



EEE – 2104

Digital Circuit Design Lab

AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT
OF
ELECTRICAL AND ELECTRONIC ENGINEERING

EEE- 2104

Digital Circuit Design Lab

Edition 2025

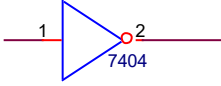
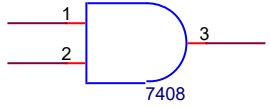
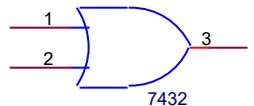
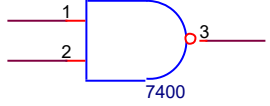
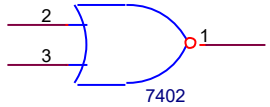
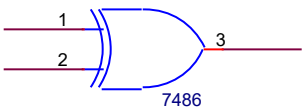
Table of Contents

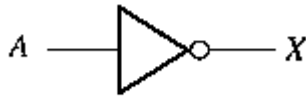
Table of Contents	3
Experiment: 1	4
Experiment: 2	12
CAD System.....	16
Experiment: 3	43
Experiment: 4	50
Experiment: 5	57
Experiment: 6	70
Experiment: 7	77
Experiment: 8	87
Experiment: 9	93
Experiment: 10	96
ANNEXURE I	99
ANNEXURE II.....	103
REFERENCE.....	109

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 pin number

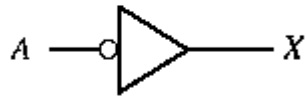
IC number	IC Name	Gate Name	Gate Schematic view
7404	Hex Inverter	NOT/INVERTER	
7408	Quad 2-input AND	AND	
7432	Quad 2-input OR	OR	
7400	Quad 2-input NAND	NAND	
7402	Quad 2-input NOR	NOR	
7486	Quad 2-input XOR	XOR	

The INVERTER/NOT Gate

A	X
0	1
1	0

$$X = \bar{A}$$

Boolean expression



Truth table

0 = LOW

1 = HIGH

Distinctive shape symbols

The output of an inverter is always the complement (opposite) of the input.

The AND Gate

Distinctive shape symbol

$$X = AB$$

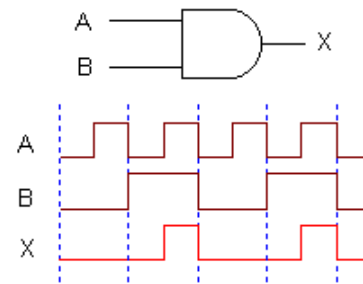
Boolean expression

B	A	X
0	0	0
0	1	0
1	0	0
1	1	1

Truth table

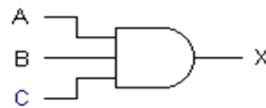
0 = LOW

1 = HIGH



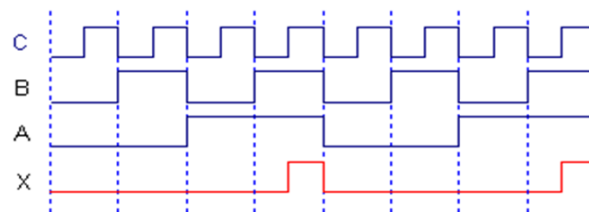
Pulsed Waveforms

The output of an AND gate is HIGH only when all inputs are HIGH.



$$X = ABC$$

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



3 Input AND Gate

The OR Gate



Distinctive shape symbol

$$X = A + B$$

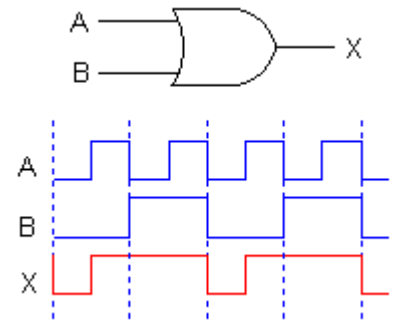
Boolean expression

B	A	X
0	0	0
0	1	1
1	0	1
1	1	1

Truth table

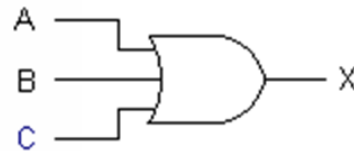
0 = LOW

1 = HIGH



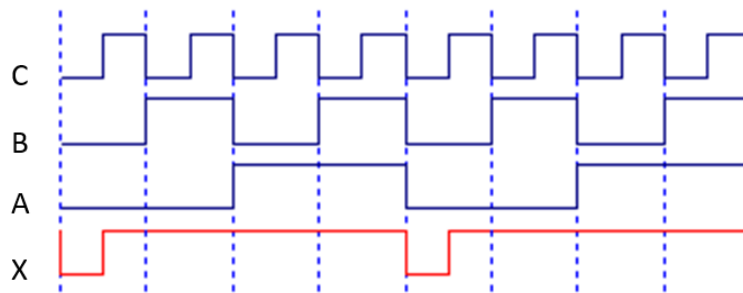
Pulsed Waveforms

The output of an OR gate is HIGH whenever one or more inputs are HIGH.



$$X = A + B + C$$

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

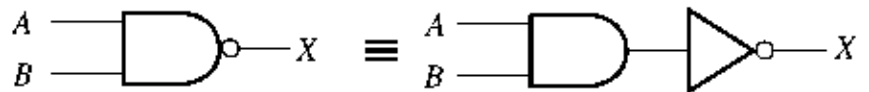


3 Input OR Gate

The NAND Gate



Distinctive shape symbol



B	A	X
0	0	1
0	1	1
1	0	1
1	1	0

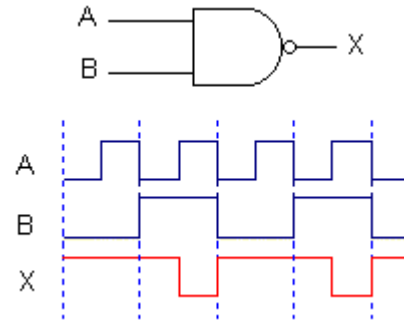
Truth table

0 = LOW

1 = HIGH

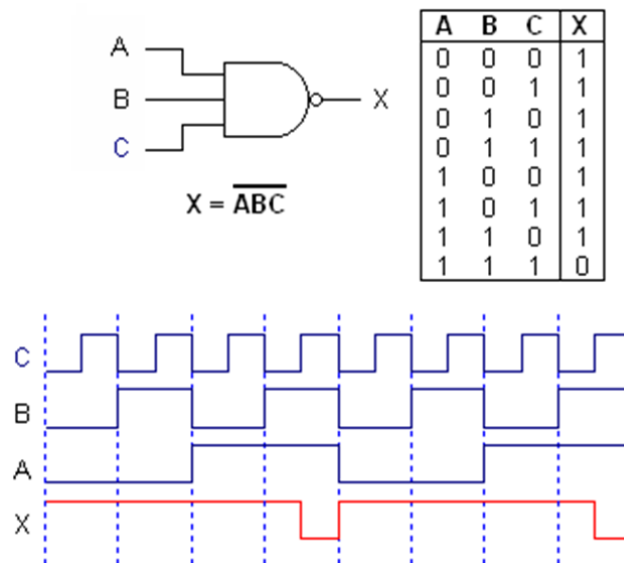
$$X = \overline{AB}$$

Boolean expression



Pulsed Waveforms

The output of a NAND gate is HIGH whenever one or more inputs are LOW.

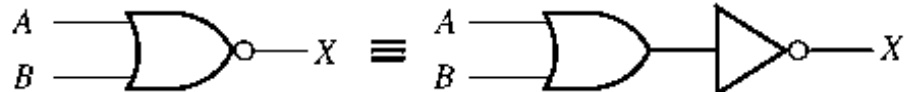


3 Input NAND Gate

The NOR Gate



Distinctive shape symbol



B	A	X
0	0	1
0	1	0
1	0	0
1	1	0

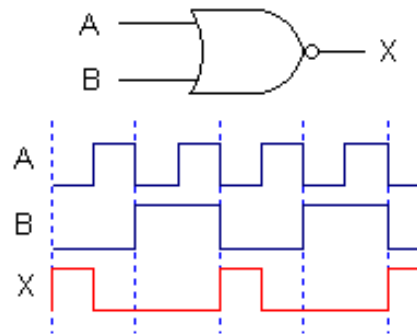
Truth table

0 = LOW

1 = HIGH

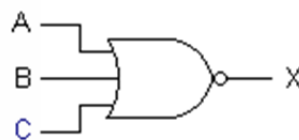
$$X = \overline{A + B}$$

Boolean expression



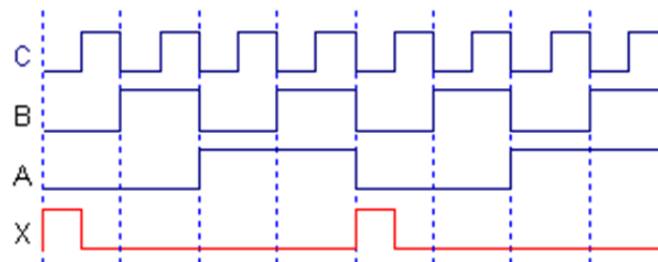
Pulsed Waveforms

The output of a NOR gate is LOW whenever one or more inputs are HIGH.



$$X = \overline{A + B + C}$$

A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



3 Input NOR Gate

Exclusive-OR Gate



Distinctive shape symbol

B	A	X
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

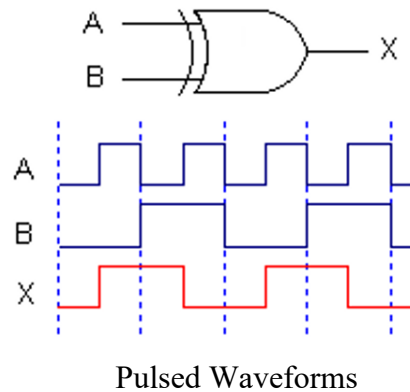
0 = LOW

1 = HIGH

$$X = A \oplus B$$

Boolean expression

The output of an XOR gate is HIGH whenever the two inputs are different.



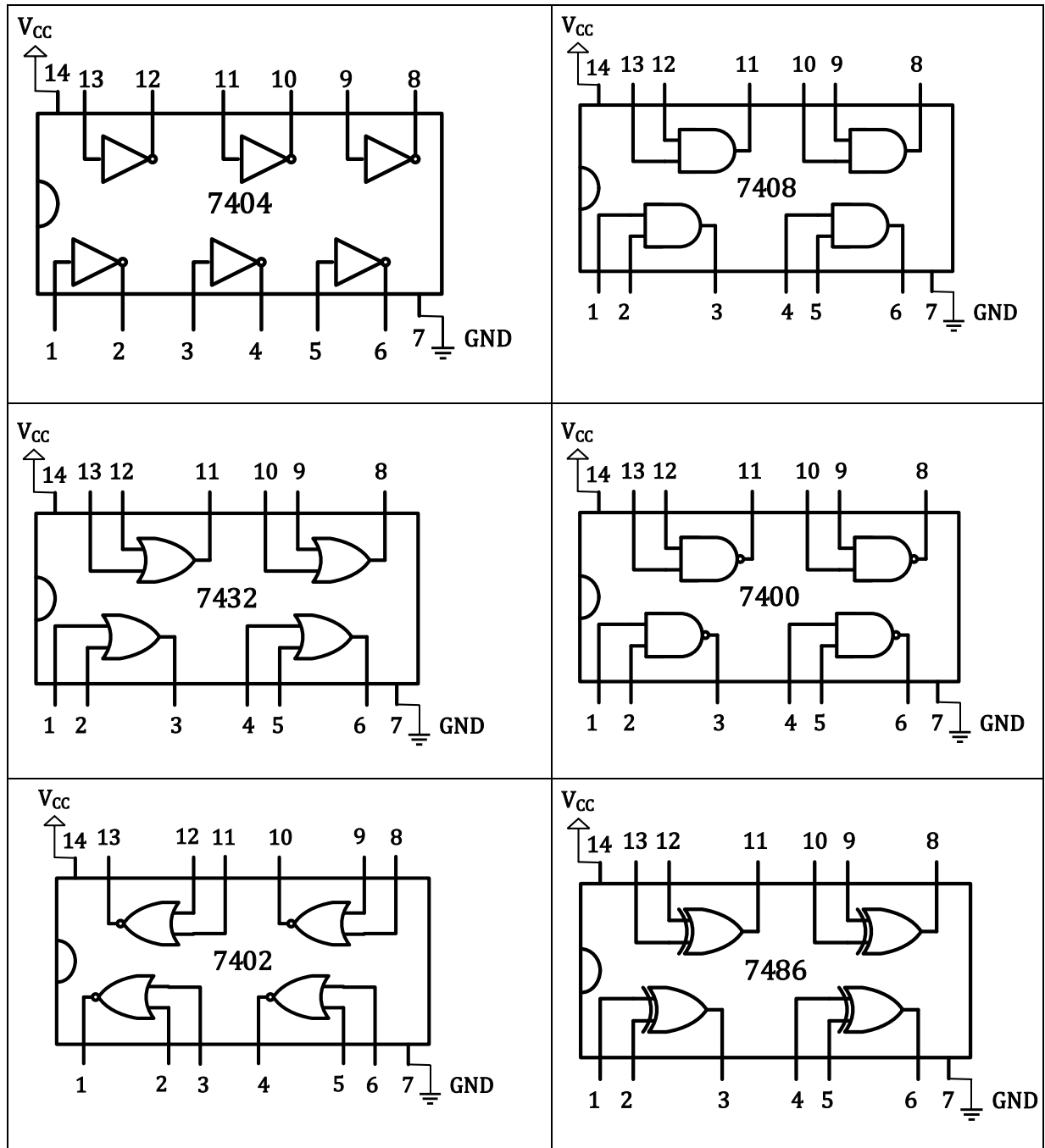
Equipment:

