_V17
S4	LEDR3	PIN_AC22	PIN_V18

3. Test the functionality of the designed circuit using switches and LEDs on the FPGA board for the following table –

A (Decimal)	A (Binary)				***B (Decimal)***	B (Binary)				***BCD SUM (Decimal)***	BCD SUM (Binary)				
	A4	A3	A2	A1		B4	B3	B2	B1		C	S4	S3	S2	S1
5	0	1	0	1	**0**	0	0	0	0	**0 5**	0	0	1	0	1
7	0	1	1	1	**4**	0	1	0	0	**1 1**	1	0	0	0	1
2	0	0	1	0	**6**	0	1	1	0	**0 8**	0	1	0	0	0
9	1	0	0	1	**8**	1	0	0	0	**1 7**	1	0	1	1	1
1	0	0	0	1	**3**	0	0	1	1	**0 4**	0	0	1	0	0

Verilog Code

4-bit Binary Adder

```
module BinAdd(A, B, S, Cout);
    input [4:1] A, B; // See *Note*
    output reg [4:1] S; // See *Note*
    output reg Cout;

    always @ (A, B)
        {Cout, S} = A + B;

endmodule
```

Special Note

For simulation (VWF) purposes, change the indexing from 4:1 to 3:0. All other lines should remain unchanged.

```
    input [3:0] A, B;
    output reg [3:0] S;
```

BCD Adder

```
module BcdAdd (A, B, S, C);
        input [4:1] A, B; // See *Note*
        output reg [4:1] S; // See *Note*
        output reg C;
        reg [5:1] Z;

        always @ (A, B)
        begin
                Z = A + B;
                if (Z > 9)
                        {C, S} = Z + 6;
                else
                        {C, S} = Z;
        end

endmodule
```

Special Note

For simulation (VWF) purposes, change the indexing from 4:1 to 3:0. All other lines should remain unchanged.

```
        input [3:0] A, B;
        output reg [3:0] S;
```

Experiment: 6

Experiment name: *Introduction to Multiplexers and Demultiplexer.*

Introduction

Multiplexers are the most important attributions of digital circuitry in communication hardware. These digital switches enable us to achieve the communication network we have today. In this experiment the students will have to construct MUX (as they call multiplexers) with simple logic gates and they will implement general logic using 8:1 MUX as the basic construction unit.

Multiplexer

A modern home stereo system may have a switch that selects music from one of four sources: a cassette tape, a compact disc (CD), a radio tuner, or auxiliary input such as audio from a VCR or DVD. The switch selects one of the electronic signals from one of these four sources and sends it to the power amplifier and speakers. In simple terms, this is what a **multiplexer (MUX)** does: it selects one of several input signals and passes it on to the output.

A digital multiplexer or data selector is a logic circuit that accepts several digital data inputs and selects one of them at any given time to pass on to the output. The routing of the desired data input to the output is controlled by SELECT inputs (often referred to as ADDRESS inputs). Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

Demultiplexer

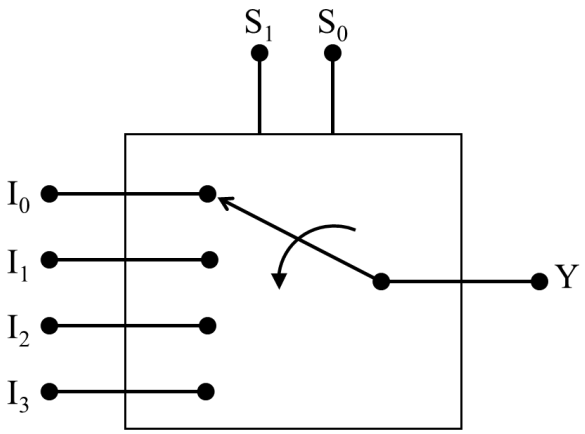
A Demultiplexer does the opposite function of multiplexers. A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by the bit values of n selection lines. The output channel can be selected depending on the combination of selection bits.

Job 1:

Implementation of a four to one way Multiplexer (4:1 MUX) with basic gates.

A **4-to-1 multiplexer (4:1 MUX)** has four data inputs $I_0 – I_3$, two select lines S_1 and S_0, and one output Y. The select lines determine which one of the four inputs is connected to the output at any time. Thus, the multiplexer allows multiple input signals to share a single output line based on the control provided by the selector inputs.

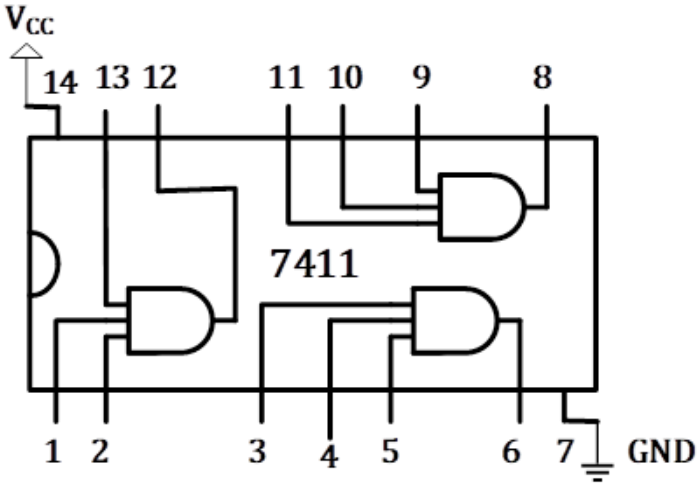
Block Diagram and Truth Table of a 4:1 MUX are shown below.



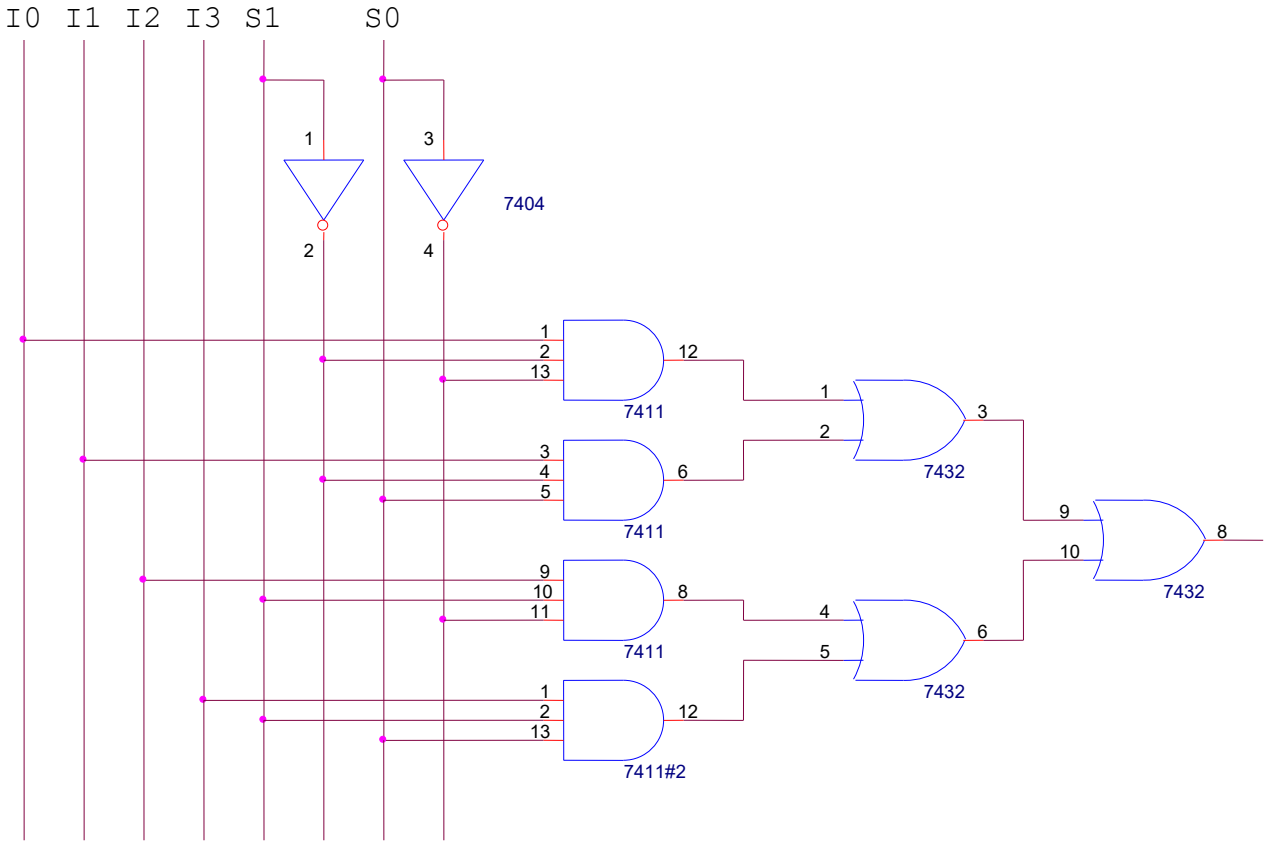
Selector		Output
S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

$$Y = \overline{S}_1\overline{S}_0.I_0 + \overline{S}_1S_0.I_1 + S_1\overline{S}_0.I_2 + S_1S_0.I_3$$

For implementing a 4-to-1 multiplexer, three-input AND gate IC 7411 is required. The pin configuration of the IC 7411 used in this circuit is shown below.



A 4 to l line multiplexer is shown in Figure. Each of the four input lines, I_0 to I_3 is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The function table, Figure lists the input-to-output path for each possible bit combination of the selection lines. To demonstrate the circuit operation, consider the case when S_1S_0 = 10. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2 thus providing a path from the selected input to the output.

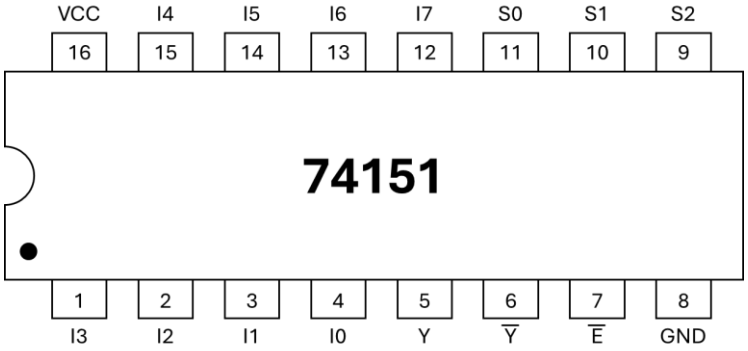


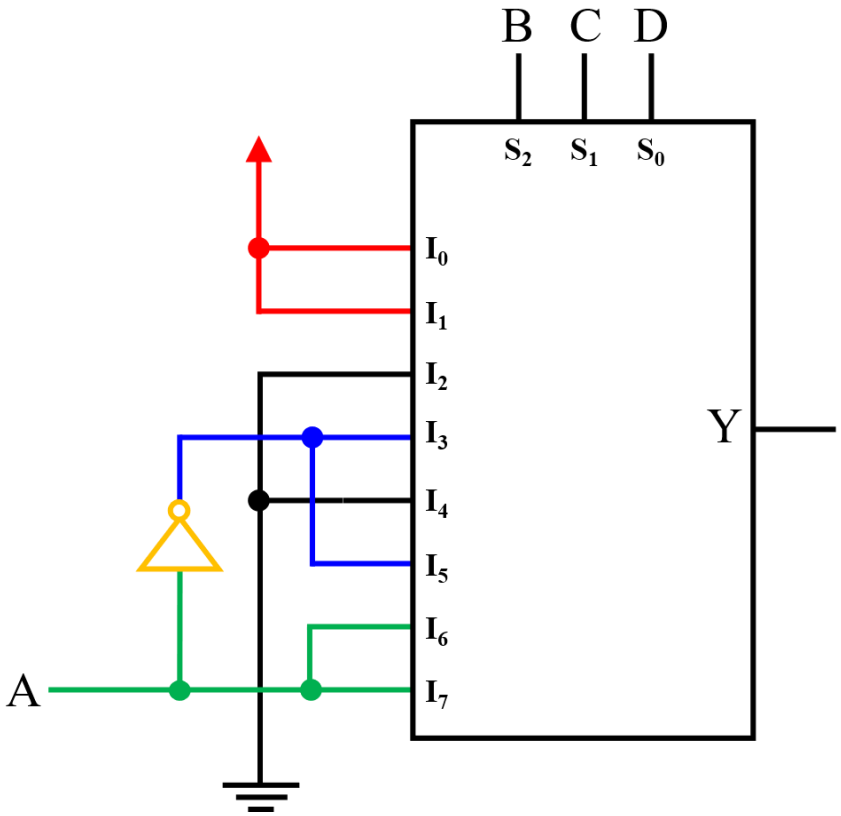
Job 2:

Implement the following function using an 8:1 MUX.

$$F(A,B,C,D)=\sum m(0,1,3,5,8,9,14,15)$$

If we have a Boolean function of $n + 1$ variables, we take n of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the selection lines, one can implement any Boolean function with a multiplexer. In this way, it is possible to generate any function of $n + 1$ variables with a 2^n to1 multiplexer.





Procedure:

1. Write the truth table for the above function.

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Let B, C, D of the 4 variables (A, B, C, D) connected to the selection lines of a multiplexer and remaining single variable A of the function is used for the inputs of the multiplexer.

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	1	1		1		1		
A	1	1					1	1
	1	**1**	**0**	A'	**0**	A'	A	A

2. Draw the logic diagram to implement the Boolean function.
3. Select ICs from the equipment list.
4. Observe and note the output logic for all combinations of inputs.

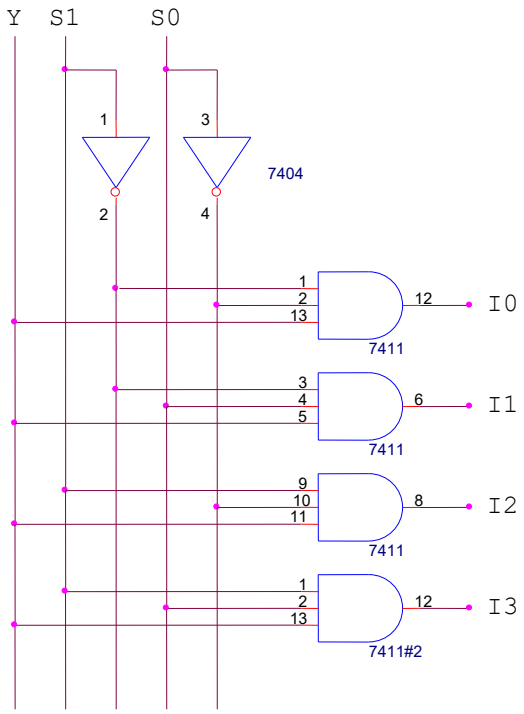
Job 3:

Implementation of a 1 to 4 Demultiplexer (1:4 DEMUX) with basic gates.