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	Ground 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



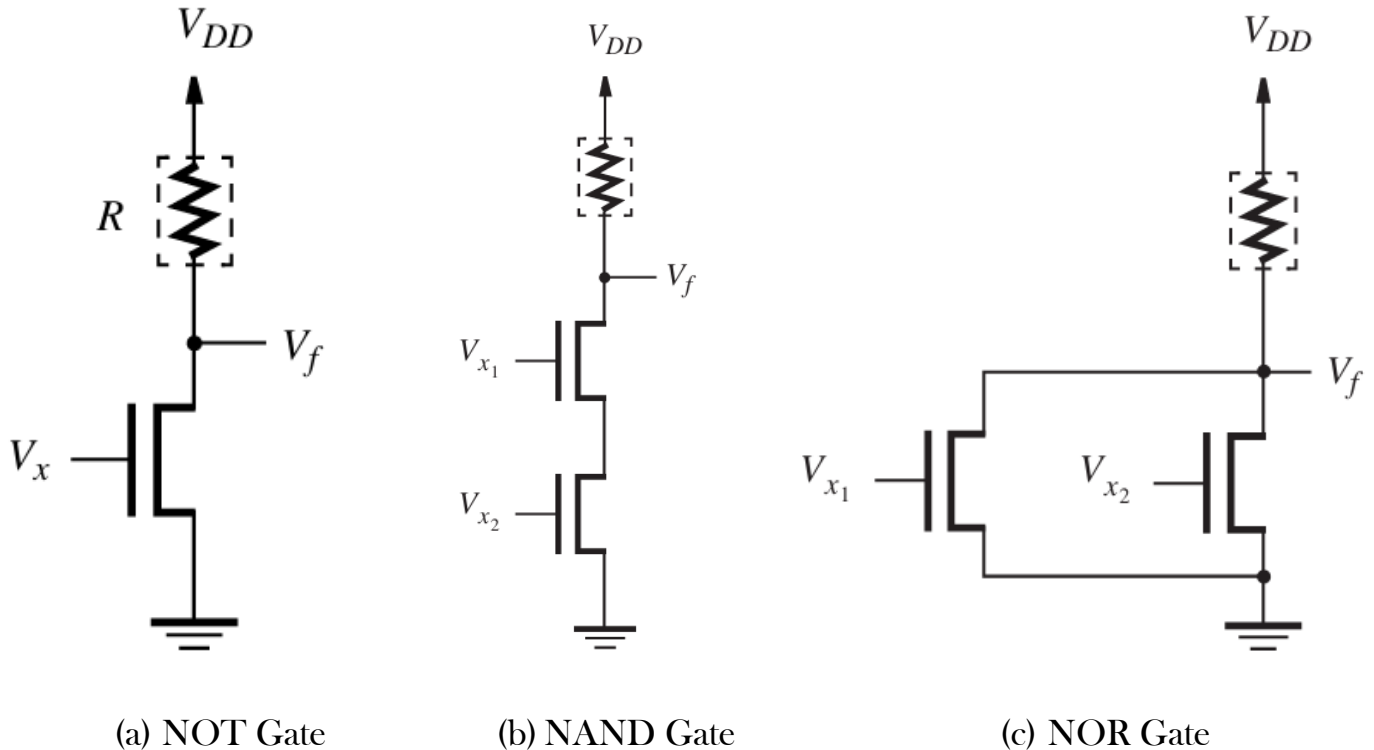
2. Note the number of gates each IC has from the handbook.

3. Now fill up the following table:

Input A	Input B	7404 NOT $Y = \overline{A}$	7432 OR $Y = A + B$	7402 NOR $Y = \overline{A + B}$	7486 XOR $Y = A \oplus B$	7408 AND $Y = AB$	7400 NAND $Y = \overline{AB}$
0	0						
0	1						
1	0						
1	1						

Implementation Technology:

The first schemes for building logic gates with MOSFETs became popular in the 1970s and relied on either PMOS or NMOS transistors. Here, we will learn how logic circuits can be built using NMOS. Such circuits are known as NMOS circuits. Here we will use the concept of transistor switching to understand the basic principle of logic gates implementation.



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 –

$$x = x.1$$

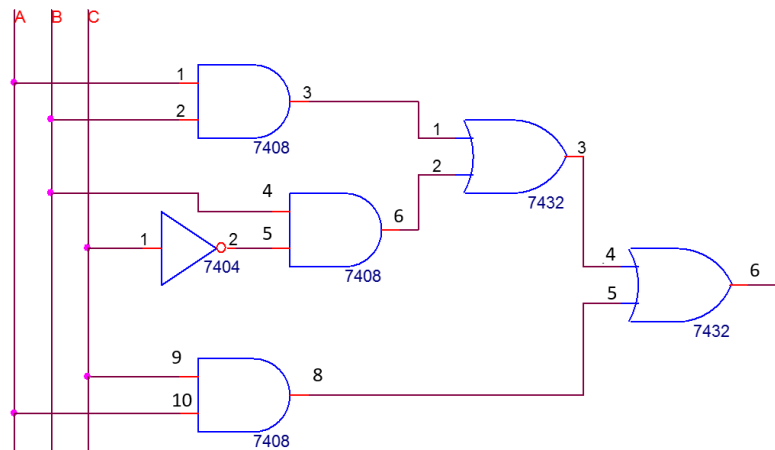
$$x + x' = 1$$

$$x + x = x$$

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

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:

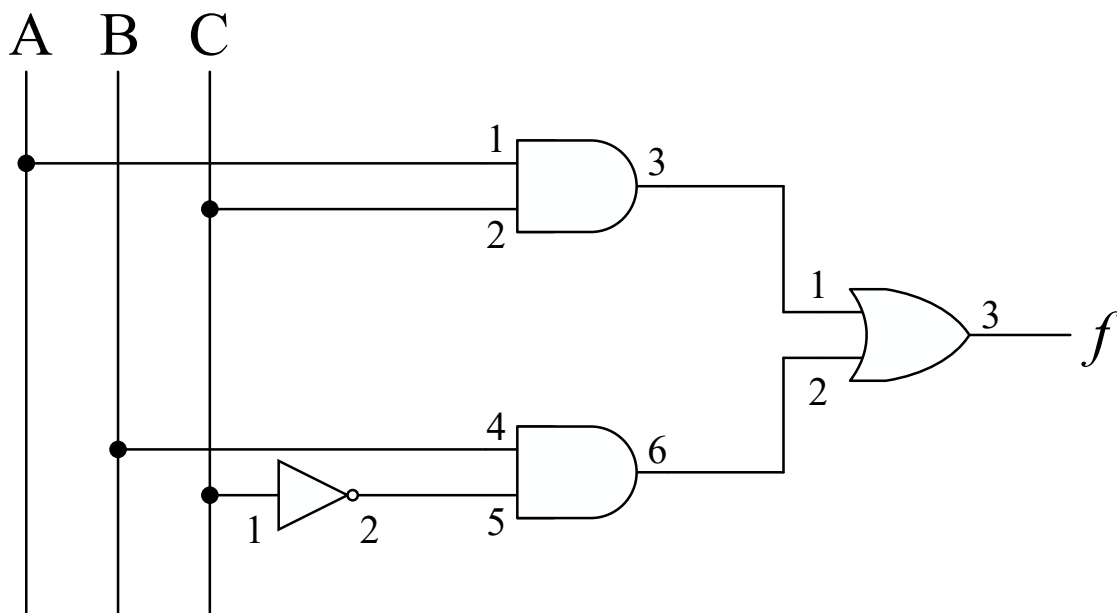
$$f = AB + BC' + CA$$

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:

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.

Report

1. $f(A, B, C, D) = \sum m(0, 1, 4, 5, 12, 13, 14)$
2. $f(A, B, C) = \prod M(0, 1, 3, 4)$

For both the functions do the following

- Simplify the function in POS form and in SOP form.
- Draw logic diagram to implement the function.
- Select ICs and mention the pin number in the diagram.
- Show the truth table of the system.

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

- CAD Tool: Quartus II (Version - 13.0sp1 Web Edition)
- Hardware Description Language (HDL): Verilog
- Development Board: Altera FPGA Board

Procedure:

For installation procedure, go to Annexure I.

Design Flow:

1. Start:

Project

2. Design:

Schematic

Verilog

3. Verification:

Simulation

FPGA Board

Function Implementation:

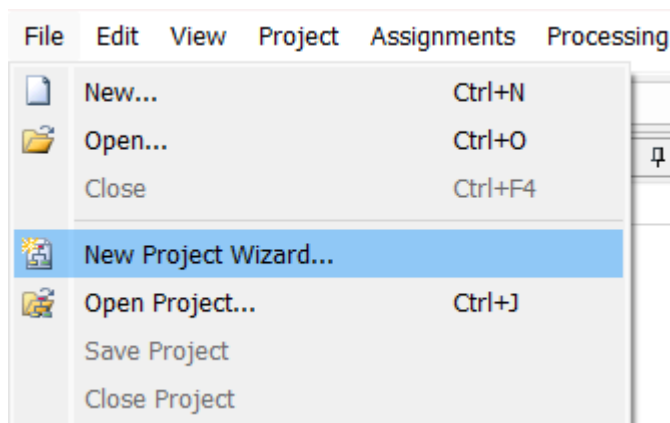
Here, we are going to implement the following function –

$$f = AC + BC'$$

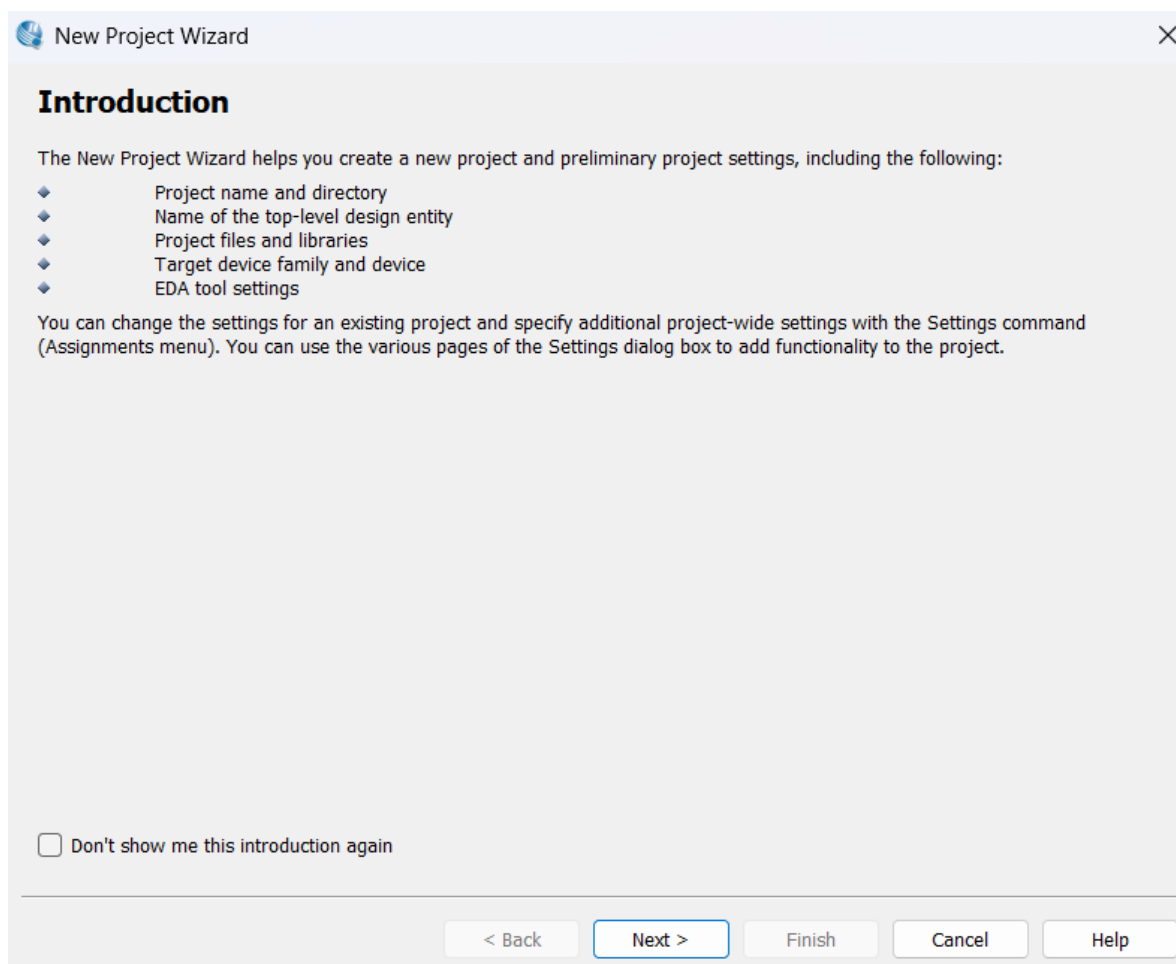
Project in Quartus II:

P1. Open **Quartus II 13.0 sp1 web edition**.