Procedure:

1. Write the truth table for one-to-four-way DEMUX.

S_1	S_0	I_3	I_2	I_1	I_0
0	0	0	0	0	Y
0	1	0	0	Y	0
1	0	0	Y	0	0
1	1	Y	0	0	0



2. Write the Boolean function for the output logic.
3. Draw the logic diagram to implement the Boolean function.
4. Select ICs from the equipment list.
5. Observe and note the output logic for all combinations of inputs.

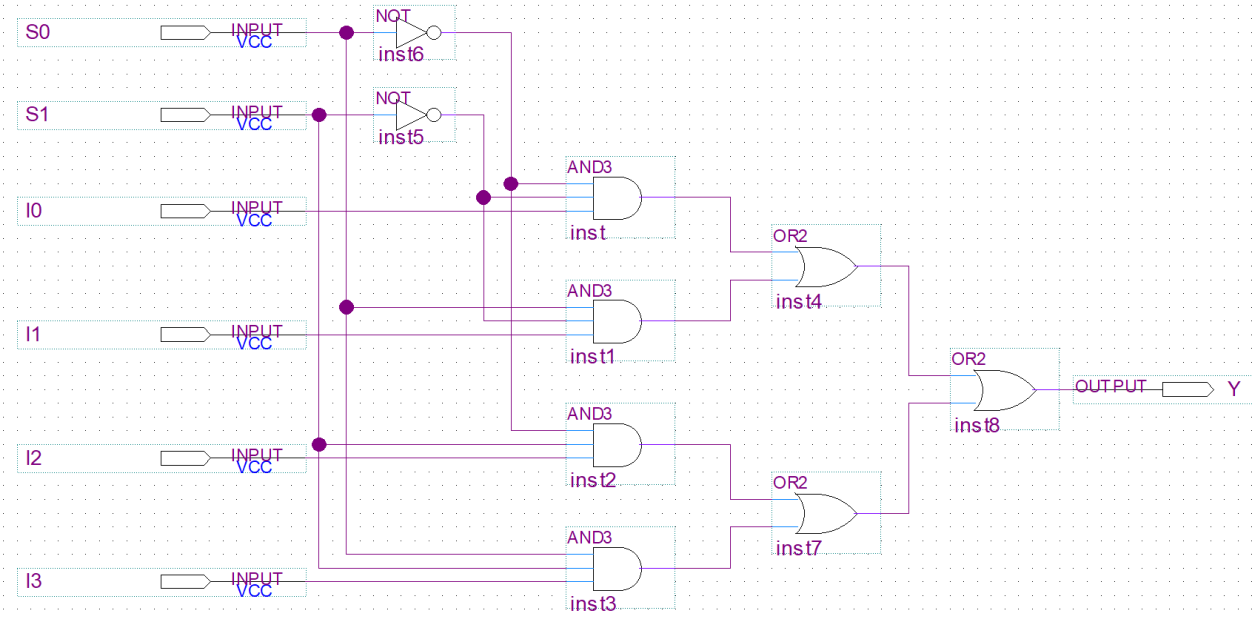
Procedure for FPGA:

Design 4:1 MUX

1. Create a new project and create a new block diagram/schematic file.
2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
I0	SW2	PIN_P25	PIN_AF9
I1	SW3	PIN_AE14	PIN_AF10
I2	SW4	PIN_AF14	PIN_AD11
I3	SW5	PIN_AD13	PIN_AD12
S0	SW0	PIN_N25	PIN_AB12
S1	SW1	PIN_N26	PIN_AC12
Y	LEDR0	PIN_AE23	PIN_V16

3. After assigning pins, the final schematic should look like the following one:



4. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

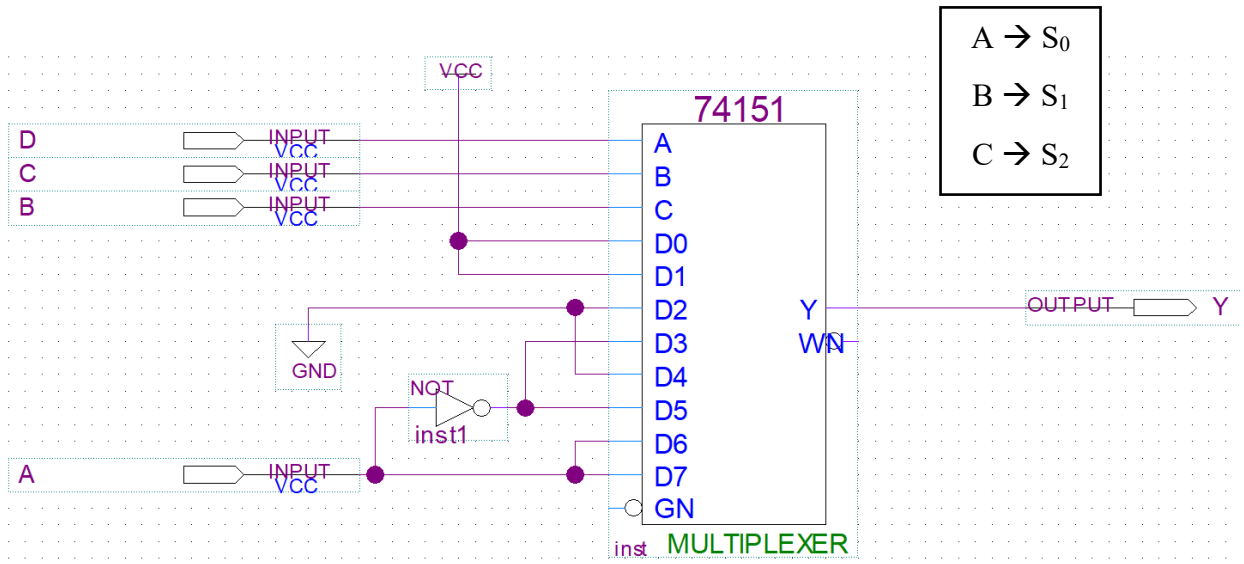
Implement the following function using an 8:1 MUX

$$F(A,B,C,D) = \sum m(0, 1, 3, 5, 8, 9, 14, 15)$$

1. Create a new project and create a new block diagram/schematic file.
2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
A	SW3	PIN_AE14	PIN_AF10
B	SW2	PIN_P25	PIN_AF9
C	SW1	PIN_N26	PIN_AC12
D	SW0	PIN_N25	PIN_AB12
Y	LEDR0	PIN_AE23	PIN_V16

3. After assigning pins, the final schematic should look like the following one:



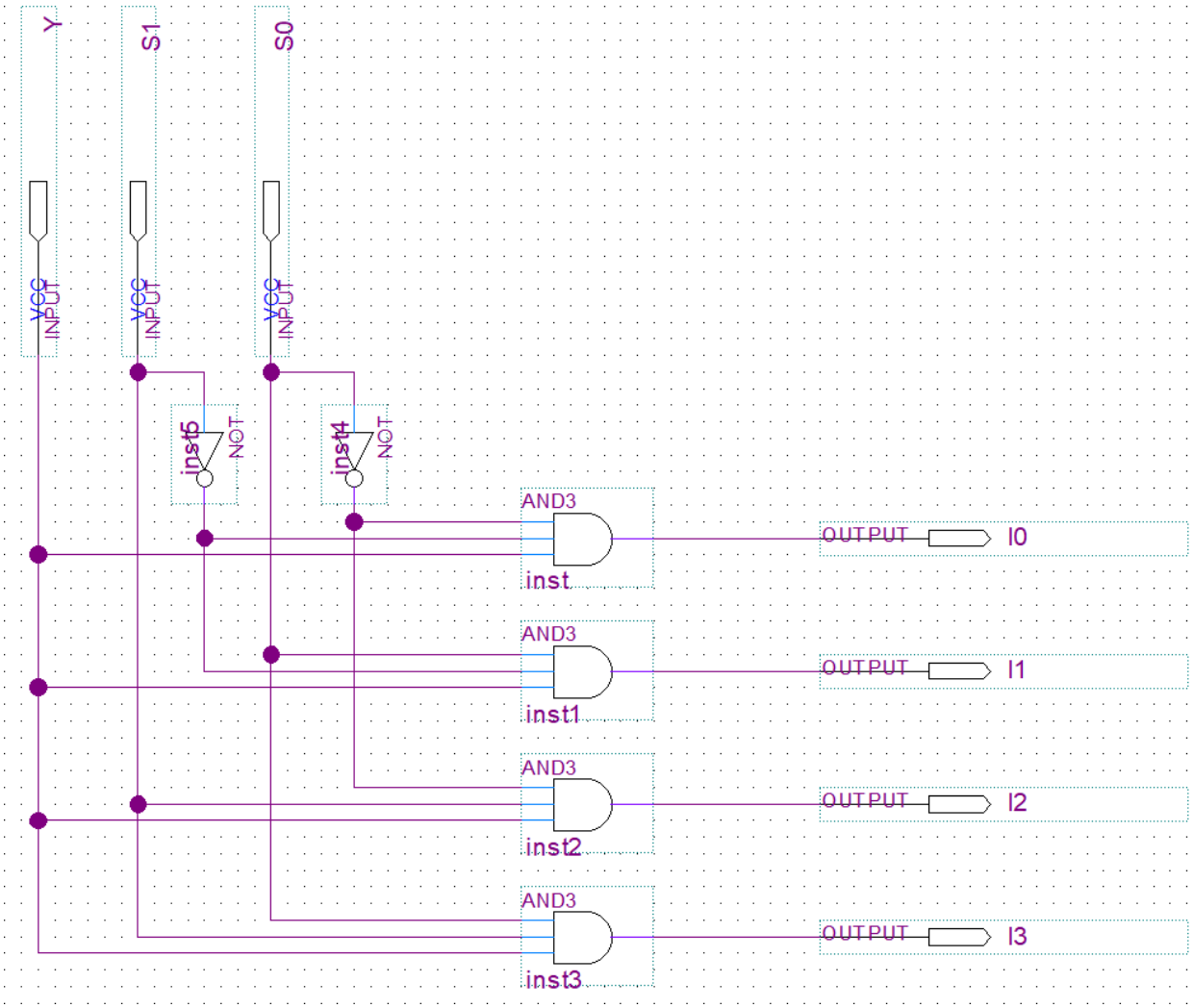
4. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

DEMUX

1. Create a new project and create a new block diagram/schematic file.
2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
S0	SW0	PIN_N25	PIN_AB12
S1	SW1	PIN_N26	PIN_AC12
Y	SW2	PIN_P25	PIN_AF9
I0	LEDR0	PIN_AE23	PIN_V16
I1	LEDR1	PIN_AF23	PIN_W16
I2	LEDR2	PIN_AB21	PIN_V17
I3	LEDR3	PIN_AC22	PIN_V18

3. After assigning pins test the functionality in FPGA board.



MUX (4 to 1)

If-Else If-Else

```
module mux4to1 (I, S, Y);
        input [3:0] I;
        input [1:0] S;
        output reg Y;

        always @ (I, S)
                if (S == 0)
                        Y = I [0];
                else if (S == 1)
                        Y = I [1];
                else if (S == 2)
                        Y = I [2];
                else
                        Y = I [3];
endmodule
```

Case

```
module mux4to1 (I, S, Y);
        input [3:0] I;
        input [1:0] S;
        output reg Y;

        always @ (I, S)
                case (S)
                        0: Y = I [0]; // 2'b00: Y = I[0]
                        1: Y = I [1]; // 2'b01: Y = I[1]
                        2: Y = I [2]; // 2'b10: Y = I[2]
                        3: Y = I [3]; // 2'b11: Y = I[3]
                endcase
 endmodule
```

DEMUX (1 to 4)

```
module demux1to4 (Y, S, I);
        input Y;
        input [1:0] S;
        output reg [3:0] I;

        always @ (Y, S)
        begin
                I = 0; // I = 4'b000
                case(S)
                        0: I [0] = Y; // 2'b00: I[0] = Y
                        1: I [1] = Y; // 2'b01: I[1] = Y
                        2: I [2] = Y; // 2'b10: I[2] = Y
                        3: I [3] = Y; // 2'b11: I[3] = Y
                endcase
        end

endmodule
```

Experiment: 7

Experiment name: *Introduction to Encoders and Decoders.*

Priority Encoder

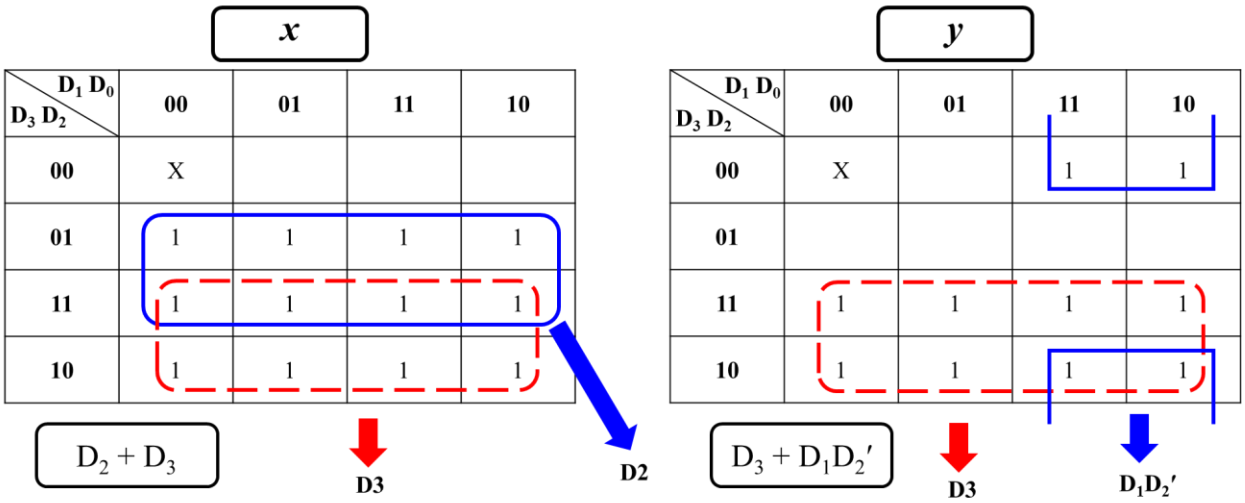
A priority encoder is an encoder circuit that includes priority function. **Priority Encoder** includes the necessary logic to ensure that when two or more inputs are activated, the output code will correspond to the highest-numbered input. The truth table for a 4 to 2 priority encoder is given in Table. The X's are don't-care conditions that designate the fact that the binary value may be equal either to 0 or 1. Input D_3 has the highest priority; so regardless of the values of the other inputs, when this input is 1, the output for **xy** is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$ provided that $D_3 = 0$, regardless of the values of the other two lower-priority inputs. The output for D_1 is generated only if higher-priority inputs are 0, and so on down the priority level. A valid-output indicator, designated by V, is set to 1 only when one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs of the circuit are not used.

Truth table of a Priority Encoder							
Inputs					Outputs		
D_3	**D_2**	**D_1**	**D_0**		***x***	***y***	**V**
0	0	0	0		X	X	0
0	0	0	1		0	0	1
0	0	1	X		0	1	1
0	1	X	X		1	0	1
1	X	X	X		1	1	1

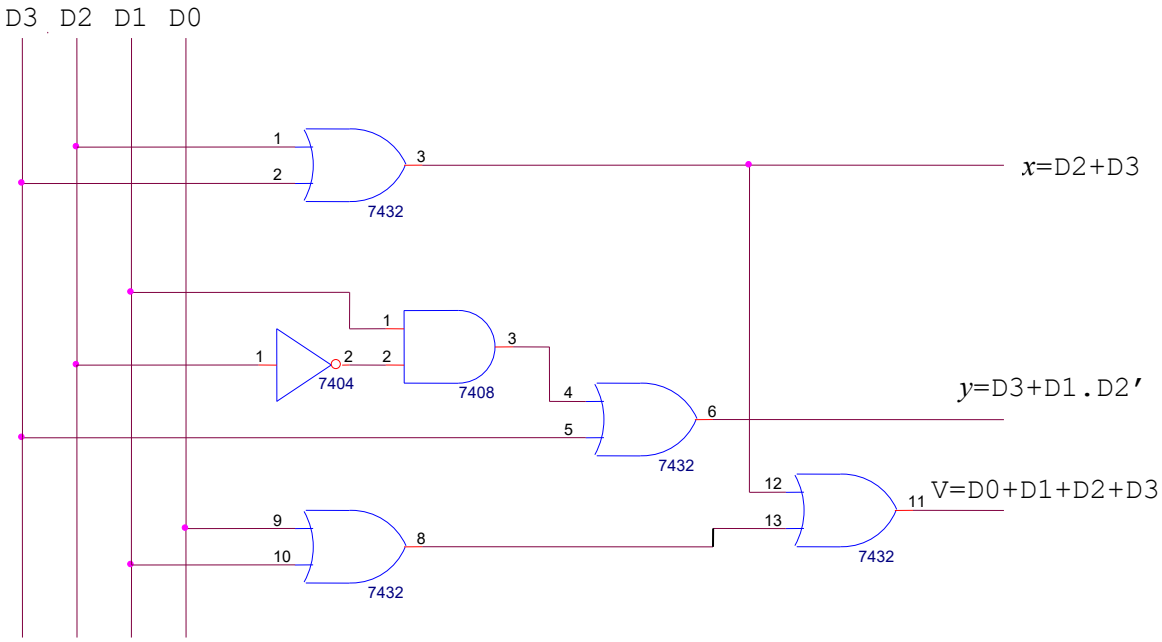
Truth Table

D_3	**D_2**	**D_1**	**D_0**	***x***	***y***	**V**
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

K Map

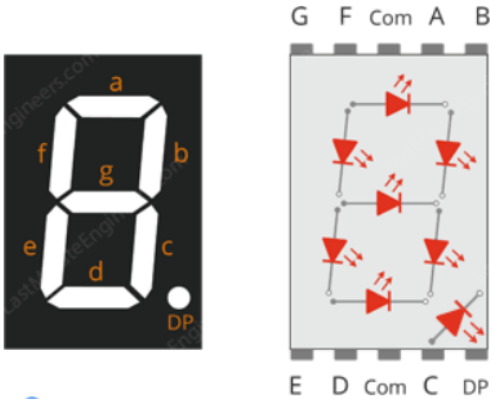


Circuit Diagram



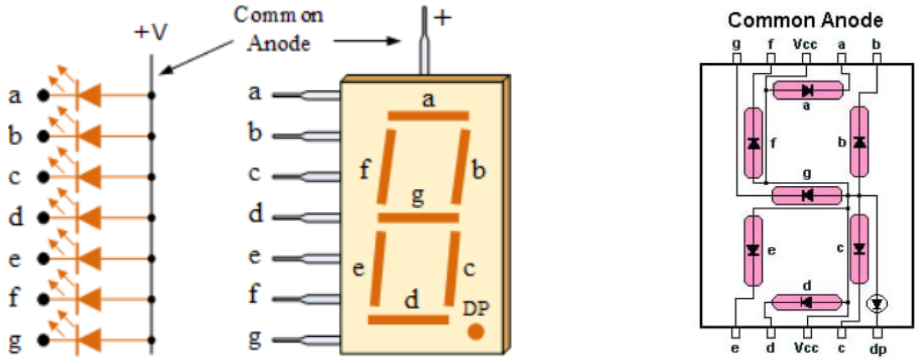
Seven Segment Decoder

The 7-segment display, also written as “seven segment display”, consists of seven LEDs arranged in a rectangular fashion. Each of the seven LEDs is called a segment because when illuminated the segment forms part of a numerical digit (both Decimal and Hex) to be displayed. Each segment has two electrical connection points, just like any standard LED. However, only one of these connection points extends outside the plastic casing as an individual pin that you can connect to. These pins are labeled with letters from ‘a’ through ‘g’. The other connection points from all seven LEDs don’t get their own individual pins. Instead, these connection points are all wired together inside the plastic case to form a single shared connection called the common pin (COM).

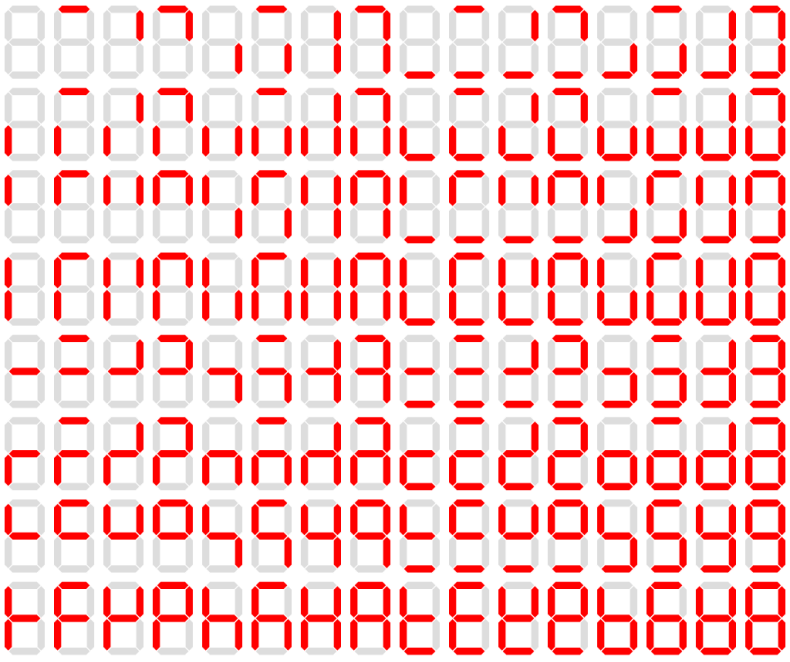


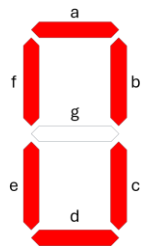
Common Anode 7-segment Display

In the common anode display, all the anode connections of the LED segments are joined together to logic "1". The individual segments are illuminated by applying a ground, logic "0" or "LOW" signal via a suitable current limiting resistor to the Cathode of the particular segment (a-g).

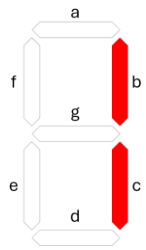


7-Segment Display Segments for all possible combinations --

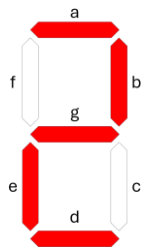




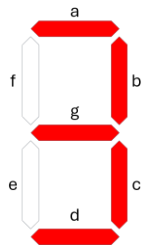
a	b	c	d	e	f	g	Base
0	0	0	0	0	0	1	Binary
0			1				Hex



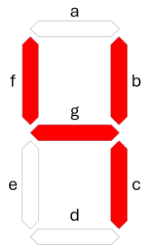
a	b	c	d	e	f	g	Base
1	0	0	1	1	1	1	Binary
4			F				Hex



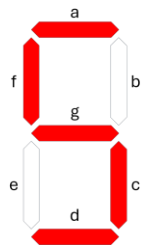
a	b	c	d	e	f	g	Base
							Binary
							Hex



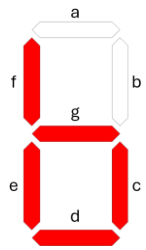
a	b	c	d	e	f	g	Base
							Binary
							Hex



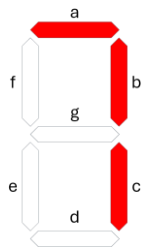
a	b	c	d	e	f	g	Base
							Binary
							Hex



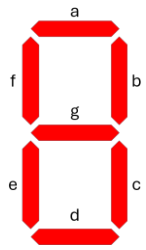
a	b	c	d	e	f	g	Base
							Binary
							Hex



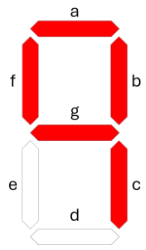
a	b	c	d	e	f	g	Base
							Binary
							Hex



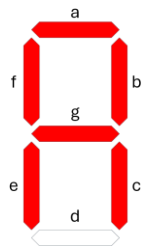
a	b	c	d	e	f	g	Base
							Binary
							Hex



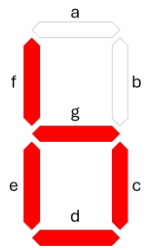
a	b	c	d	e	f	g	Base
							Binary
							Hex



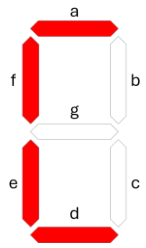
a	b	c	d	e	f	g	Base
							Binary
							Hex



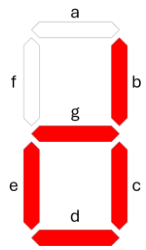
a	b	c	d	e	f	g	Base
							Binary
							Hex



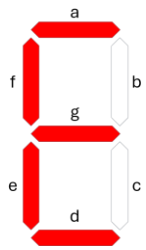
a	b	c	d	e	f	g	Base
							Binary
							Hex



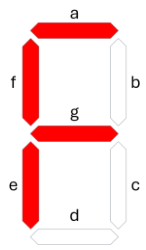
a	b	c	d	e	f	g	Base
							Binary
							Hex



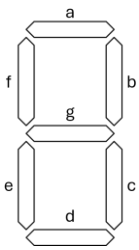
a	b	c	d	e	f	g	Base
							Binary
							Hex



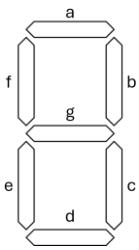
a	b	c	d	e	f	g	Base
							Binary
							Hex



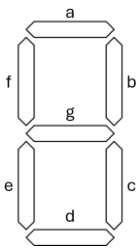
a	b	c	d	e	f	g	Base
							Binary
							Hex



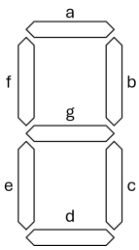
a	b	c	d	e	f	g	Base
							Binary
							Hex



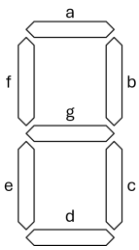
a	b	c	d	e	f	g	Base
							Binary
							Hex



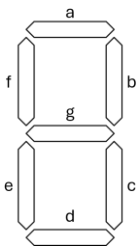
a	b	c	d	e	f	g	Base
							Binary
							Hex



a	b	c	d	e	f	g	Base
							Binary
							Hex



a	b	c	d	e	f	g	Base
							Binary
							Hex



a	b	c	d	e	f	g	Base
							Binary
							Hex

Procedure for FPGA

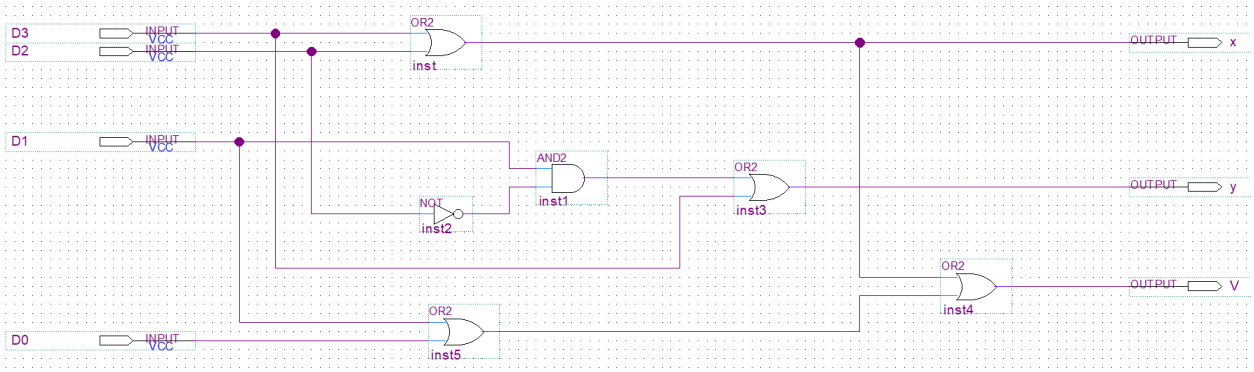
Priority Encoder

1. Create a new project and create a new block diagram/schematic file.

2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
D0	SW0	PIN_N25	PIN_AB12
D1	SW1	PIN_N26	PIN_AC12
D2	SW2	PIN_P25	PIN_AF9
D3	SW3	PIN_AE14	PIN_AF10
V	LEDR0	PIN_AE23	PIN_V16
x	LEDR2	PIN_AB21	PIN_V17
y	LEDR1	PIN_AF23	PIN_W16

3. After assigning pins, the final schematic should look like the following one:



4. Test the functionality of the designed circuit using switches and LEDs on the FPGA board

Seven Segment Display

The FPGA Board has several 7-segment displays. These displays are arranged into pairs or groups, with the intent of displaying numbers of various sizes the seven segments are connected to pins. Applying a low logic level to a segment causes it to light up, and applying a high logic level turns it off. Each segment in a display is identified by an index from 0 to 6 (correspond to **a** to **g** segments respectively), Note that the dot in each display is unconnected and cannot be used.