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\DL D) 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.**

New Project Wizard

Directory, Name, Top-Level Entity [page 1 of 5]

What is the working directory for this project?

D:/Sumit ...

What is the name of this project?

SOPfund ...

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

SOPfunc ...

Use Existing Project Settings...

< Back Next > Finish Cancel Help

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 **Cyclone II** under **Device family** and type **EP2C35F672C6** in **Name filter**. Double click on it under **Available devices** so that **Specific device selected in “Available devices’ list** is selected under **Target Device**.

Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

Device family

Family: Cyclone II

Devices: All

Target device

☐ Auto device selected by the Fitter

☒ Specific device selected in 'Available devices' list

☐ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Speed grade: Any

Name filter: EP2C35F672C6

☒ Show advanced devices ☐ HardCopy compatible only

Available devices:

Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements
EP2C35F672C6	1.2V	33216	475	483840	70

Companion device

HardCopy:

☐ Limit DSP & RAM to HardCopy device resources

< Back

Next >

Finish

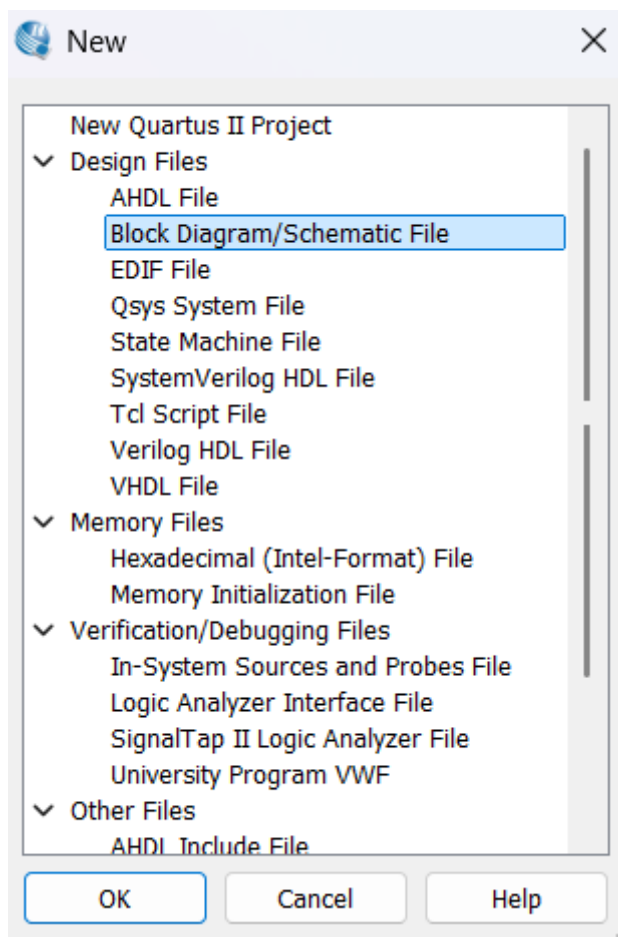
Cancel

Help

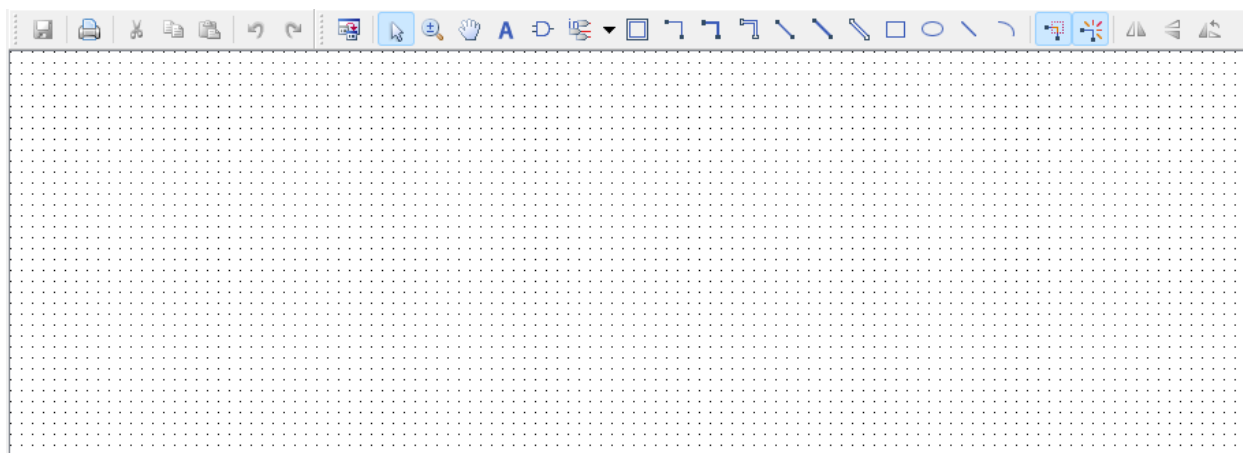
P7. Click **Finish**. This completes the steps for creating a project file.

Block Diagram/Schematic Design:

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



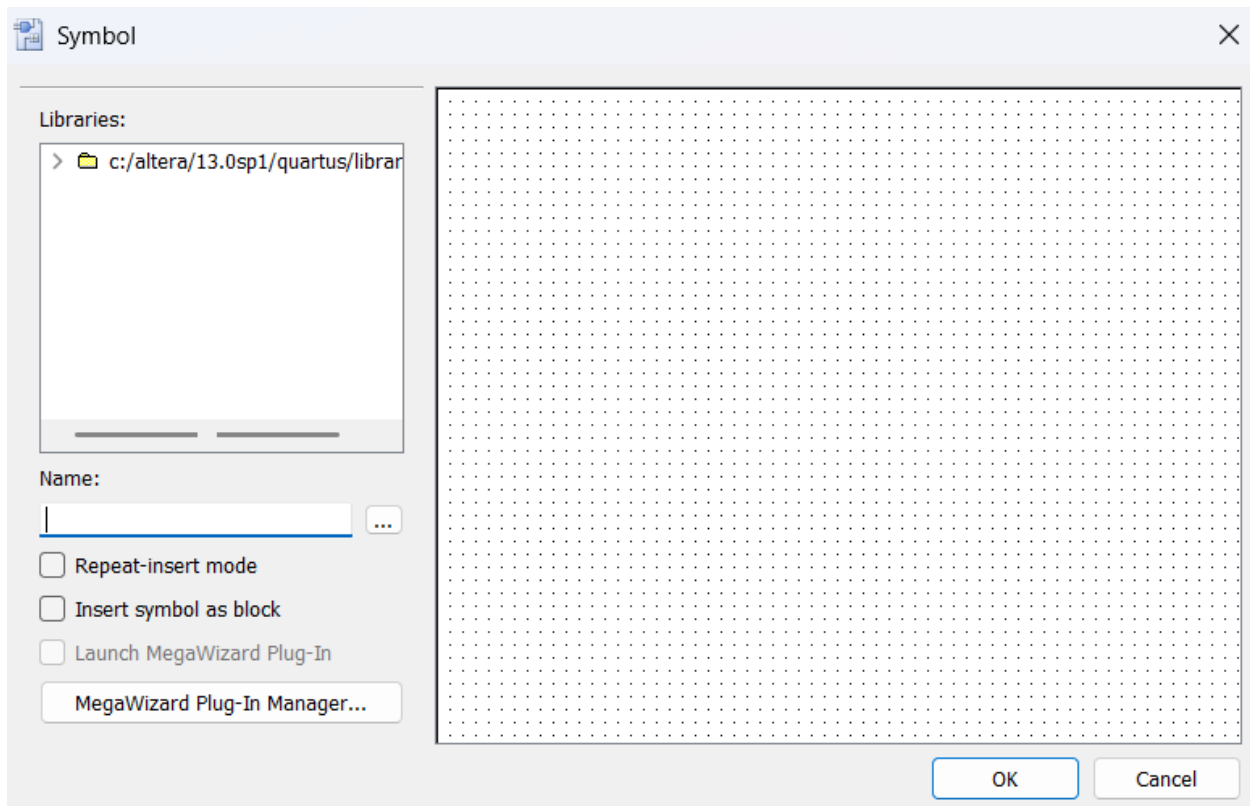
A blank block diagram window will appear.



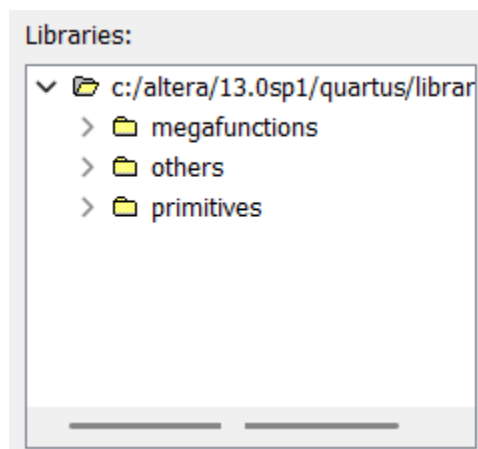
B2. To implement the circuit, we will need an AND, OR & NOT gate. From the menu bar, click on icon for **Symbol Tool**, or alternatively double click on the blank schematic window.



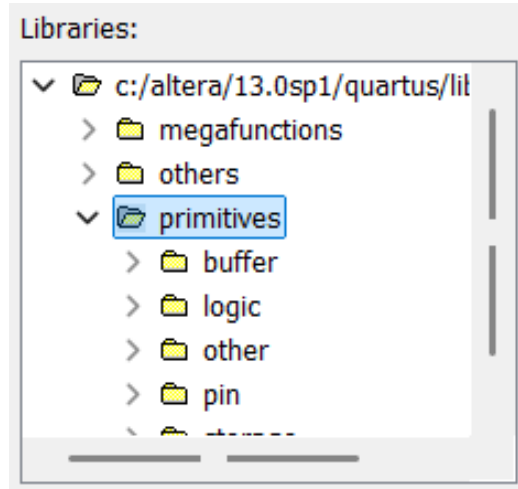
B3. The following window will appear. Under **Libraries**, click on the plus icon beside **c:/altera/...**



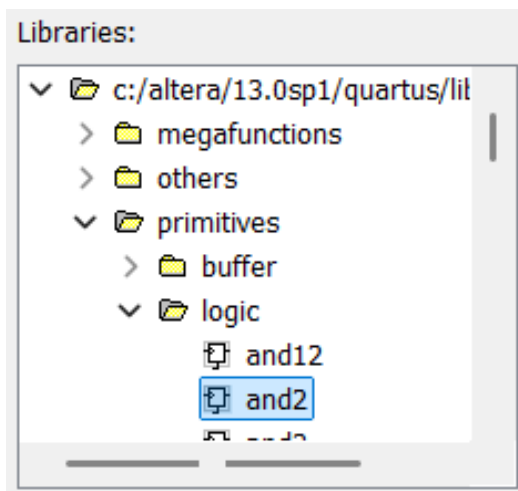
B4. After expanding the plus icon, you will see the following library directories:



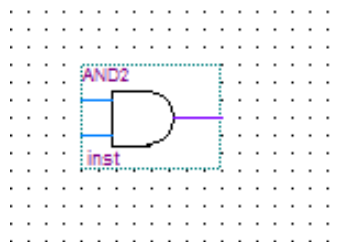
B5. Click on the plus sign beside **primitives**.



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**.

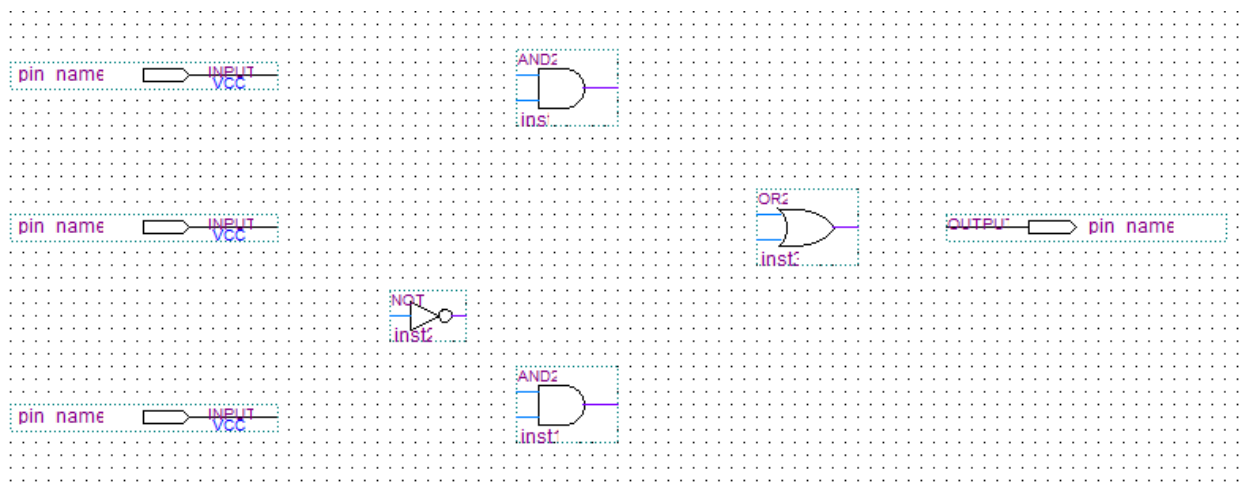


B7. Then go to block diagram window and place the symbol on it.