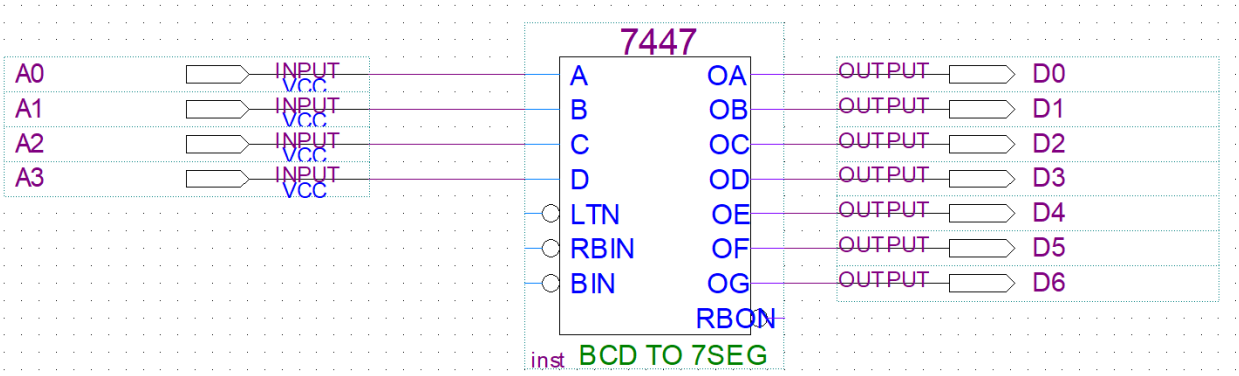


BCD Decoder

1. Create a new project and create a new block diagram/schematic file.

2. Search for **7447** in the name box to find BCD to 7-Segment Common Anode (CA) Decoder.

3. After completing the schematic, it should look like the following:



4. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
A0	SW0	PIN_N25	PIN_AB12
A1	SW1	PIN_N26	PIN_AC12
A2	SW2	PIN_P25	PIN_AF9
A3	SW3	PIN_AE14	PIN_AF10
D0	HEX0[0]	PIN_AF10	PIN_AE26
D1	HEX0[1]	PIN_AB12	PIN_AE27
D2	HEX0[2]	PIN_AC12	PIN_AE28
D3	HEX0[3]	PIN_AD11	PIN_AG27
D4	HEX0[4]	PIN_AE11	PIN_AF28
D5	HEX0[5]	PIN_V14	PIN_AG28
D6	HEX0[6]	PIN_V13	PIN_AH28

5. After assigning pins, the final schematic should look like the following one:

6. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

Verilog Code

Priority Encoder

```
module priority (D, x, y, V);

        input [3:0] D; // Input D3 > D2 > D1 > D0

        output reg x, y, V; // Output: x --> MSB, y --> LSB, V --> Validity


        always @ (D)

        begin

                V = D[3] | D[2] | D[1] | D[0]; // OR operation

                casex (D)

                        4'b1xxx: {x, y} = 2'b11; // {x, y} = 3

                        4'b01xx: {x, y} = 2'b10; // {x, y} = 2

                        4'b001x: {x, y} = 2'b01; // {x, y} = 1

                        4'b0001: {x, y} = 2'b00; // {x, y} = 0

                        default: {x, y} = 2'bx;  // {x, y} = 2'bxx

                endcase

        end


endmodule
```

BCD Decoder with 7-segment Display

```
module seg7(A, D);

    input [3:0] A;

    output reg [0:6] D; // abcdefg


    always @ (A)

        case (A)

            0: D = 7'b0000001;  //  7'b 000_0001 (for better readability)

            1: D = 7'b1001111; //  7'h4F (Hex Version)

            2: D = 7'b0010010;

            3: D = 7'b0000110;

            4: D = 7'b1001100; //  4'b0100 (A ← Input)

            5: D = 7'b0100100;

            6: D = 7'b1100000; //  7'b 010_0000 (with Tail)

            7: D = 7'b0001111;

            8: D = 7'b0000000;

            9: D = 7'b0001100; //  7'b 000_0100 (Tail version)

            default: D = 7'b1111111; //  Blank Display or 7'bx (Don't Care)

        endcase

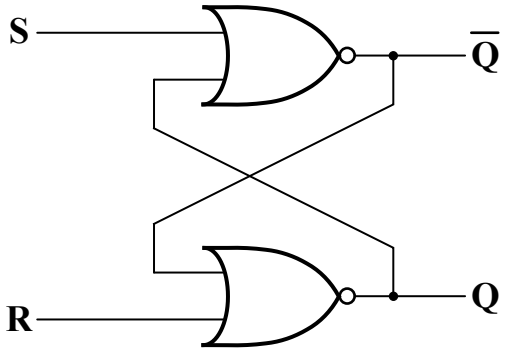

endmodule
```

Experiment: 8

Experiment name: *Introduction to Sequential Logic Circuits.*

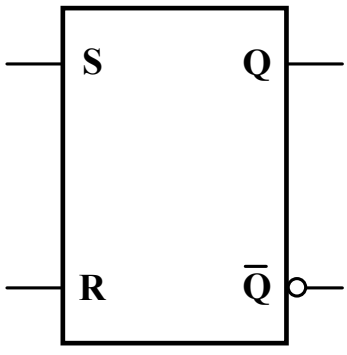
SR Latch

An **SR latch** (Set–Reset latch) is the simplest memory element used in digital logic. The SR latch is a circuit with two cross-coupled NOR gates, and two inputs labeled S for set and R for reset. This latch can store one bit of information. This latch has two outputs, Q and Q′ – which are normally the complement of each other.



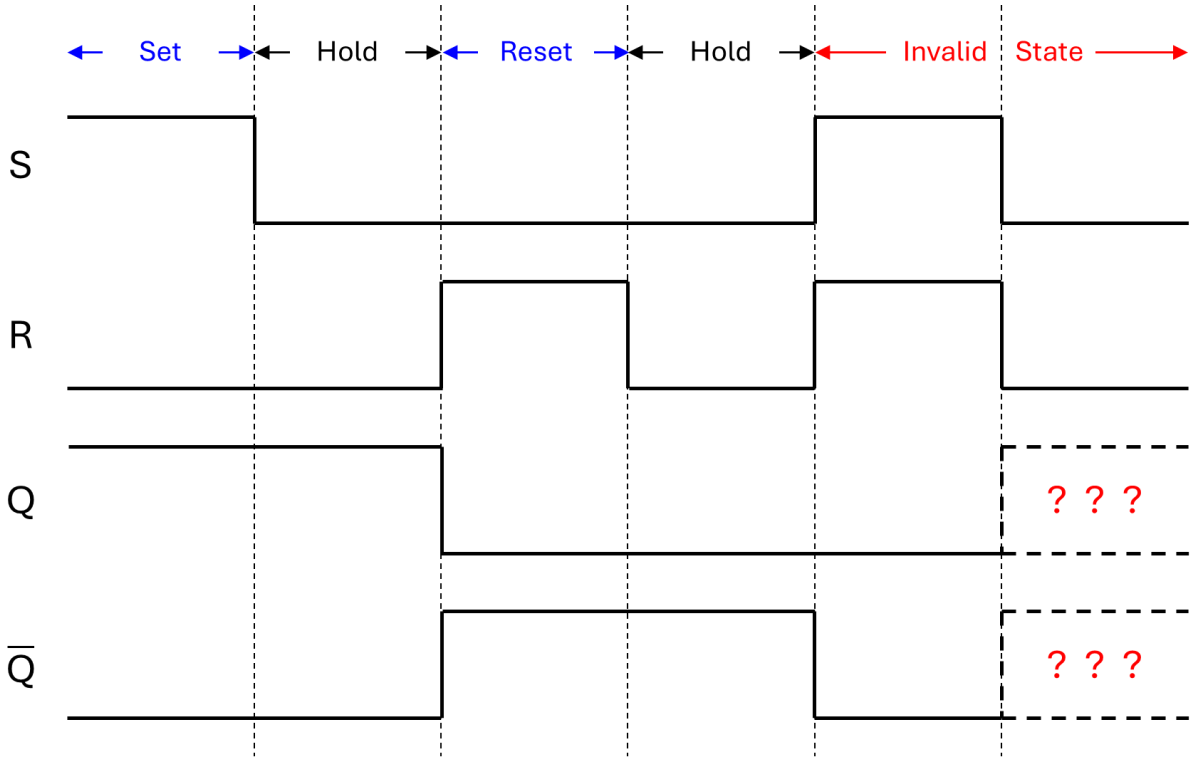
- S = R = 0, this is the normal resting state for the NOR latch, and it has no effect on the output state. Q and Q′ will remain in whatever state they were in prior to the occurrence of this input condition.
- S = 1, R = 0, this will always set Q = 1, where it will remain even after SET returns to 0.
- R = 1, S = 0, this will always clear Q = 0, where it will remain even after RESET returns to 0.
- S = R = 1, this condition tries to set and reset the latch at the same time, and it produces Q = Q′ = 0. If the inputs are returned to 0 simultaneously, the resulting output state is unpredictable. This input condition should not be used.

S	R	Q	Q′	State
0	0	NC	NC	Hold (No Change)
0	1	0	1	Reset
1	0	1	0	Set
1	1	0	0	Invalid



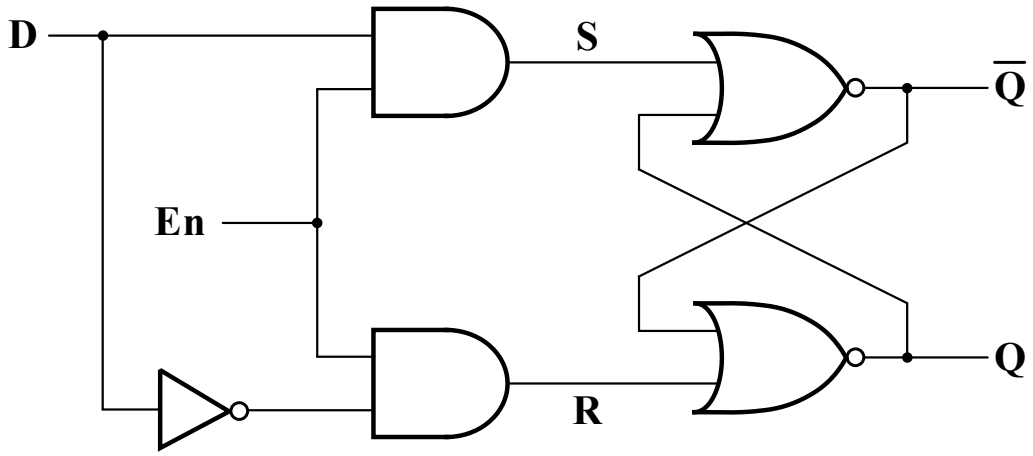
- The application of a momentary 1 to the S input causes the latch to go to the set state. Removing the active input from S leaves the circuit in the same state.
- After both inputs return to 0, it is then possible to shift to the reset state by momentary applying a 1 to the R input. The 1 can then be removed from R, where upon the circuit remains in the reset state.
- If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0.

Input		Output		State
S	**R**	**Q**	**Q′**	
1	0	1	0	Set
0	0	1	0	Hold
0	1	0	1	Reset
0	0	0	1	Hold
1	1	0	0	Invalid
0	0	?	?	

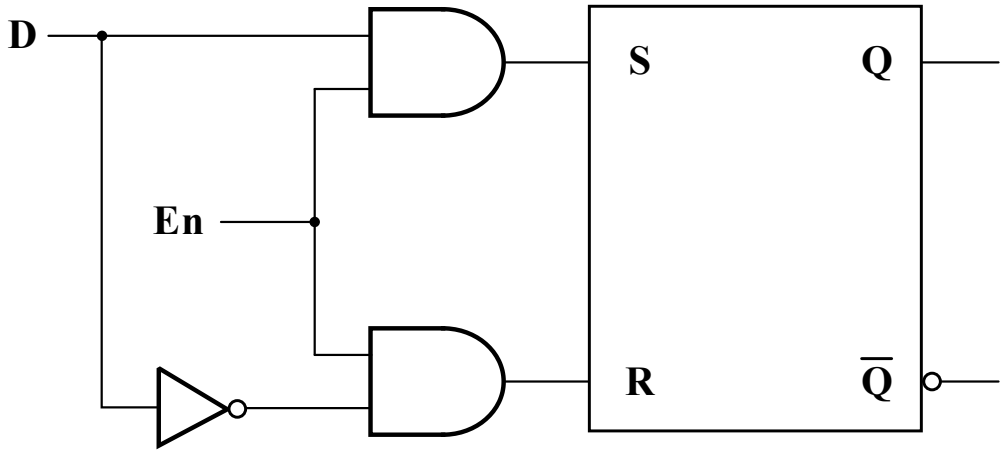


D Latch

Although SR latch is useful it has one major problem with its indeterminate state making the circuit unstable for precise applications. One way to eliminate the undesirable condition of the indeterminate state is to ensure that inputs **S** and **R** are never equal to 1 at the same time. This is done in the D latch. This latch has only two inputs: **D** (data) and **En** (enable). This single data input (D) stores the value on this input, under the control of Enable (En) signal. It is also called a ***Gated D Latch*** as the Enable signal acts as a gate to the latch which controls when the latch will respond to changes in Data input.

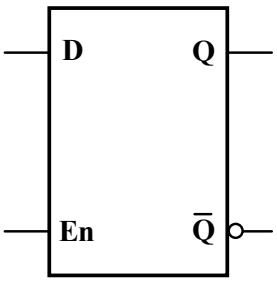


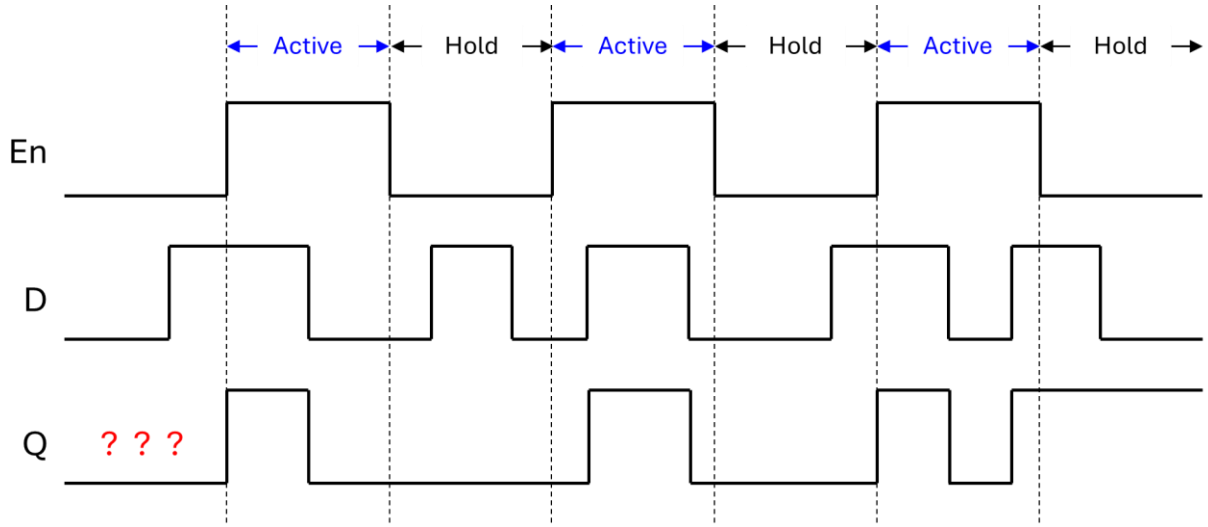
- As long as the Enable input signal is LOW (En = 0), the cross-coupled SR latch has both inputs at the 0 level and the circuit cannot change state regardless of the value of D.
- The D input is sampled when Enable input signal is HIGH (En = 1) –
  - If D = 1, then S = 1 and R = 0, the Q output goes to 1, placing the circuit in the SET state.
  - If D = 0, then S = 0 and R = 1, output Q goes to 0, placing the circuit in the RESET state.



The D latch receives that designation from its ability to hold data in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a transparent latch. When the enable input signal is de-asserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the Q output until the enable input is asserted again. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be ***level sensitive***.

En	D	Q(t+1)	State
0	x	Q(t)	Hold
1	0	0	Reset
1	1	1	Set





Level vs Edge Sensitive

Latch are known as level-sensitive circuits as they respond to inputs whenever the enable or clock signal is at a particular logic level, meaning the output can change continuously during that time. In contrast, edge-sensitive circuits update their output only at the moment of a clock transition, either the rising or falling edge – making them more stable and immune to glitches. This edge-sensitive circuit is known as Flip-Flop. Thus, level-sensitive devices like latches are transparent during the active level, while edge-sensitive devices like flip-flops capture data only at the edge.



(a) Response to positive level



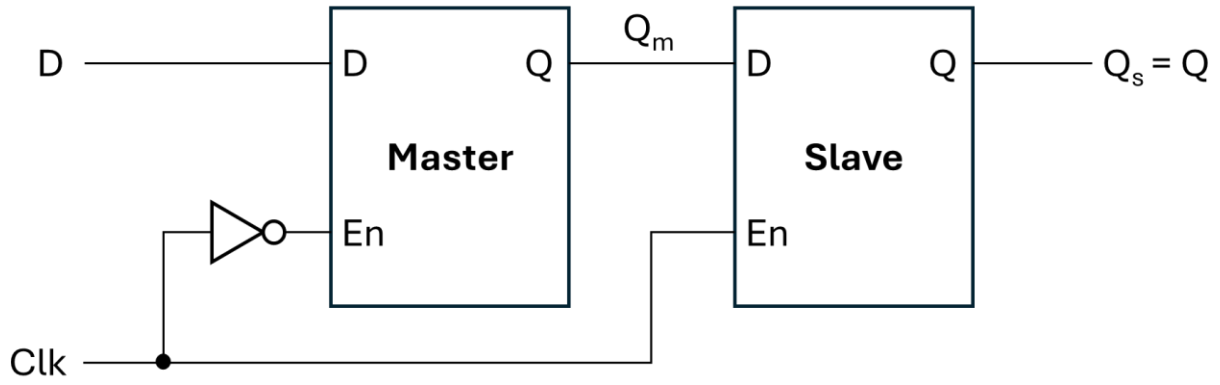
(b) Positive-edge response

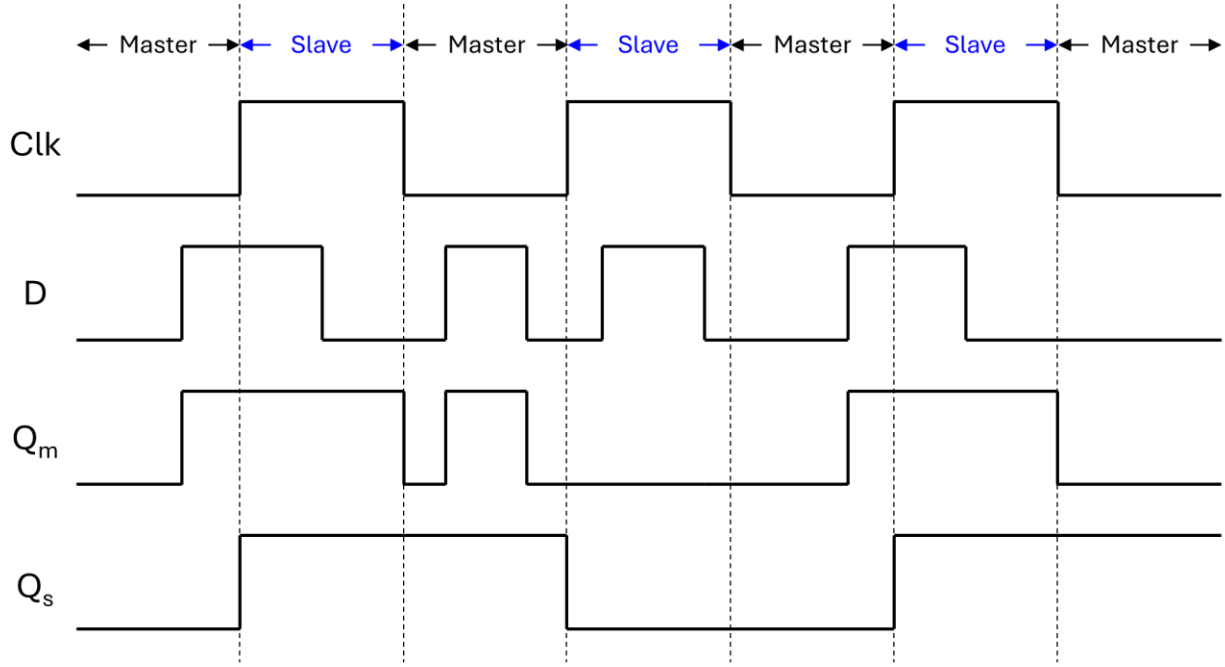


(c) Negative-edge response

D Flip-Flop

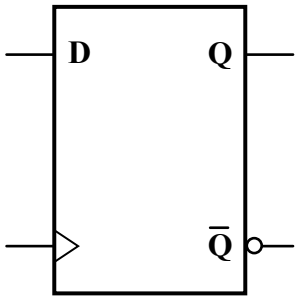
Now we will construct a positive edge sensitive Flip-Flop known as D Flip-Flop. Consider the circuit which consists of two gated D latches. The first, called ***master***, changes its state while Clock signal is LOW (Clk = 0). The second, called ***slave***, changes its state while Clock signal is HIGH (Clk = 1). The operation of the circuit is such that when the clock is low, the master tracks the value of the D input signal and the slave does not change. Thus, the value of Q_m follows any changes in D, and the value of Q_s remains constant. When the clock signal changes to 1, the master stage stops following the changes in the D input. At the same time, the slave stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while Clock = 1, the slave stage can undergo at most one change of state during a clock cycle.





From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the positive-going edge of the clock. The ***positive edge*** is the edge where the clock signal changes from 0 to 1. Regardless of the number of changes in the D input to the master stage during one clock cycle, the observer of the Q_s signal will see only the change that corresponds to the D input at the positive edge of the clock. The circuit is called a ***master-slave D flip-flop***. This storage element only changes its output state at the edge of a controlling clock signal. In the symbol we use the > mark to denote that the flip-flop responds to the ***active edge*** of the clock.

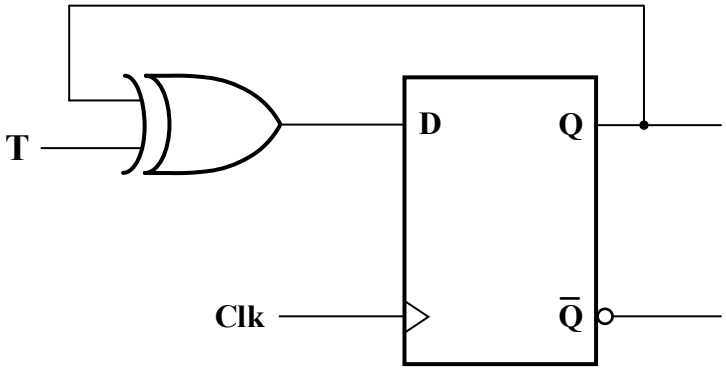
D	Q(t+1)	State
0	0	Reset
1	1	Set



T Flip-Flop

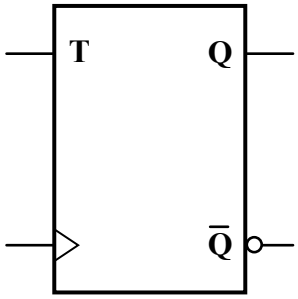
The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. A special can be constructed with a D flip-flop and an exclusive-OR gate. The expression for the D input is –

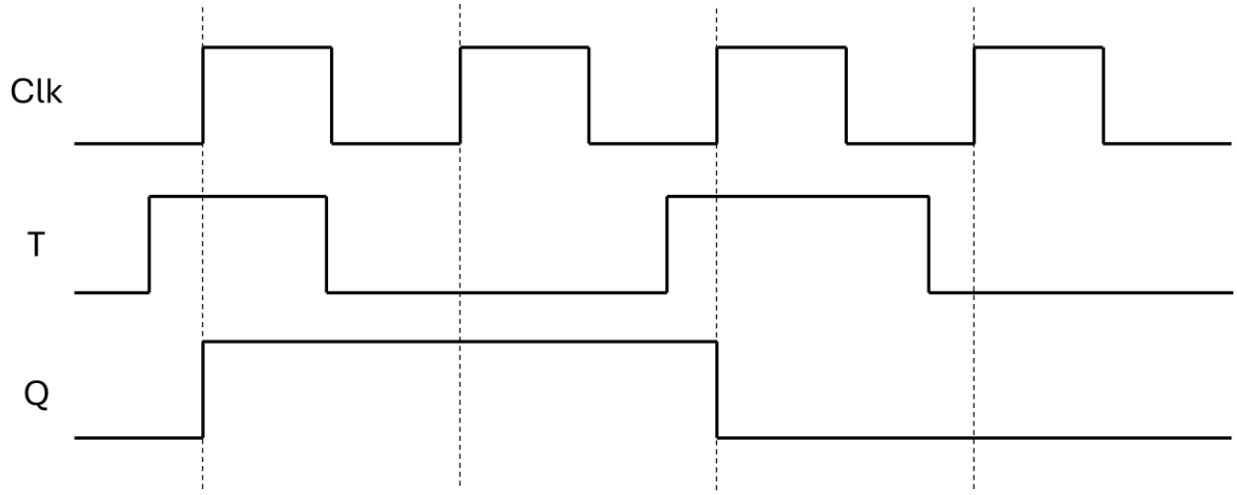
$$D = T \oplus Q = TQ' + T'Q$$



Here, the input signal D equal to either the value of Q or Q′ depending on the signal that is labeled T. On each positive edge of the clock, the flip-flop may change its state. If T = 0, then D = Q and the state will remain the same. But if T = 1, then D = Q′. Therefore, the overall operation of the circuit is that it retains its present state if T = 0, and it reverses its present state if T = 1. This circuit configuration is called a T flip-flop. The name T flip-flop derives from the behavior of the circuit, which ***toggles*** its state when T = 1. The toggle feature makes the T flip-flop a useful element for building counter circuits.

T	Q(t+1)	State
0	Q(t)	Hold
1	Q′(t)	Toggle

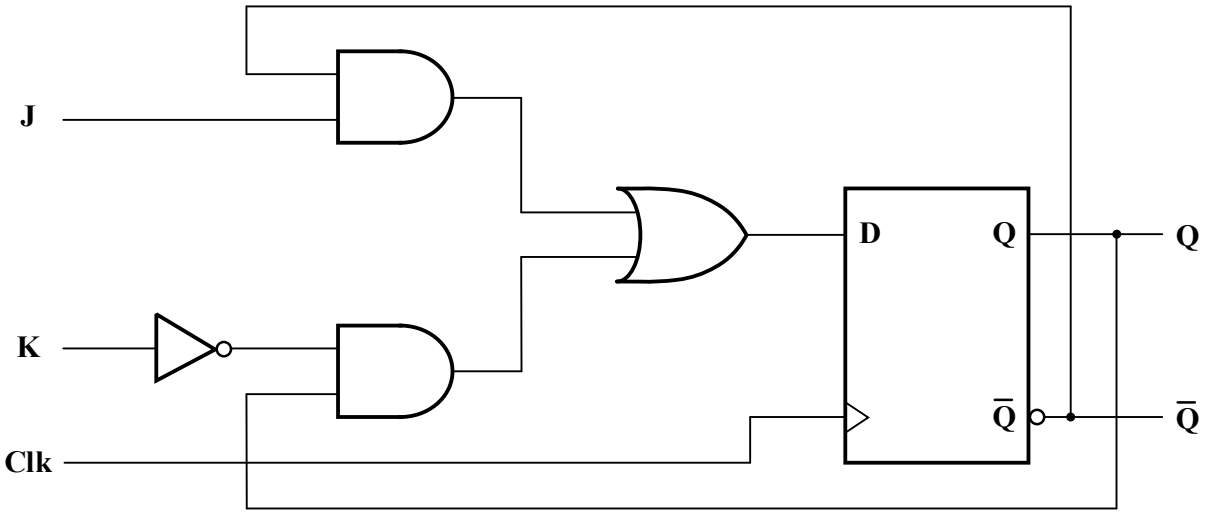




JK Flip-Flop

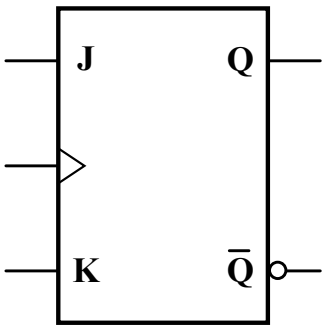
Another interesting circuit can be derived when instead of using a single control input, T, we can use two inputs, J and K. For this circuit the input D is defined as –

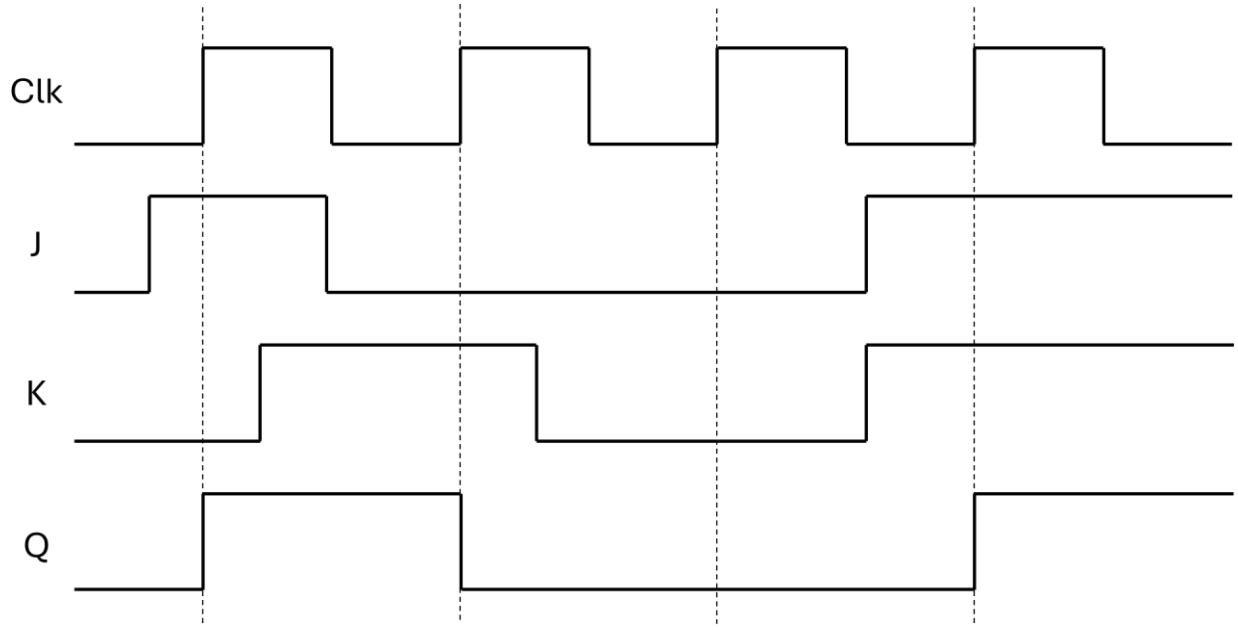
$$D = JQ' + K'Q$$



The circuit is called a JK f lip-flop. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop, where J = S and K = R, for all input values except J = K = 1. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop. The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the J and K inputs together.