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.

Pin Properties

General Format

To create multiple pins, enter a name in AHDL bus notation
(For example: "name[3..0]"), or enter a comma-seperated list of names.

Pin name(s):

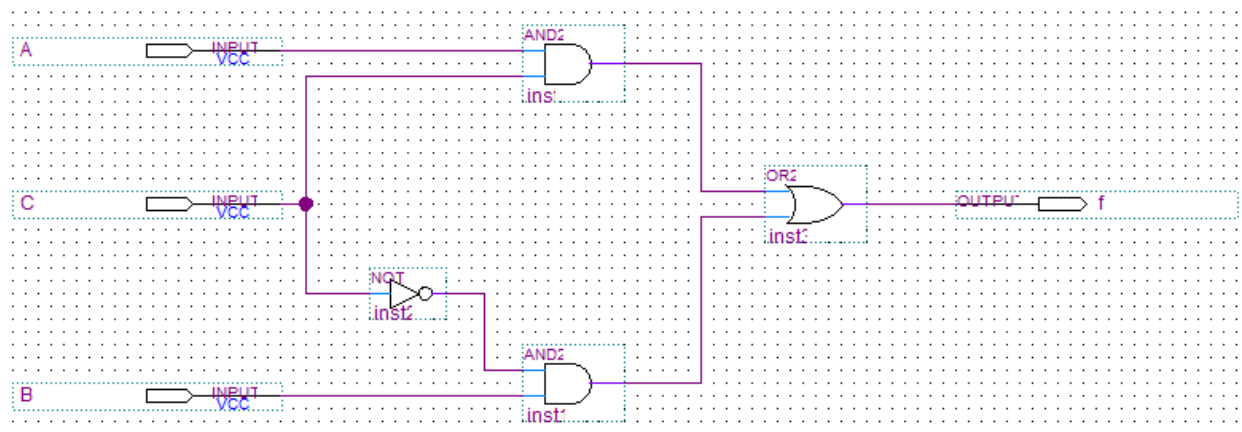
Default value:

OK Cancel Help

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.



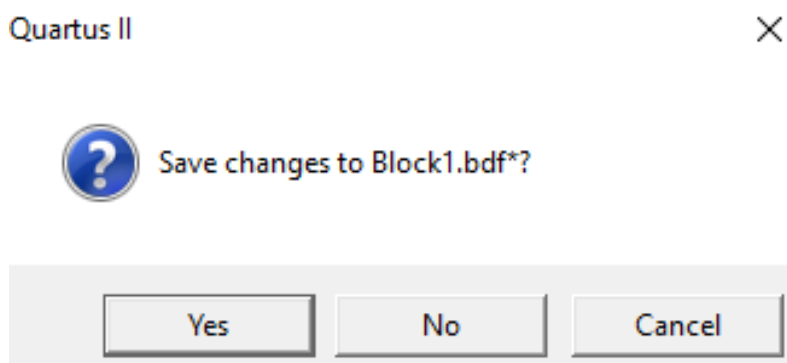
B12. Now, wire the gates and pins to construct the circuit. When completed, it should look like the following –



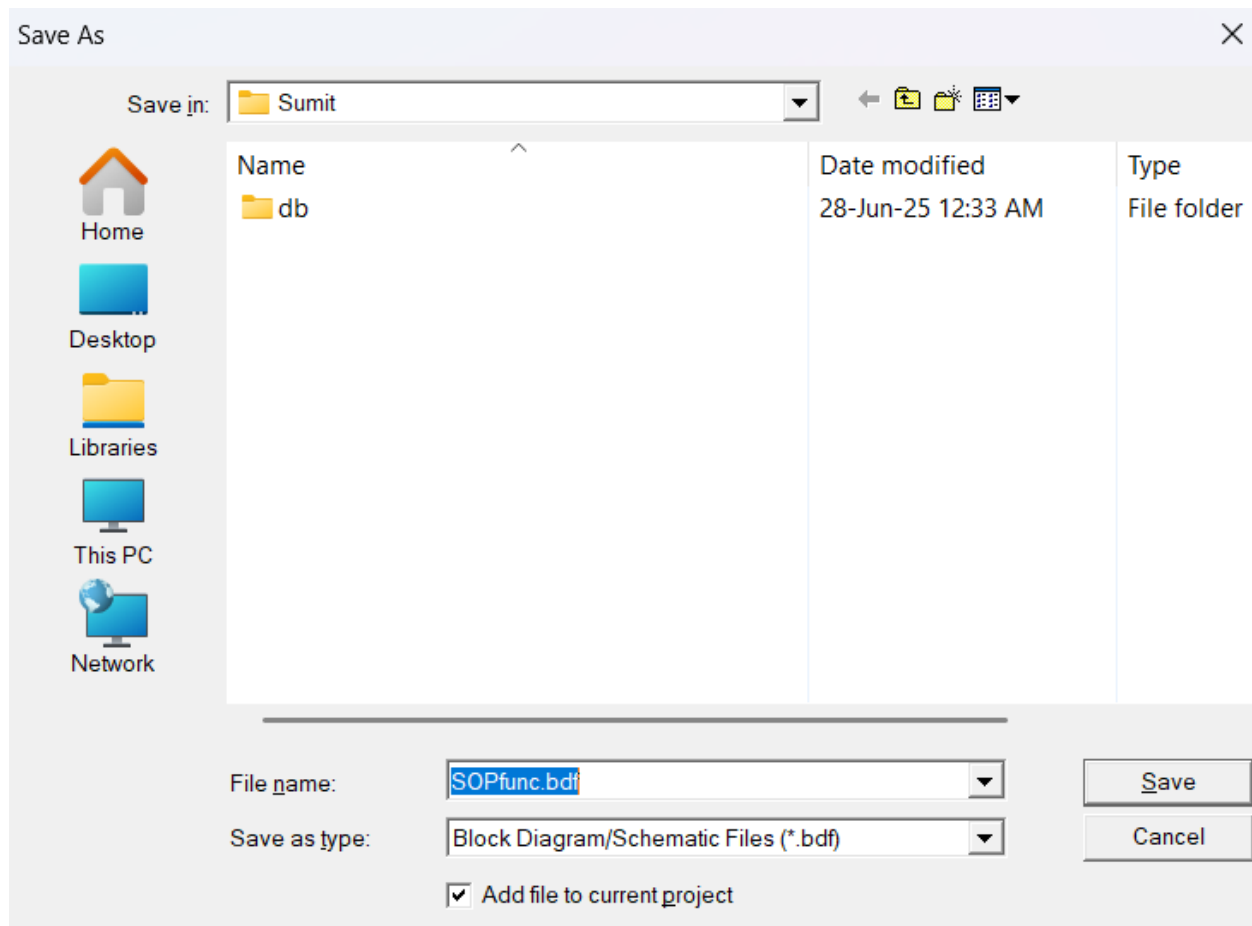
B13. Click on **start compilation** button on the top menu bar.



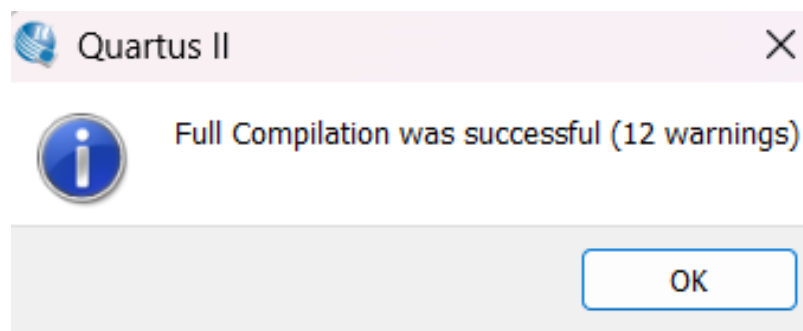
B14. Click **Yes**, if you are prompted to save the block diagram.



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:

Table of Contents		Flow Summary
Flow Summary		Flow Status
Flow Settings		Quartus II 64-Bit Version
Flow Non-Default Global Settings		Revision Name
Flow Elapsed Time		Top-level Entity Name
Flow OS Summary		Family
Flow Log		Device
> Analysis & Synthesis		Timing Models
> Fitter		Total logic elements
> Flow Messages		Total combinational functions
> Flow Suppressed Messages		Dedicated logic registers
> Assembler		Total registers
> TimeQuest Timing Analyzer		Total pins
		Total virtual pins
		Total memory bits
		Embedded Multiplier 9-bit elements
		Total PLLs

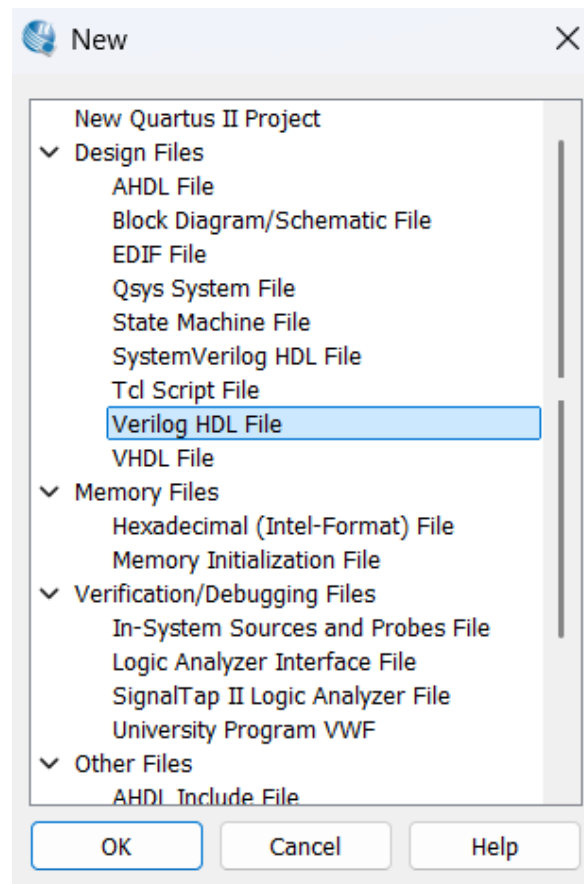
If any error occurs, you will find them at the bottom window.

Type	Message
Info	Running Quartus II Analysis & Synthesis
Info	Command: quartus_map --read_settings_files=on --write_settings_files=off halfadder -c halfadder
Info	Found 1 design units, including 1 entities, in source file halfadder.bdf
Info	Elaborating entity "halfadder" for the top level hierarchy
Error	Node "inst1" is missing source
Error	Quartus II Analysis & Synthesis was unsuccessful. 1 error, 0 warnings
Error	Quartus II Full Compilation was unsuccessful. 3 errors, 0 warnings

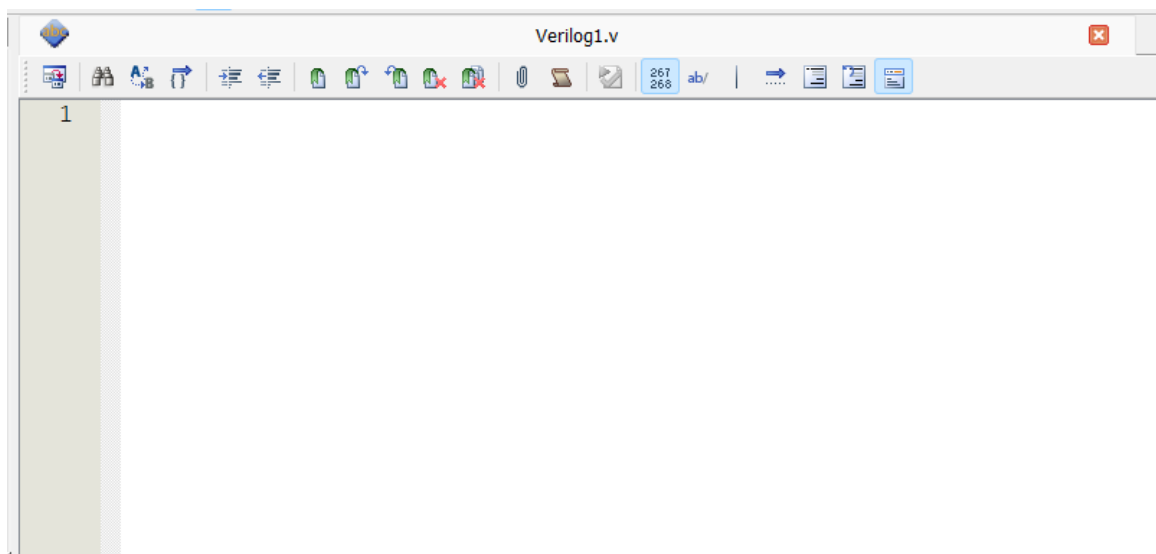
In case an error occurs, find the error on the block diagram and rerun compilation.