J	K	Q(t+1)	State
0	0	Q(t)	Hold
0	1	0	Reset
1	0	1	Set
1	1	Q′(t)	Toggle





- When J = 1 and K = 0, D = Q′ + Q = 1, so the next clock edge sets the output to 1.
- When J = 0 and K = 1, D = 0, so the next clock edge resets the output to 0.
- When both J = K = 1, D = Q′, so the next clock edge complements the output.
- When both J = K = 0, D = Q, so the next clock edge leaves the output unchanged.

Procedure for FPGA:

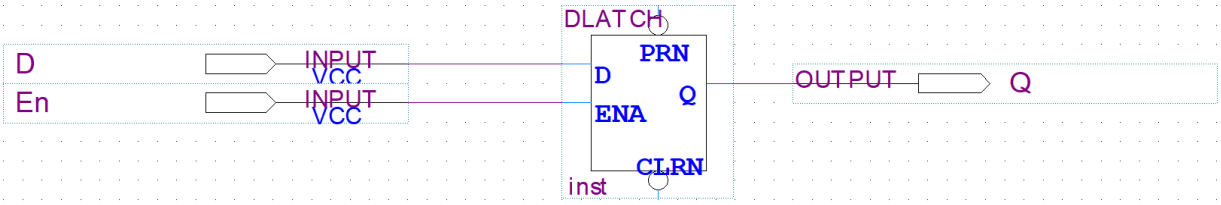
> Do NOT give project name -- dlatch, dff, tff, jkff

D Latch

1. Create a new project and create a new block diagram/schematic file.

2. Complete the circuit using block for D Latch.

   Block name: **dlatch** (Library: Primitives → Storage)



3. Compile and simulate the schematic.

4. Assign pins using the following table.

Signal	Name	Cyclone II	Cyclone V
En	SW1	PIN_N26	PIN_AC12
D	SW0	PIN_N25	PIN_AB12
Q	LEDR0	PIN_AE23	PIN_V16

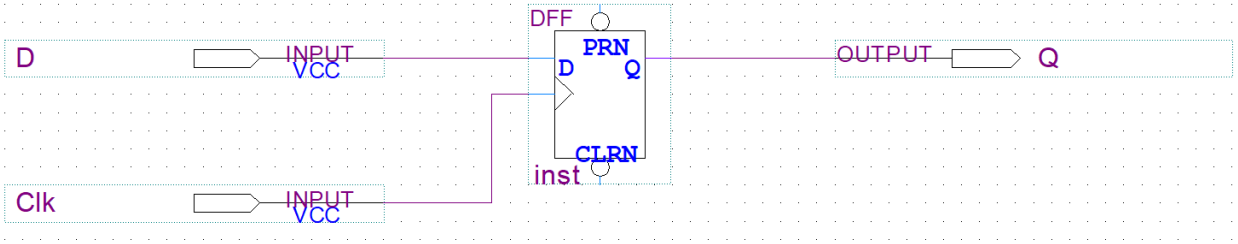
5. Verify the functionality of your schematic.

D Flip-Flop

1. Create a new project and create a new block diagram/schematic file.

2. Complete the circuit using block for D Flip-Flop.

   Block name: **dff** (Library: Primitives → Storage)



3. Compile and simulate the schematic.

4. Assign pins using the following table.

Signal	Name	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
D	SW0	PIN_N25	PIN_AB12
Q	LEDR0	PIN_AE23	PIN_V16

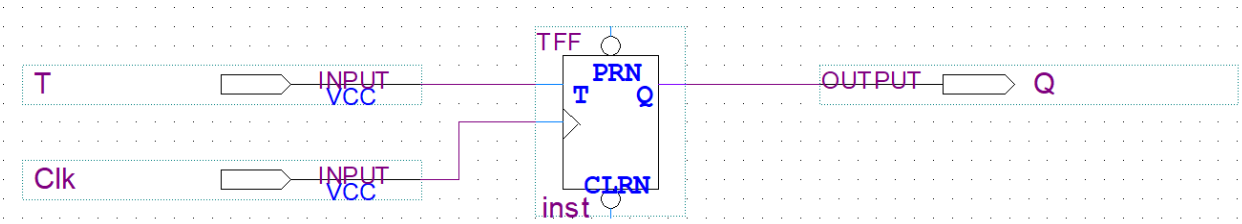
5. Verify the functionality of your schematic.

T Flip-Flop

1. Create a new project and create a new block diagram/schematic file.

2. Complete the circuit using block for T Flip-Flop.

   Block name: **tff** (Library: Primitives → Storage)



3. Compile and simulate the schematic.

4. Assign pins using the following table.

Signal	Name	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
T	SW0	PIN_N25	PIN_AB12
Q	LEDR0	PIN_AE23	PIN_V16

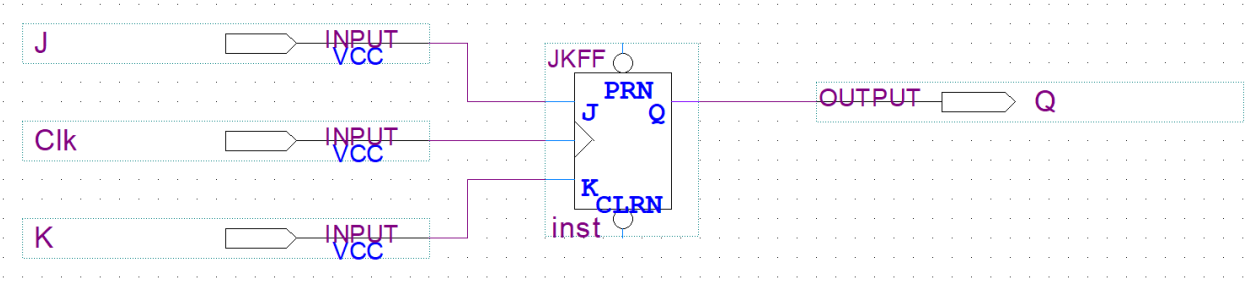
5. Verify the functionality of your schematic.

JK Flip-Flop

1. Create a new project and create a new block diagram/schematic file.

2. Complete the circuit using block for JK Flip-Flop.

   Block name: **jkff** (Library: Primitives → Storage)



3. Compile and simulate the schematic.

4. Assign pins using the following table.

Signal	Name	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
J	SW1	PIN_N26	PIN_AC12
K	SW0	PIN_N25	PIN_AB12
Q	LEDR0	PIN_AE23	PIN_V16

5. Verify the functionality of your schematic.

Verilog Code

D Latch

```
module d_latch(En, D, Q);
        input En, D;
        output reg Q;

        always @ (En, D)
                if (En == 1) // if (En)
                        Q <= D; // Data (Set or Reset)
                else // (En == 0) → "else" block is optional
                        Q <= Q; // Hold

endmodule
```

D Flip Flop

```
module d_flipflop(Clk, D, Q);
        input Clk, D;
        output reg Q;

        always @ (posedge Clk)
                Q <= D;

endmodule
```

T Flip Flop

```
module t_flipflop (Clk, T, Q);
        input Clk, T;
        output reg Q;

        always @ (posedge Clk)
                if (T)
                        Q <= ~ Q; // Toggle
endmodule
```

JK Flip Flop

```
module jk_flipflop (Clk, J, K, Q);
        input Clk, J, K;
        output reg Q;

        always @ (posedge Clk)
                case ( {J , K} )
                        2'b00: Q <= Q; // Hold
                        2'b01: Q <= 0; // Reset
                        2'b10: Q <= 1; // Set
                        2'b11: Q <= ~Q; // Toggle
                endcase
endmodule
```

Experiment: 9

Experiment name: *Introduction to Shift-Register & Counter.*

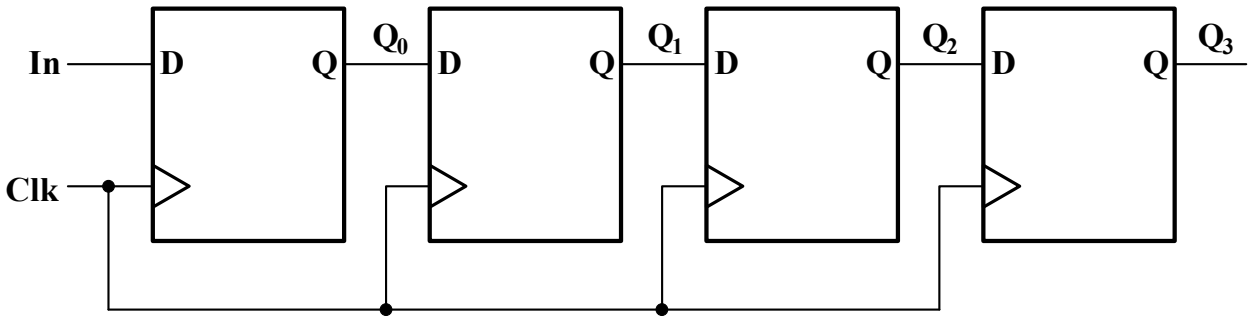
Shift Register

A register capable of shifting its binary information either to the right or to the left is called shift register. The logical configuration of a shift register consists of a chain of flip flops connected in cascade with the output of one flipflop connected to the input of the next flipflop. All flipflops receive a common clock pulse which causes the shift from one stage to the next.

A register is a digital circuit with two basic functions: data storage and data movement. The storage capability of a register makes it an important type of memory device. The storage capacity of the register is the total number of bits (1s and 0s) of digital data it contains. Each stage(flipflop) in a shift register represents one bit of storage capacity, therefore the number of stages in a register determines its storage capacity.

Serial in Serial out Shift Register

A serial-in serial-out (SISO) shift register is a digital logic circuit where data bits are entered one at a time through a single input line and are retrieved one at a time from a single output line, typically using a series of flip-flops connected in a chain. Each clock pulse shifts the existing data through the flip-flops, with a new bit entering the first flip-flop and the oldest bit exiting the last flip-flop. This operation effectively delays the data by one clock cycle for each flip-flop in the register.

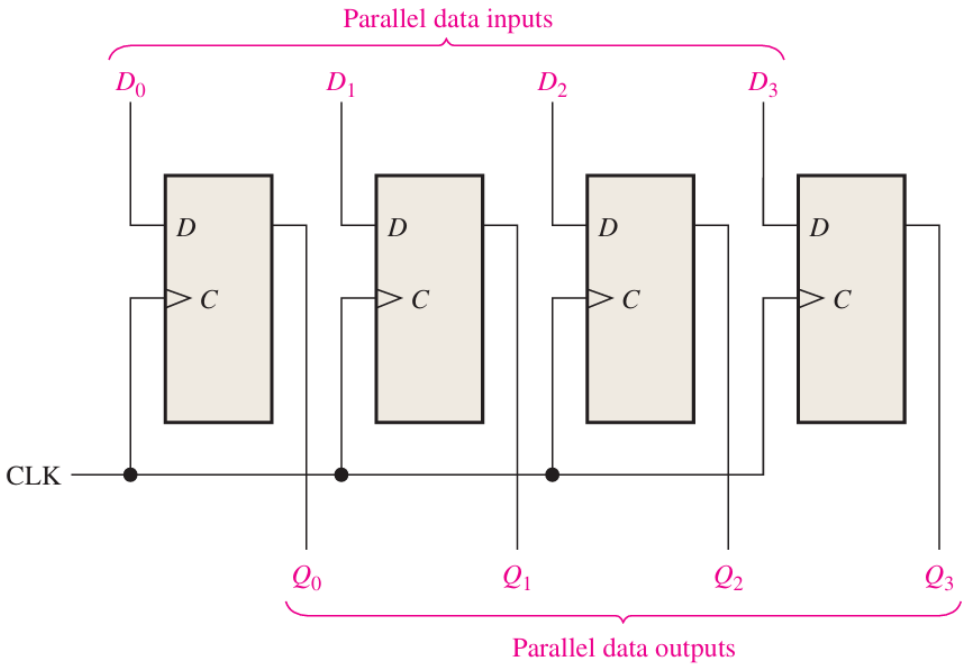


For example, suppose we want to serially load the data **0011** into the register. Here, the leftmost **0** is the Most Significant Bit (MSB) and the rightmost **1** is the Least Significant Bit (LSB). The register is initially cleared, meaning all outputs are low before any clock pulses are applied ($Q_0Q_1Q_2Q_3 = 0000$). To load the data, we first set the serial input to the correct bit value and then apply a clock pulse. The loading process begins with the **LSB** (which is **1** in this example). After that, each subsequent bit is applied on successive clock pulses, and the sequence finishes with the **MSB** (which is **0** here).

INPUT		OUTPUT			
Clk	**In**	**Q0**	**Q1**	**Q2**	**Q3**
x	x	0	0	0	0
1st	1	1	0	0	0
2nd	1	1	1	0	0
3rd	0	0	1	1	0
4th	0	0	0	1	1

Parallel in Parallel out Shift Register

For a register with parallel-in parallel-out (PIPO), the bits are entered simultaneously into their respective stages on parallel lines rather than on a bit-by-bit basis on one line as serial data inputs. Also, the data bits are taken out parallel manner. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously rather than on a bit-by-bit basis as with the serial output.