Verilog HDL File:

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

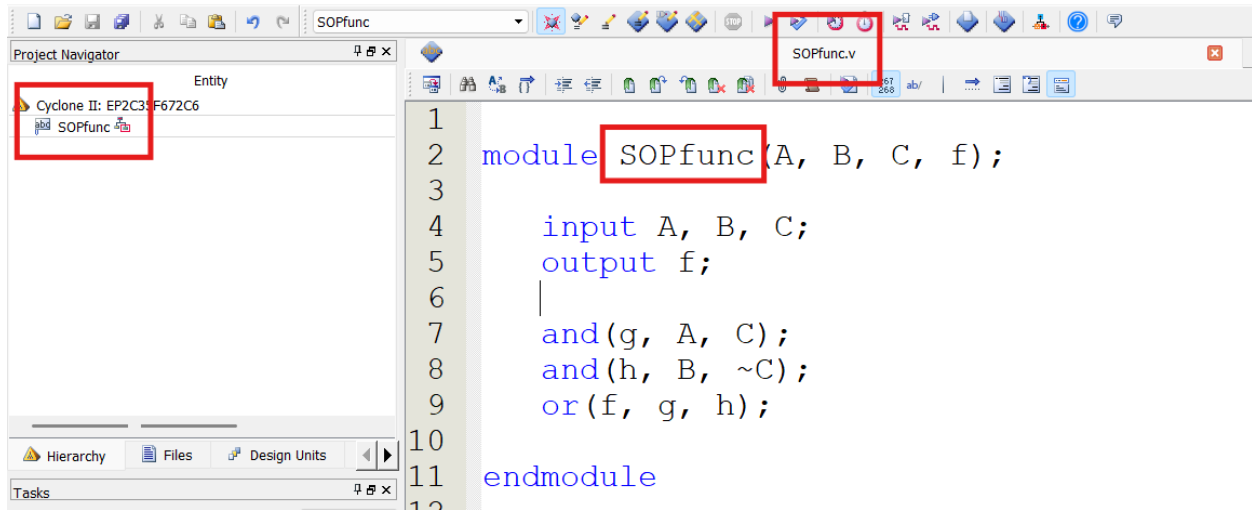


V2. A blank window will appear where we will write our Verilog HDL code.



V3. Complete the Code and Save the File.

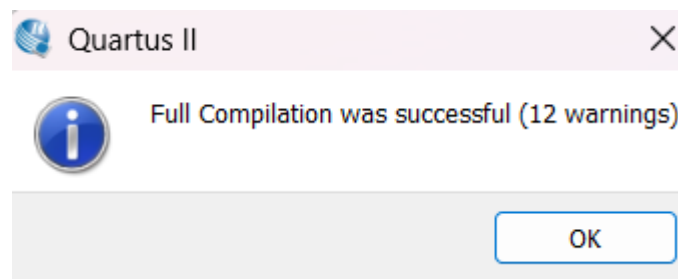
*** **Make sure Module Name, Verilog File Name, Top-level Entity (Project) Name is exactly same.** (i.e. SOPfunc in these 3 cases)



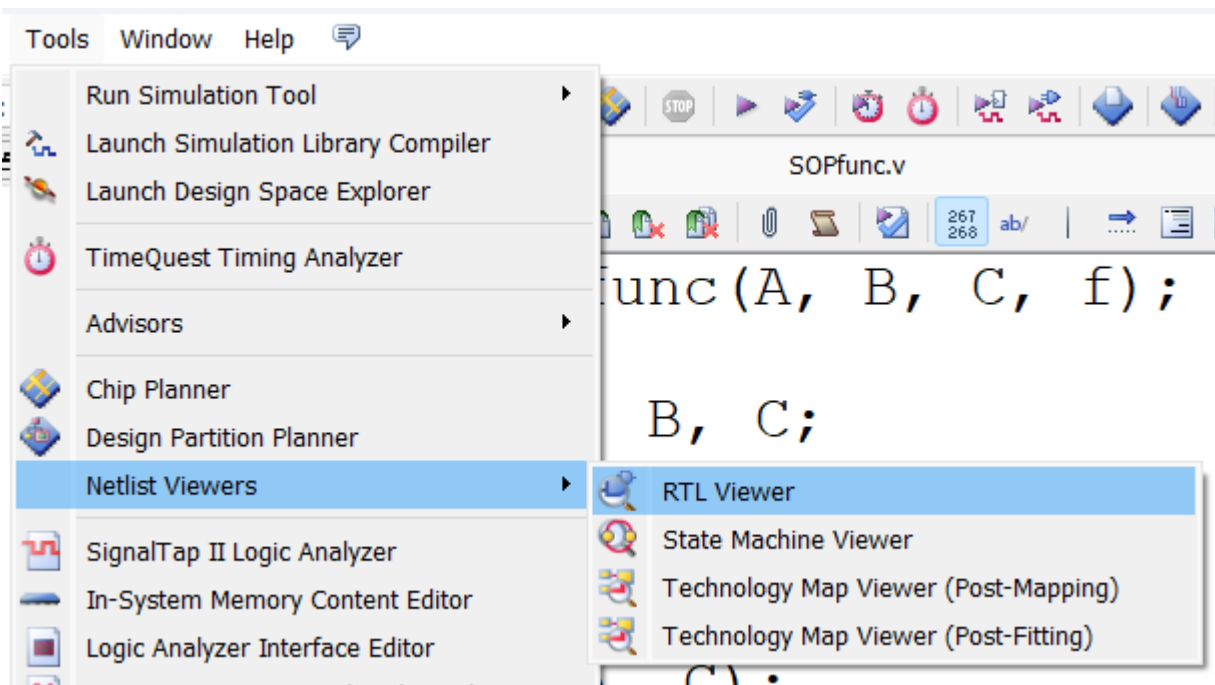
V4. Click on **start compilation** button on the top menu bar.



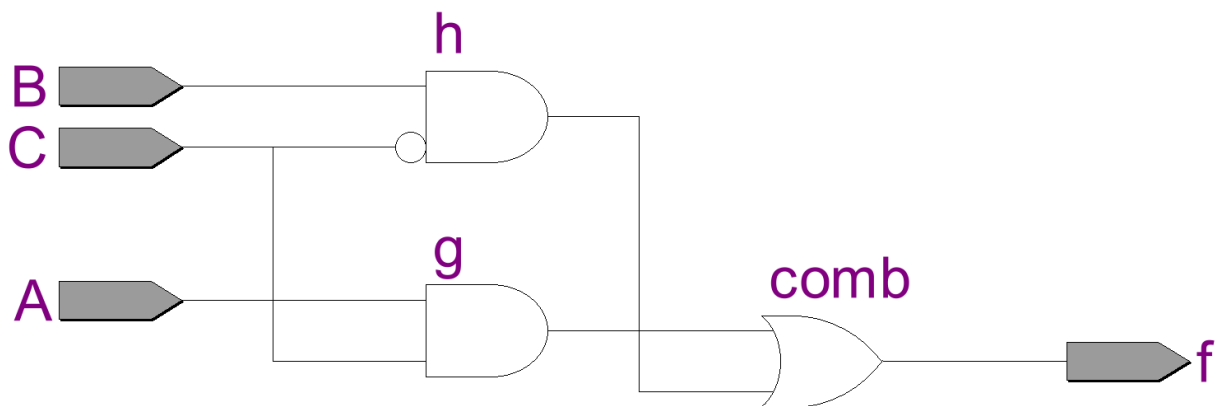
V5. If compilation is successful, you will get a message like the following. Click **OK**.



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

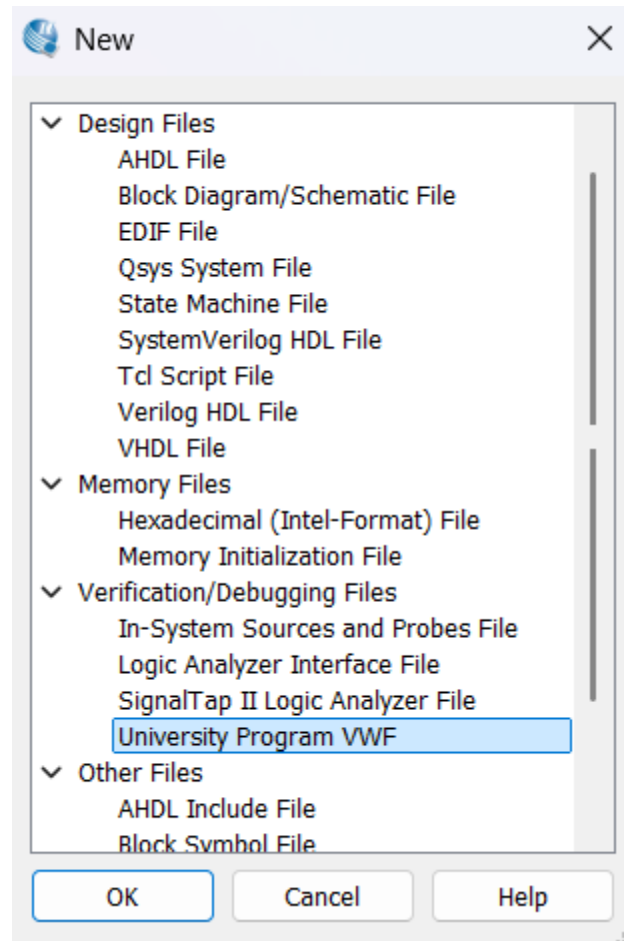


V7. You can see the RTL (Register Transfer Level) view of your Verilog Code.

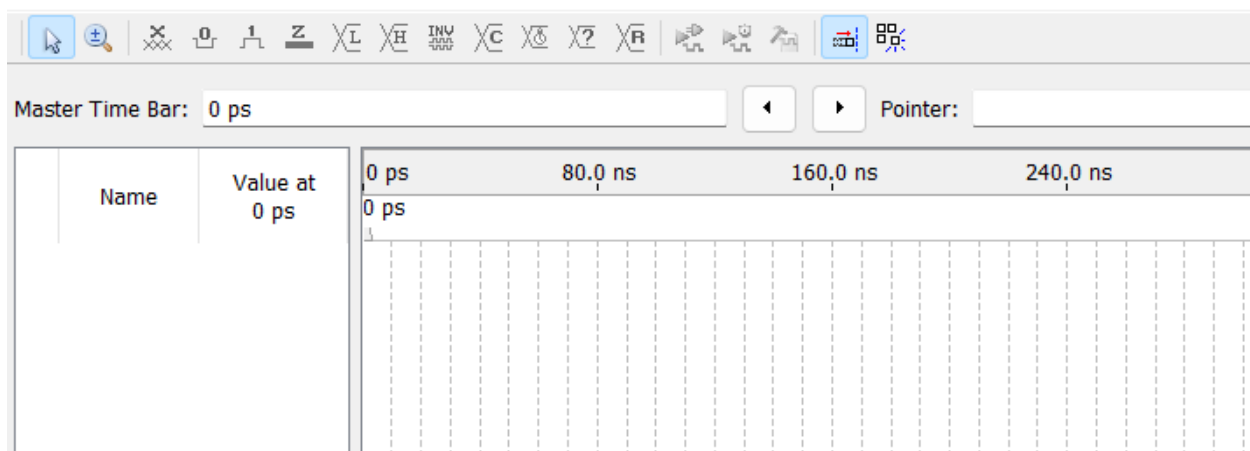


Simulation:

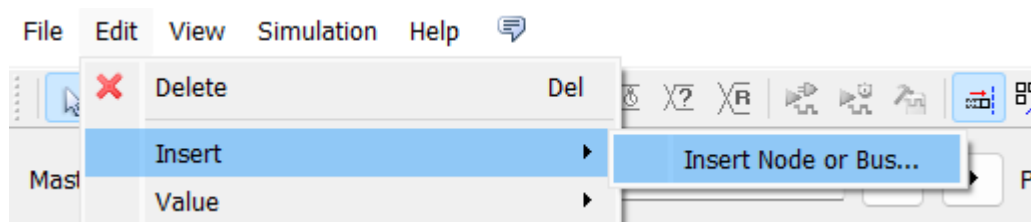
S1. Go to **File** → **New** to create a Vector Waveform File (VWF) which is required for simulating inputs and outputs. Select **University Program VWF** under **Verification/Debugging Files** and click **OK**.



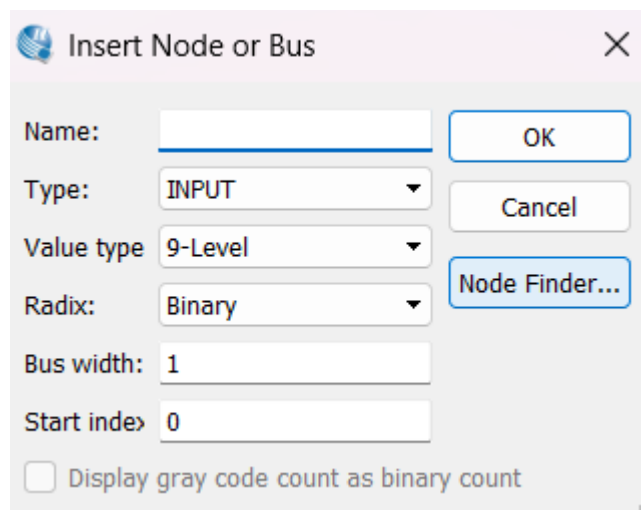
S2. The following window will open up.



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

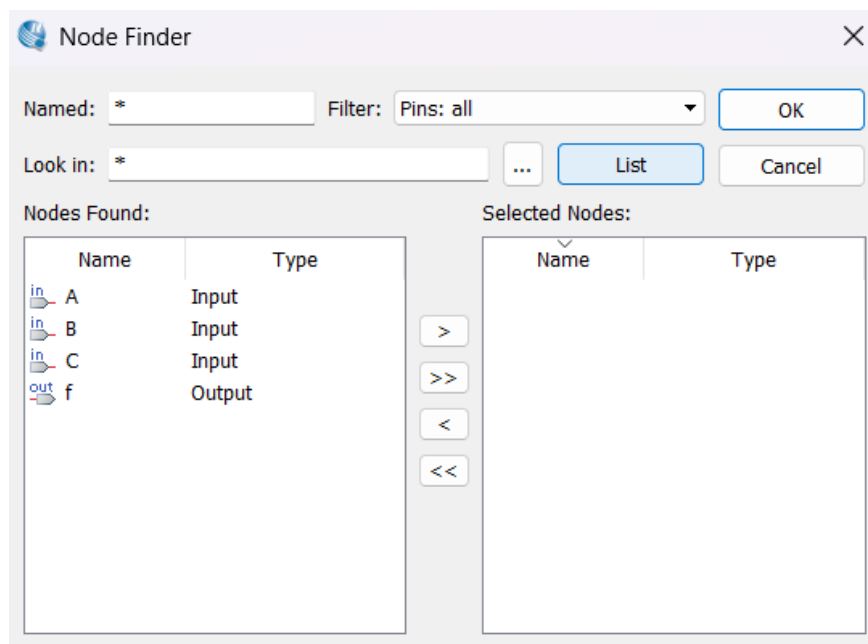


S4. In the **Insert Node or Bus** window, click **Node Finder**.

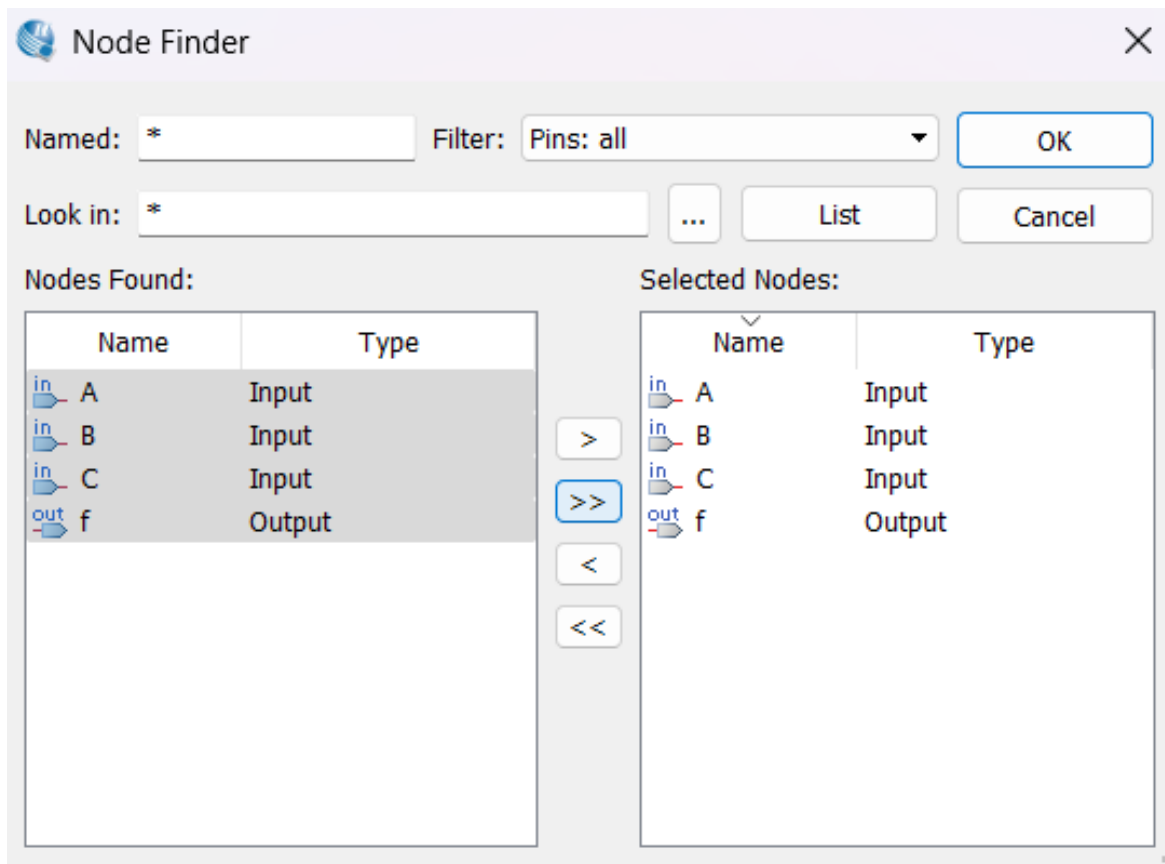


S5. In the **Node Finder** window, Click **List**. Make sure **Pins:all** is selected under **Filter**.

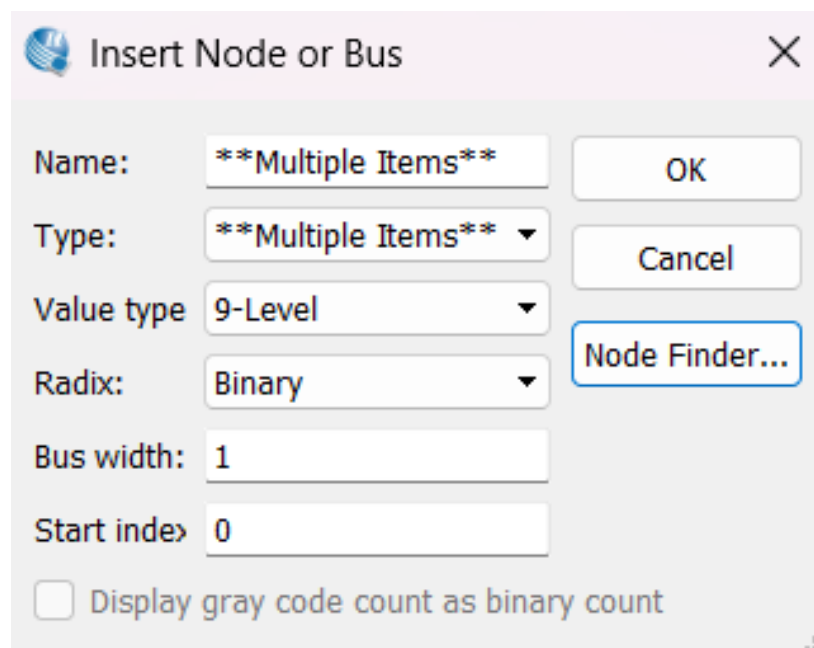
S6. Now the window will look like the following. Click on the '>>' button.



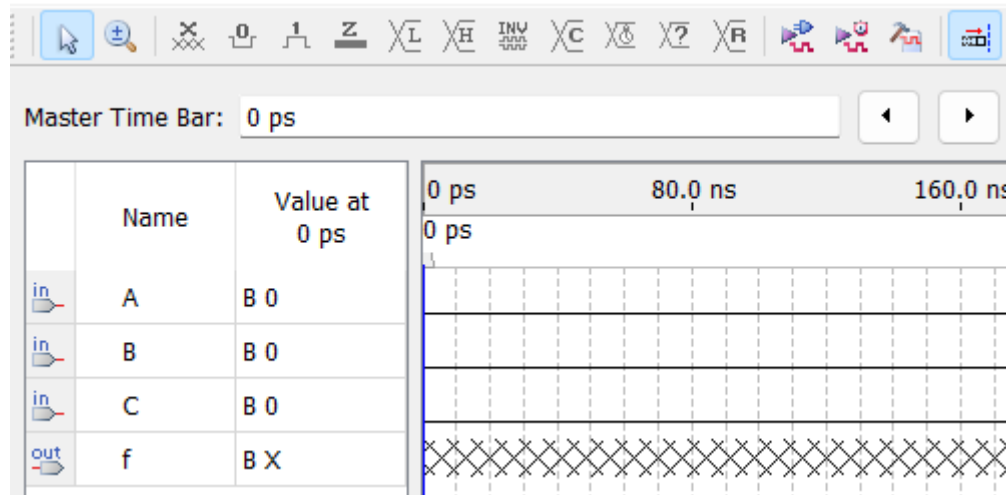
S7. Now, it should appear like the following. Click **OK**.



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

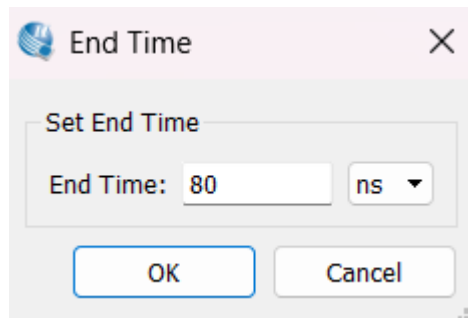


S9. The vector waveform file will now look like the following:

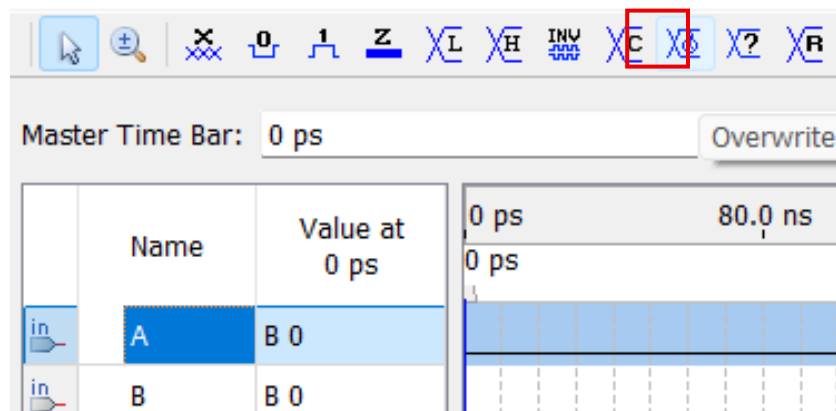


Now, you can clearly see the inputs and outputs.

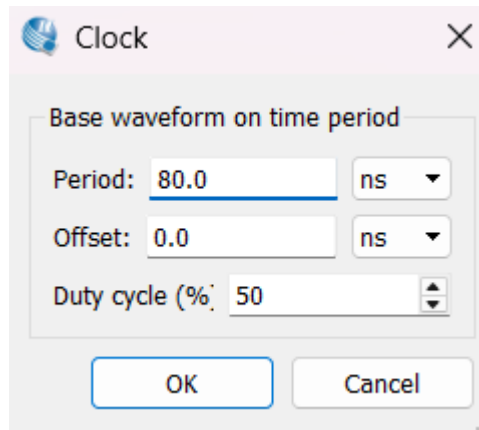
S10. Click on **Edit → Set End Time...** and set the time to **80ns**.



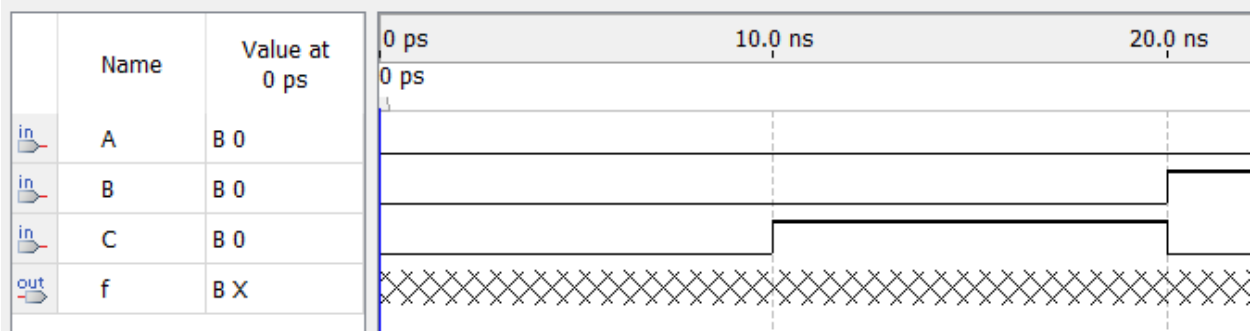
S11. Select an input and from tool bar, click on the **Overwrite Clock** icon.