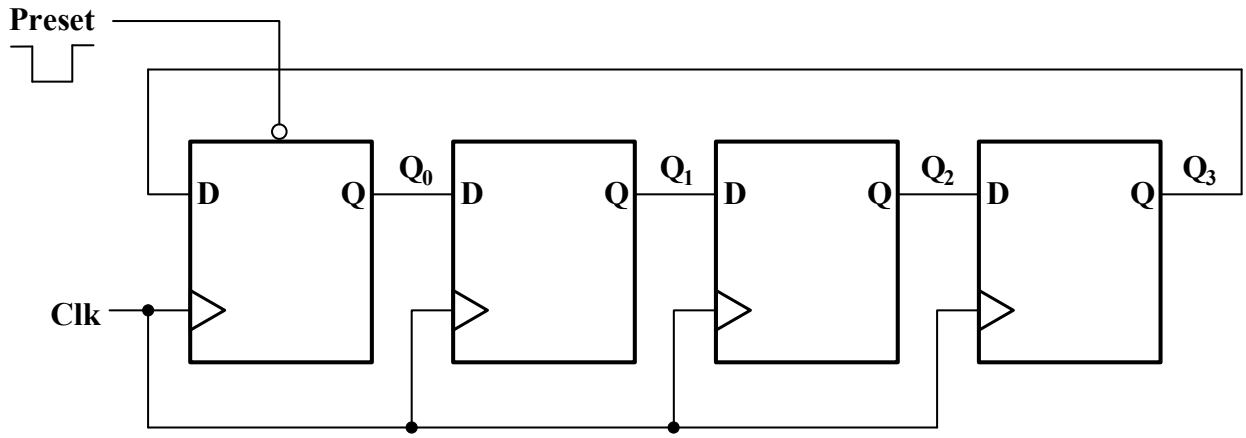


SHIFT REGISTER COUNTER

A shift register counter is basically a shift register with the serial output connected back to the serial input to produce special sequences. These devices are often classified as counters because they exhibit a specific sequence of states. Here we will discuss two of most common types of shift register counters.

RING COUNTER

A ring counter is a type of counter made from a shift register where the output of the last flip-flop is fed back to the input of the first, forming a closed loop. It is essentially a circular shift register that circulates a single logic "1" around the ring with each clock pulse, so only one flip-flop is HIGH at a time. For an n-bit ring counter, there are n distinct states if it contains n flip-flops, and the "1" continues to circulate through these states. The initial state must be set properly, usually by presetting one flip-flop to 1 and the others to 0. Ring counters are mainly used in sequence generation, timing applications, and control circuits.

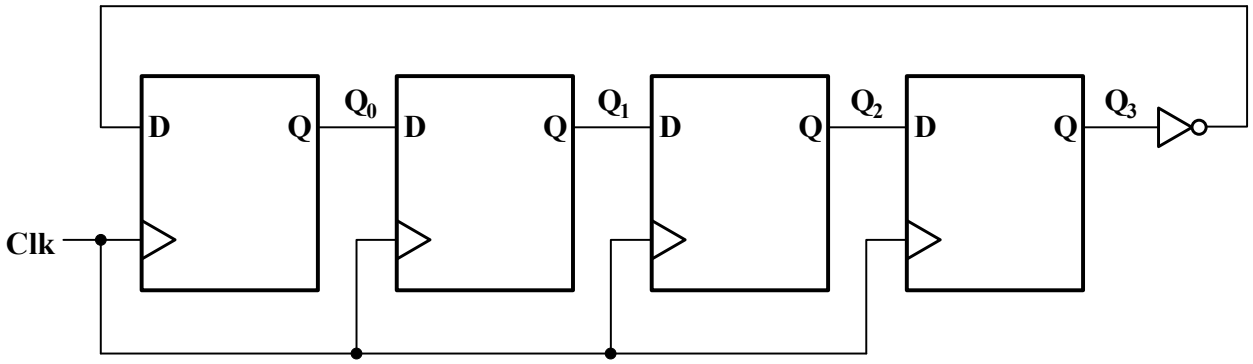


We will construct a 4-bit Ring Counter. Outputs are shown in table below –

Clock Pulse	Q0	Q1	Q2	Q3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

JOHNSON COUNTER

In a Johnson counter the complement of the last flipflop is connected back to the D input of the first flipflop. It is essentially a circular shift register with feedback, and for an n-bit Johnson counter, it produces **2n distinct states.** This feedback arrangement produces a characteristic sequence of states which is shown in the following table for a 4-bit device. 4-bit sequence has a total of eight states or bit patterns.



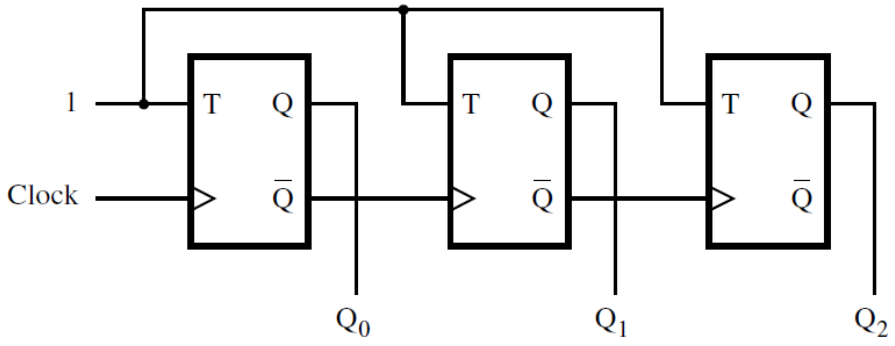
Clock Pulse	Q0	Q1	Q2	Q3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

BINARY COUNTERS

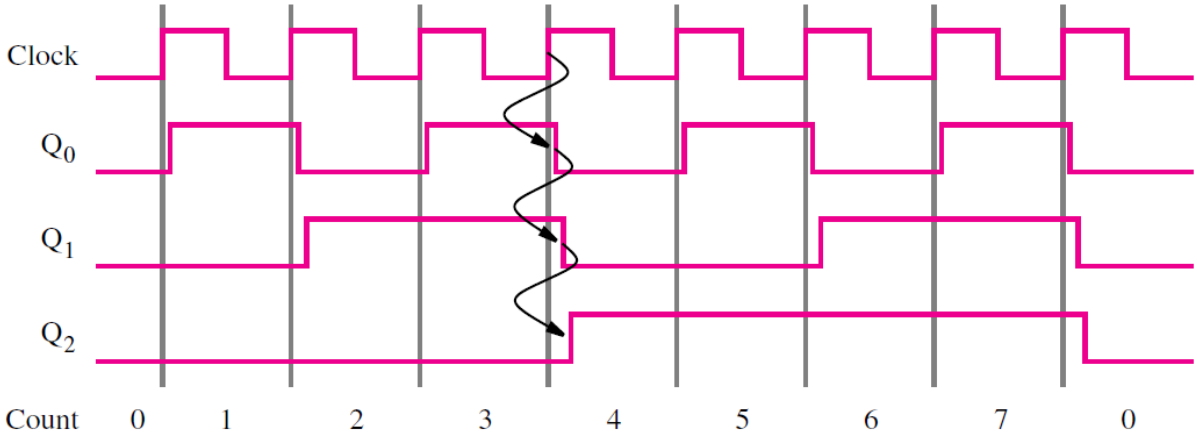
For counting, we want to design circuits that can increment or decrement a count by 1. The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation. These types of counters are called ***Asynchronous*** because not all the flipflops are ***clocked*** by the same ***clock*** signal.

UP COUNTER

A three-bit up counter is capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The T input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. The **clock input** of the first flip-flop is connected to the ***Clock*** line. The other two flip-flops have their clock inputs driven by the **Q′** ***output of the preceding flip-flop***. Therefore, they toggle their state whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a positive edge of the **Q′** signal.



(a) Circuit

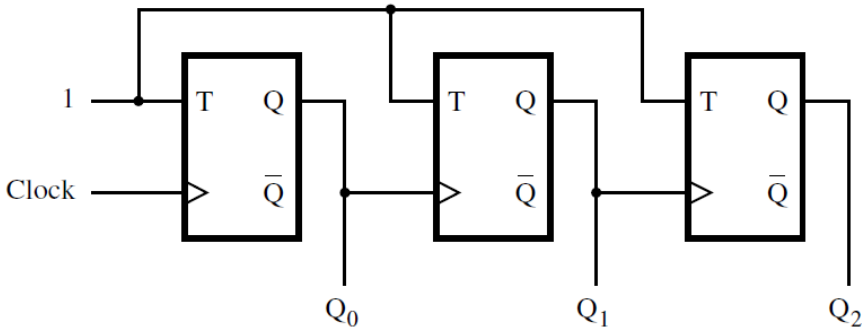


(b) Timing diagram

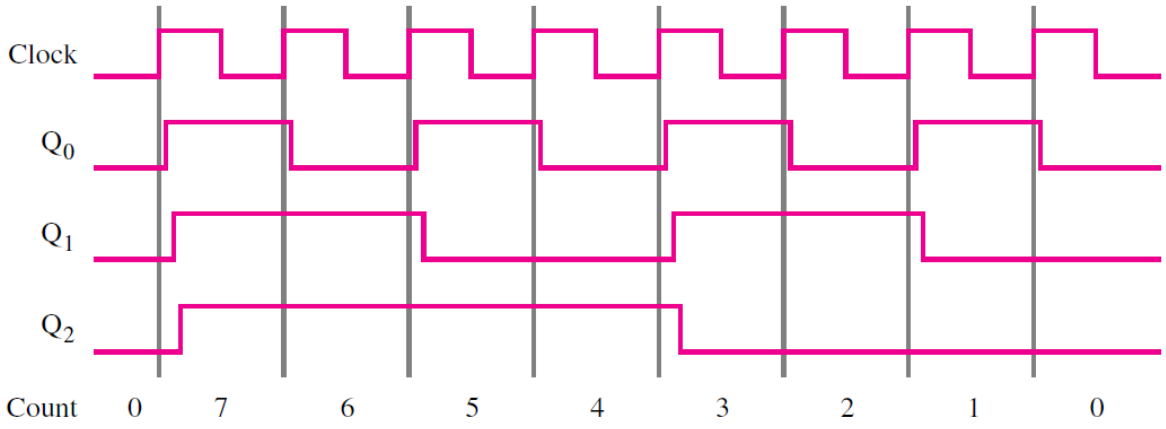
The value of Q_0 toggles once each clock cycle. The change takes place shortly after the positive edge of the ***Clock*** signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by Q_0', the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. As indicated by the arrows in the timing diagram, this change requires the toggling of the states of all three flip-flops. If we look at the values Q_2 Q_1 Q_0 as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an ***up-counter***.

DOWN COUNTER

A slight modification of the circuit in up counter will generate down counter. The only difference is that the clock inputs of the second and third flip-flops are driven by the **Q** outputs of the preceding stages, rather than by the **Q′** outputs. The timing diagram shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a ***down-counter***.



(a) Circuit

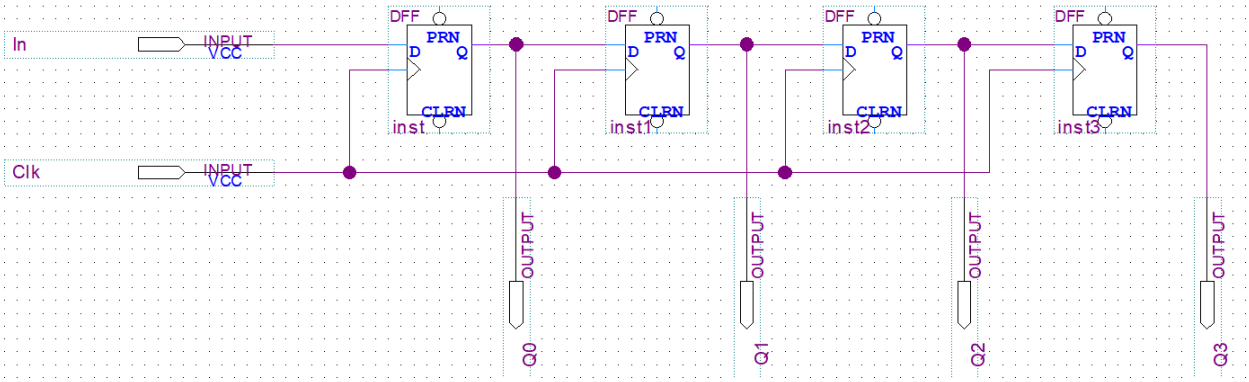


(b) Timing diagram

Procedures in FPGA

Serial in Serial out Shift Register

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

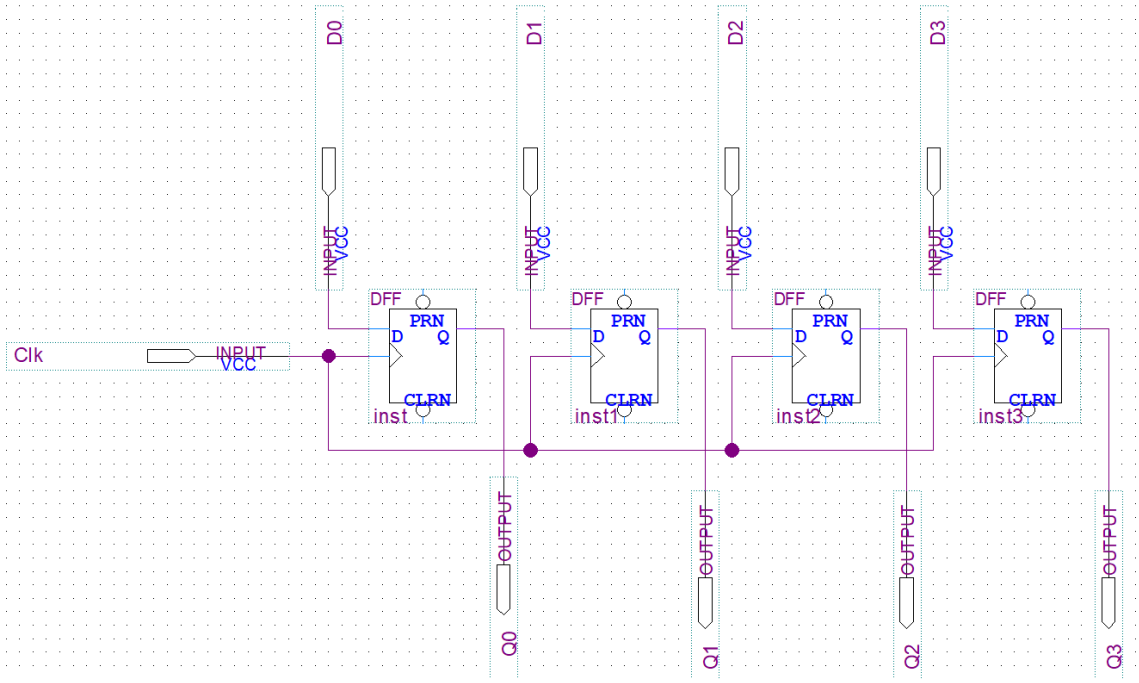
Signal	Name	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
In	SW0	PIN_N25	PIN_AB12
Q0	LEDR3	PIN_ AC22	PIN_V18
Q1	LEDR2	PIN_ AB21	PIN_V17
Q2	LEDR1	PIN_ AF23	PIN_W16
Q3	LEDR0	PIN_ AE23	PIN_V16