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:

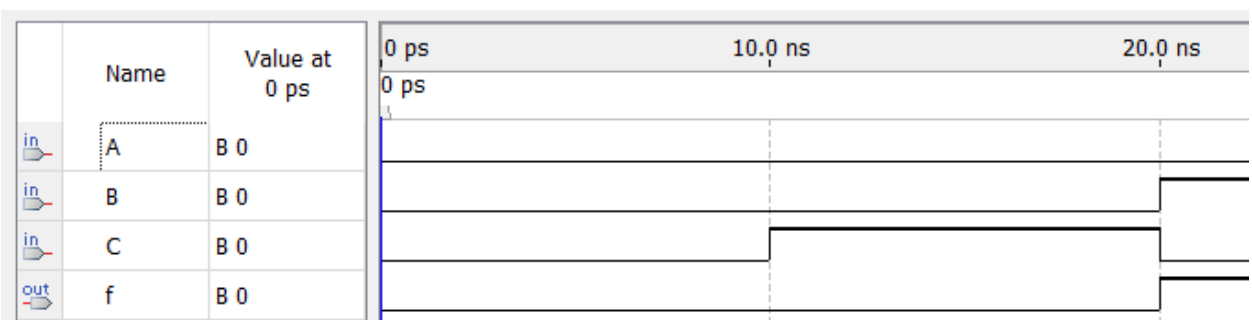


S14. Save the **Vector Waveform File**.

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



S16. Now observe the simulated waveforms..



FPGA Programming:

F1. To load the program on to the FPGA board, go to **Assignments** → **Pins**.

Top View - Wire Bond
Cyclone II - EP2C35F672C6

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength	Differential Pair
A	Input				3.3-V LVTTTL (default)		24mA (default)	
B	Input				3.3-V LVTTTL (default)		24mA (default)	
C	Input				3.3-V LVTTTL (default)		24mA (default)	
f	Output				3.3-V LVTTTL (default)		24mA (default)	
<<new node>>								

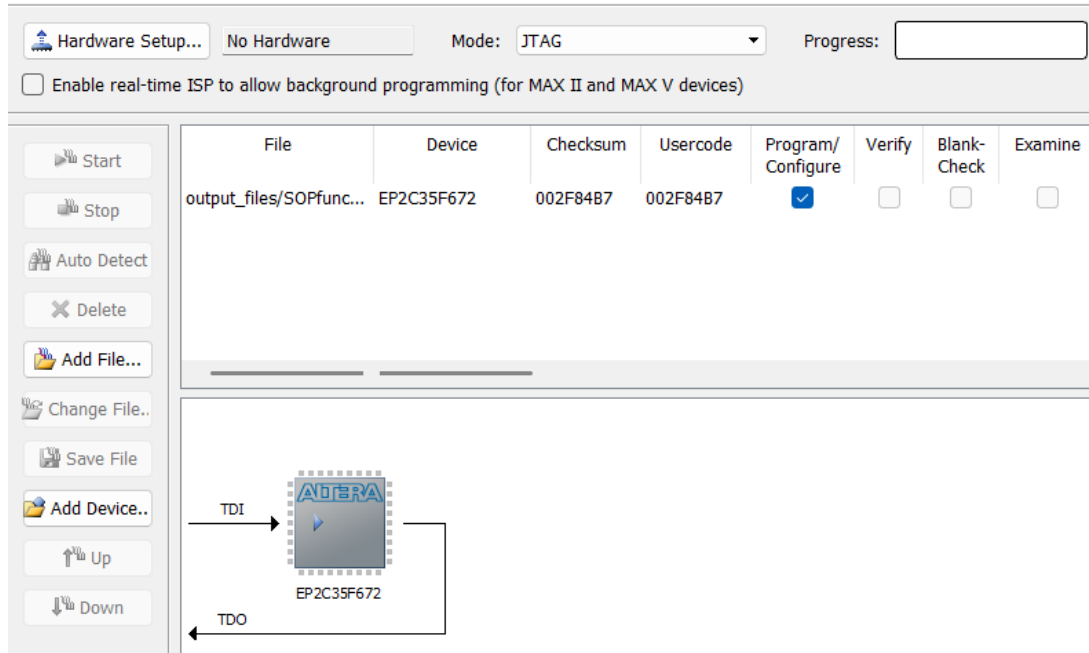
F2. We will be using toggle switches for inputs and Red LED for output. Assign the pins as follows. Click on **Location** and type in the pin name.

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			

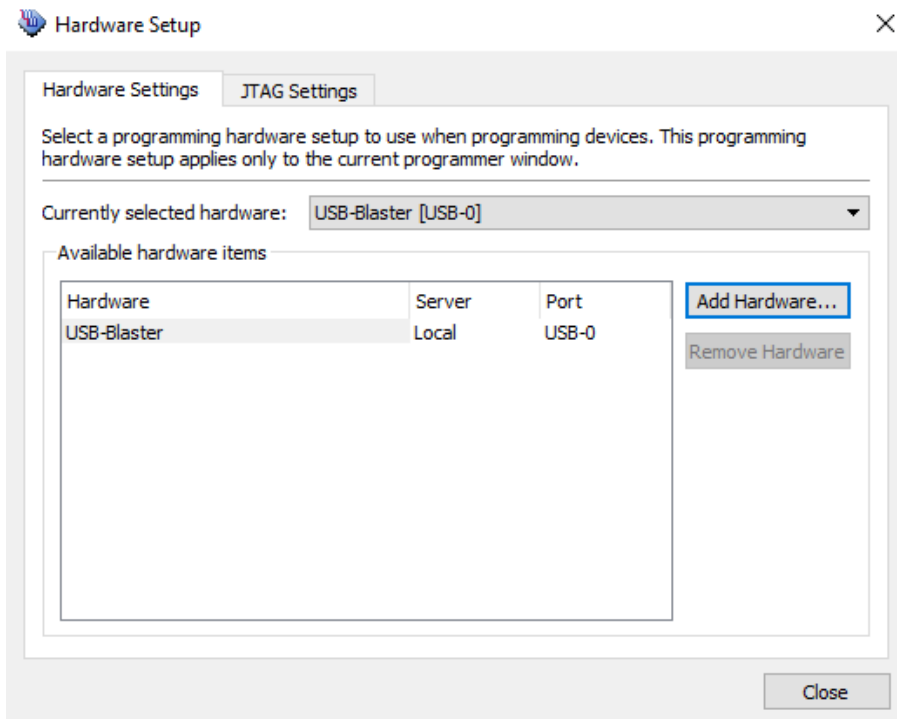
Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength
A	Input	PIN_P25	6	B6_N0	3.3-V LVTTTL (default)		24mA (default)
B	Input	PIN_N26	5	B5_N1	3.3-V LVTTTL (default)		24mA (default)
C	Input	PIN_N25	5	B5_N1	3.3-V LVTTTL (default)		24mA (default)
f	Output	PIN_AE23	7	B7_N0	3.3-V LVTTTL (default)		24mA (default)

F3. Close the window. Note that now you will have to compile the code again. So go **to start compilation**.

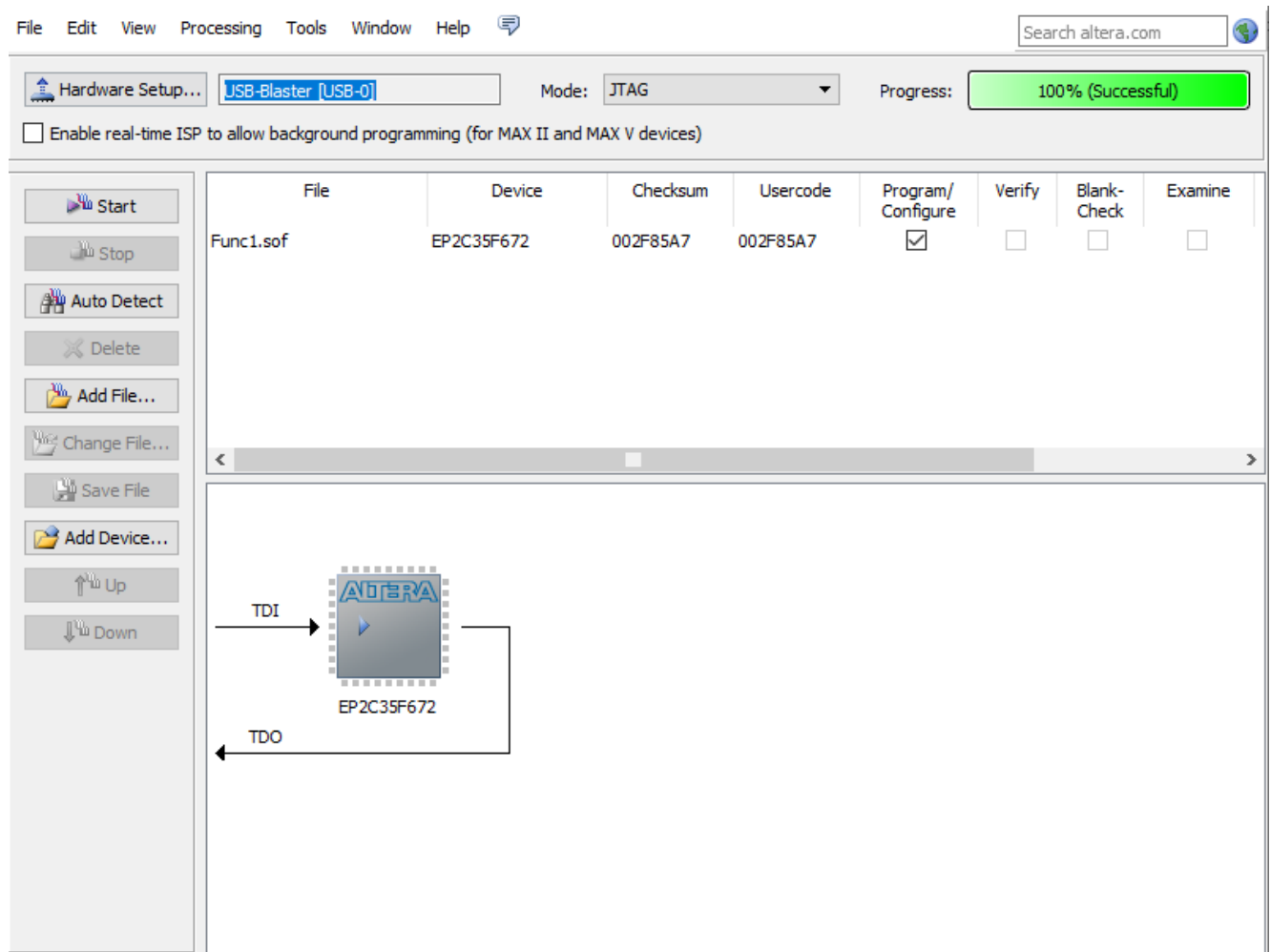
F4. After successful compilation, go to **Tools → Programmer**.



F5. Select **Hardware Setup**. In the **Hardware Setup** window, select **USB-Blaster (USB-0)** and click **Close**.



F6. 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:

Type	Message
	Info: Ended Programmer operation at Sat Mar 11 10:24:10 2017
	Info: Started Programmer operation at Sat Mar 11 10:24:30 2017
	Info: Configuring device index 1
	Info: Device 1 contains JTAG ID code 0x020B40DD
	Error: CONF_DONE pin failed to go high in device 1
	Error: Operation failed
	Info: Ended Programmer operation at Sat Mar 11 10:24:32 2017

F7. Verify the outputs in FPGA board.