3. Test the functionality of your designed circuit. (First Set **In** than apply **Clk**)

INPUT		**OUTPUT**			
Clk	**In**	**Q0**	**Q1**	**Q2**	**Q3**
x	x	0	0	0	0
1st	1	1	0	0	0
2nd	1	1	1	0	0
3rd	0	0	1	1	0
4th	0	0	0	1	1

Parallel in Parallel out Shift Register

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



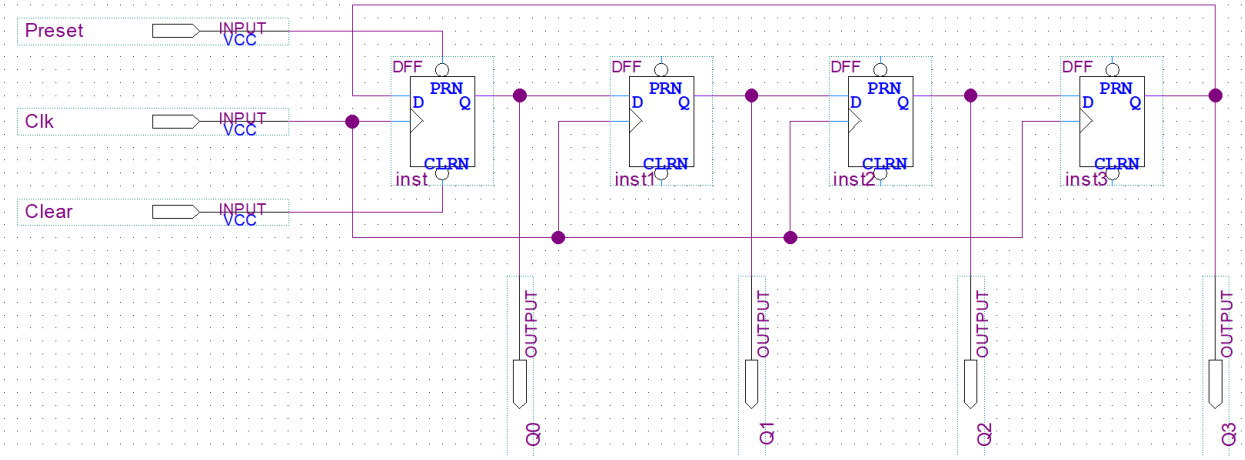
2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Name	Cyclone II	Cyclone V
D3	SW0	PIN_N25	PIN_AB12
D2	SW1	PIN_N26	PIN_AC12
D1	SW2	PIN_P25	PIN_AF9
D0	SW3	PIN_AE14	PIN_AF10
Clk	KEY3	PIN_W26	PIN_Y16
Q3	LEDR0	PIN_AE23	PIN_V16
Q2	LEDR1	PIN_AF23	PIN_W16
Q1	LEDR2	PIN_AB21	PIN_V17
Q0	LEDR3	PIN_AC22	PIN_V18

3. Verify the functionality in FPGA board.

Ring Counter:

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Switch No.	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
Clear	SW0	PIN_N25	PIN_AB12
Preset	SW1	PIN_N26	PIN_AC12
Q0	LEDR3	PIN_ AC22	PIN_V18
Q1	LEDR2	PIN_ AB21	PIN_V17
Q2	LEDR1	PIN_ AF23	PIN_W16
Q3	LEDR0	PIN_ AE23	PIN_V16

3. Follow this process –

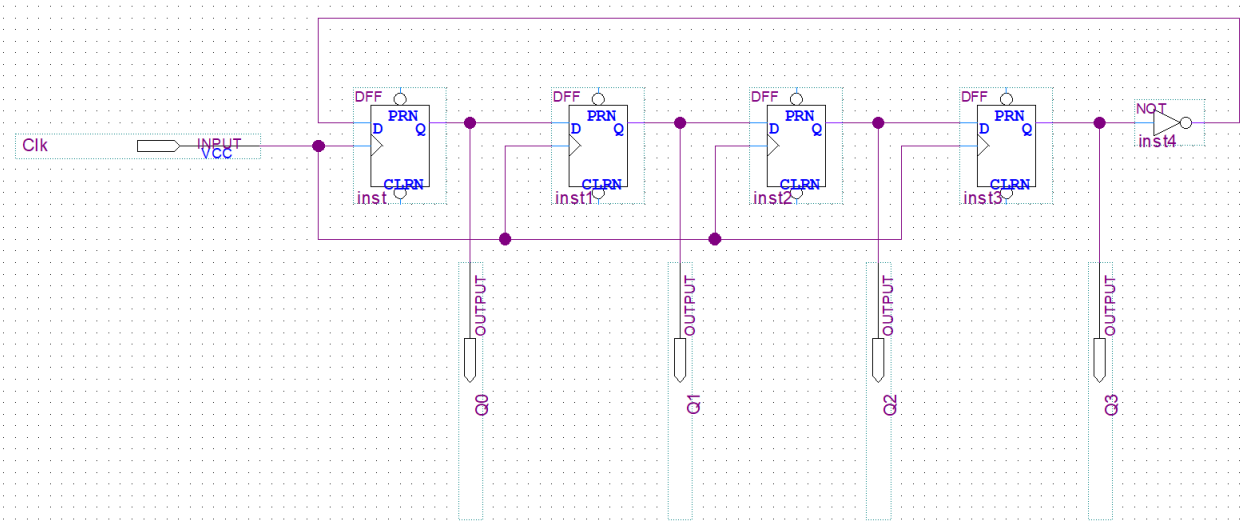
- Clear (Clear = 0, Preset = 1)
- Disable (Clear = 1, Preset = 1)
- Preset (Clear = 1, Preset = 0)
- Disable (Clear = 1, Preset = 1)
- Start Clock Cycles

4. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

Sequence: 1000 → 0100 → 0010 → 0001 → 1000 → …

Johnson Counter:

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

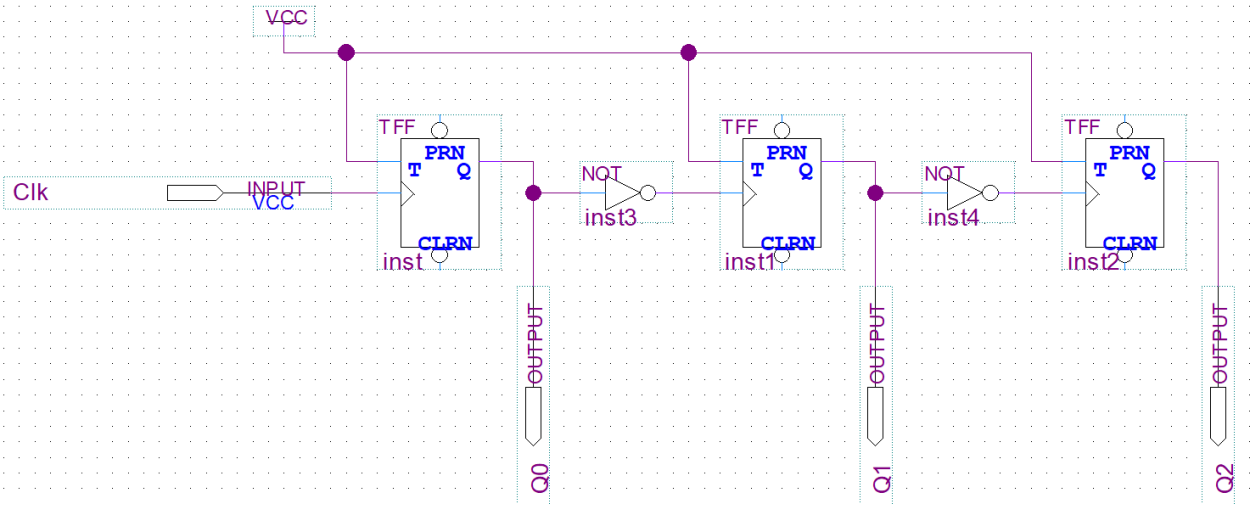
Signal	Switch No.	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
Q0	LEDR3	PIN_ AC22	PIN_V18
Q1	LEDR2	PIN_ AB21	PIN_V17
Q2	LEDR1	PIN_ AF23	PIN_W16
Q3	LEDR0	PIN_ AE23	PIN_V16

3. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

Sequence: 0000 → 1000 → 1100 → 1110 → 1111 → 0111 → 0011 → 0001 → 0000 → …

Up Counter:

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

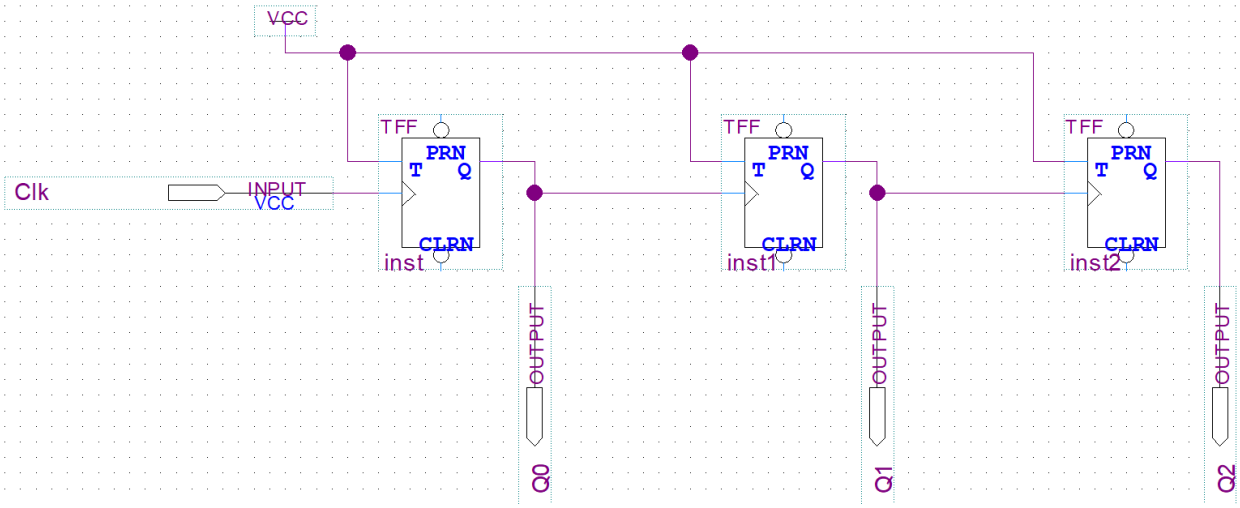
Signal	Switch No.	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
Q2	LEDR2	PIN_ AB21	PIN_V17
Q1	LEDR1	PIN_ AF23	PIN_W16
Q0	LEDR0	PIN_ AE23	PIN_V16

3. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

Sequence: 000 → 001 → 010 → 011 → 100 → 101 → 110 → 111 → 000 → …

Down Counter:

1. Create a new project and create a new block diagram/schematic file. After completing the schematic, it should look like the following:



2. Compile and simulate the schematic. If everything is ok, assign pins as follows:

Signal	Switch No.	Cyclone II	Cyclone V
Clk	KEY3	PIN_W26	PIN_Y16
Q2	LEDR2	PIN_ AB21	PIN_V17
Q1	LEDR1	PIN_ AF23	PIN_W16
Q0	LEDR0	PIN_ AE23	PIN_V16

3. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

Sequence: 111 → 110 → 101 → 100 → 011 → 010 → 001 → 000 → 111 → …

Verilog Code

Serial in Serial Out Shift Register

Simple Version

```
module Shift_reg (In, Clk, Q);
        input In, Clk;
        output reg [0:3]Q;

        always @ (posedge Clk)
        begin
                Q[0] <= In;
                Q[1] <= Q[0];
                Q[2] <= Q[1];
                Q[3] <= Q[2];
        end

endmodule
```

Compact Version

```
module Shift_reg (In, Clk, Q);
        input In, Clk;
        output reg [0:3]Q;

        always @ (posedge Clk)
                Q <= { In, Q[0:2] };

endmodule
```

Loop Version

```
module Shift_reg (In, Clk, Q);
        input In, Clk;
        output reg [0:3]Q;
        integer i; // Loop Variable (Must be declared outside)

        always @ (posedge Clk)
        begin
                Q[0] <= In;
                for(i = 1; i <= 3; i = i + 1) // i++ NOT supported
                        Q[i] <= Q[i-1];
        end

endmodule
```

Parallel in Parallel Out Shift Register

```
module Parallel_reg (Clk, D, Q);
        input Clk;
        input [0:3]D;
        output reg [0:3]Q;

        always @ (posedge Clk)
                Q <= D;

endmodule
```

Ring Counter

Simple Version

```
module Ring_c(Preset, Clear, Clk, Q);
	input Preset, Clear, Clk;
	output reg [0:3]Q;

	always @ (posedge Clk, negedge Preset, negedge Clear)
	begin
		if (!Clear) // Clear == 0
			Q[0] <= 0;
		else if (!Preset) // Preset == 0
			Q[0] <= 1;
		else
		begin
			Q[0] <= Q[3];
			Q[1] <= Q[0];
			Q[2] <= Q[1];
			Q[3] <= Q[2];
		end
	end

endmodule
```

Compact Version

```
module Ring_c(Preset, Clear, Clk, Q);
	input Preset, Clear, Clk;
	output reg [0:3]Q;

	always @ (posedge Clk, negedge Preset, negedge Clear)
	begin
		if(!Clear)
			Q[0] <= 0;
		else if(!Preset)
			Q[0] <= 1;
		else
			Q <= {Q[3], Q[0:2]};
	end

endmodule
```

Johnson Counter

Simple Version

```
module Johnson_c(Clk, Q);
        input Clk;
        output reg [0:3]Q;

        always @ (posedge Clk)
        begin
                Q[0] <= ~Q[3];
                Q[1] <=  Q[0];
                Q[2] <=  Q[1];
                Q[3] <=  Q[2];
        end

endmodule
```

Compact Version

```
module Johnson_c(Clk, Q);
        input Clk;
        output reg [0:3]Q;

        always @ (posedge Clk)
                Q <= {~Q[3], Q[0:2]};

endmodule
```

Binary Counter

Up Counter

```
// 3-bit Up Counter
// Counts continuously: 0 -> 1 -> ... -> 7 -> 0 -> ...

module Up_counter (Clk, Q);
        input Clk;
        output reg [2:0]Q;

        always @ (posedge Clk)
                Q <= Q + 1;

endmodule
```

Down Counter

```
// 3-bit Down Counter
// Counts continuously: 7 -> 6 -> ... -> 0 -> 7 -> ...

module Down_counter (Clk, Q);
        input Clk;
        output reg [2:0] Q = 3'b111; // initialize to 7 (Optional)

        always @ (posedge Clk)
                Q <= Q - 1;

endmodule
```