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

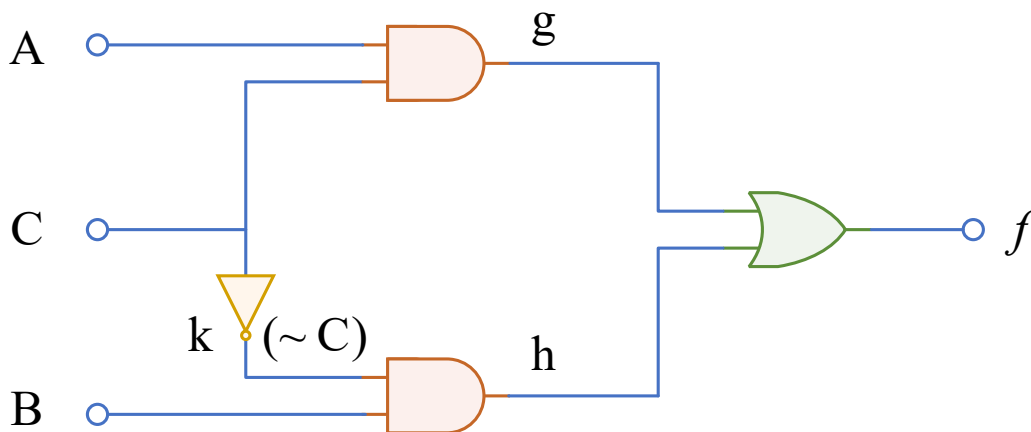
We will focus on Verilog HDL for our Lab. In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation,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


Behavioral

Basic

```
module SOPfunc (A, B, C, f);  
    input A, B, C;  
    output reg f;  
    reg g, h;  
  
    always @ (A, B, C)  
    begin  
        g = A & C;  
        h = B & ~C;  
        f = g | h;  
    end  
  
endmodule
```

Simplified

```
module SOPfunc (A, B, C, f);  
    input A, B, C;  
    output reg f;  
  
    always @ (A, B, C)  
        f = (A & C) | (B & ~C);  
  
endmodule
```

Using Case (Basic)

```

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

    always @ (A, B, C)
        case ({A, B, C})
            3'b000: f = 0;
            3'b001: f = 0;
            3'b010: f = 1;
            3'b011: f = 0;
            3'b100: f = 0;
            3'b101: f = 1;
            3'b110: f = 1;
            3'b111: f = 1;
        endcase

endmodule

```

Using Case (Simplified)

```

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

    always @ (A, B, C)
        case ({A, B, C})
            0: f = 0;
            1: f = 0;
            2: f = 1;
            3: f = 0;
            4: f = 0;
            5: f = 1;
            6: f = 1;
            7: f = 1;
        endcase

endmodule

```

Experiment: 3**Experiment name:** *Introduction to Adder Circuit.***Introduction:**

Adders and sub tractors are the basic operational units of simple digital arithmetic operations. In this experiment, the students will construct the basic adder and sub tractor 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 x and y 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 –

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

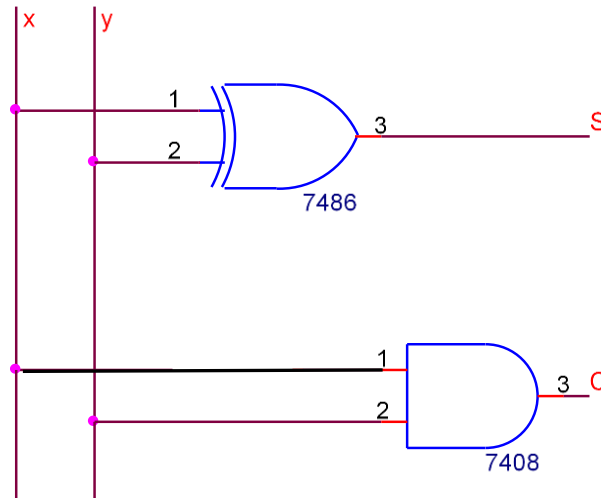
The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are –

$$S = x'y + xy' = x \oplus y$$

$$C = xy$$

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 x and y , represent the two significant bits to be added. The third input, z , represents the carry from the previous lower 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 for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is

x	y	z	C	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

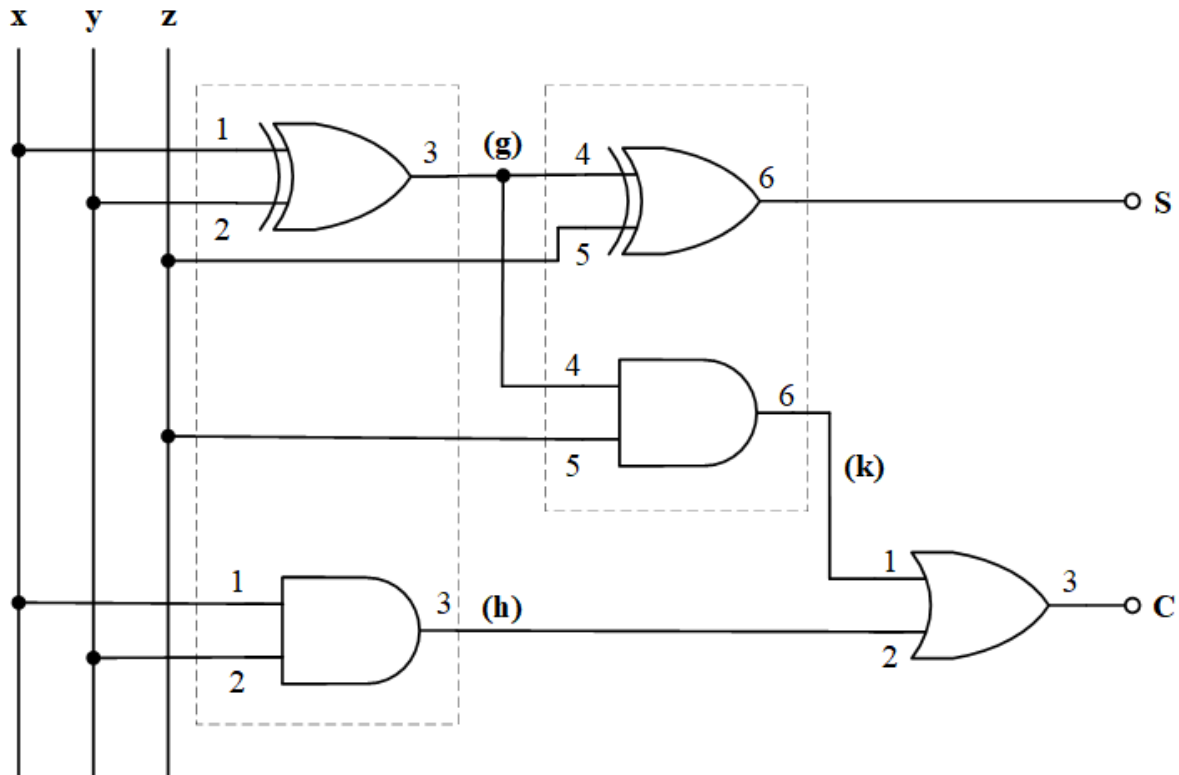
The eight rows under the input variables designate all possible combinations of 1's and 0's that these variables may have. The 1's and 0's for the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0's, the output is 0. The S output is equal to 1 when only 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:

$$S = x'y'z + x'yz' + xy'z' + xyz = z'(x'y + xy') + z(x'y' + xy) = z'(x \oplus y) + z(x \oplus y)' = x \oplus y \oplus z$$

$$C = x'yz + xy'z + xyz' + xyz = (x'y + xy')z + xy(z + z') = (x \oplus y)z + xy$$

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



Equipment:

1. Trainer Board
2. IC 7408, 7432, 7486

Procedure:

1. Fill up the truth table for a half adder.
2. Verify the Boolean function for a half adder.
3. Construct the logic diagram from the Boolean functions.
4. Select the ICs from the equipment list.
5. Implement the output logic.
6. Fill up the truth table for a full adder.
7. Verify the Boolean function for a full adder.
8. Construct the logic diagram from the Boolean functions.
9. Select the ICs from the equipment list.
10. Implement the output logic.

Procedure for FPGA

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

Half Adder:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
x	SW1	PIN_N26	C	LEDR1	PIN_AF23
y	SW0	PIN_N25	S	LEDR0	PIN_AE23

Full Adder:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
x	SW2	PIN_P25	C	LEDR1	PIN_AF23
y	SW1	PIN_N26	S	LEDR0	PIN_AE23
z	SW0	PIN_N25			

Report

1. Find out the expressions of Sum (S) & Carry (C) bit of the Full-Adder using K-Map.
2. Design a Full-Adder using two Half-Adder block and basic gates.
3. Design a 4-bit Adder using four (4) Full-Adder blocks.

Half Adder (Structural)

```
module HalfAdder (x, y, S, C);  
    input x, y;  
    output S, C;  
  
    xor (S, x, y);  
    and (C, x, y);  
endmodule
```

Half Adder (Data Flow)

```
module HalfAdder (x, y, S, C);  
    input x, y;  
    output S, C;  
  
    assign S = x ^ y;  
    assign C = x & y;  
endmodule
```

Half Adder (Behavioral)

```
module HalfAdder (x, y, S, C);  
    input x, y;  
    output reg S, C;  
  
    always @ (x, y)  
        {C, S} = x + y;  
endmodule
```

Full Adder (Structural)

```

module FullAdder (x, y, z, S, C);
    input x, y, z;
    output S, C;
    wire g, h, k;

    // Sum (S)
    xor (g, x, y);
    xor (S, g, z);

    // Carry (C)
    and (h, x, y);
    and (k, g, z);
    or (C, h, k);
endmodule

```

Full Adder (Data Flow)

```

module FullAdder (x, y, z, S, C);
    input x, y, z;
    output S, C;

    assign S = x ^ y ^ z;
    assign C = ((x ^ y) & z) | (x & y);
endmodule

```

Full Adder (Behavioral)

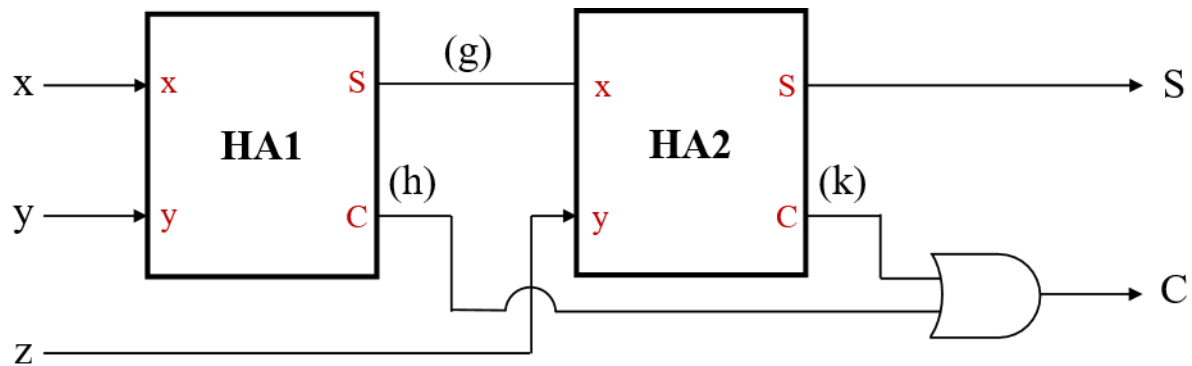
```

module FullAdder (x, y, z, S, C);
    input x, y, z;
    output reg S, C;

    always @ (x, y, z)
        {C, S} = x + y + z;
endmodule

```


Full Adder from Half Adder



// Full Adder (Top Level Hierarchy)

```
module FA (x, y, z, S, C);
```

```
    input x, y, z;
```

```
    output S, C;
```

```
    wire g, h, k;
```

```
    HA HA1 (x, y, g, h);
```

```
    HA HA2 (g, z, S, k);
```

```
    or (C, h, k);
```

```
endmodule
```

// Half Adder

```
module HA (x, y, S, C);
```

```
    input x, y;
```

```
    output reg S, C;
```

```
    always @ (x, y)
```

```
        {C, S} = x + y;
```

```
endmodule
```

Experiment: 4**Experiment name: *Introduction to BCD Adder*****Introduction:**

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:

$$\begin{array}{rcl}
 45 & 0100\ 0101 & \leftarrow \text{BCD for 45} \\
 +33 & +\ 0011\ 0011 & \leftarrow \text{BCD for 33} \\
 \hline
 78 & 0111\ 1000 & \leftarrow \text{BCD for 78}
 \end{array}$$

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:

$$\begin{array}{rcl}
 6 & 0110 & \leftarrow \text{BCD for 6} \\
 +7 & + 0111 & \leftarrow \text{BCD for 7} \\
 \hline
 +13 & 1101 & \leftarrow \text{invalid code group for BCD}
 \end{array}$$

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:

$$\begin{array}{rcl}
 & 0110 & \leftarrow \text{BCD for 6} \\
 + & 0111 & \leftarrow \text{BCD for 7} \\
 \hline
 & 1101 & \leftarrow \text{invalid sum} \\
 + & 0110 & \leftarrow \text{add 6 for correction} \\
 \hline
 \underbrace{0001}_1 & \underbrace{0011}_3 & \leftarrow \text{BCD for 13}
 \end{array}$$

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:

$$\begin{array}{rcl}
 & & \downarrow 1 \\
 59 & 0101 & 1001 \leftarrow \text{BCD for 59} \\
 +38 & + 0011 & 1000 \leftarrow \text{BCD for 38} \\
 \hline
 97 & 1001 & 0001 \leftarrow \text{perform addition} \\
 & & 0110 \leftarrow \text{add 6 to correct} \\
 \hline
 & \underbrace{1001}_9 & \underbrace{0111}_7 \quad \text{BCD for 97}
 \end{array}$$

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 K , Z_4 , Z_3 , Z_2 , and Z_1 . K 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 $K = 1$. 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

$$C = K + Z_4Z_3 + Z_4Z_2$$

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.

Caution:

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

Procedure:

1. Draw the logic diagram to implement the task.
2. Select the required ICs.
3. Verify the following truth table for 20 output values (0-20).

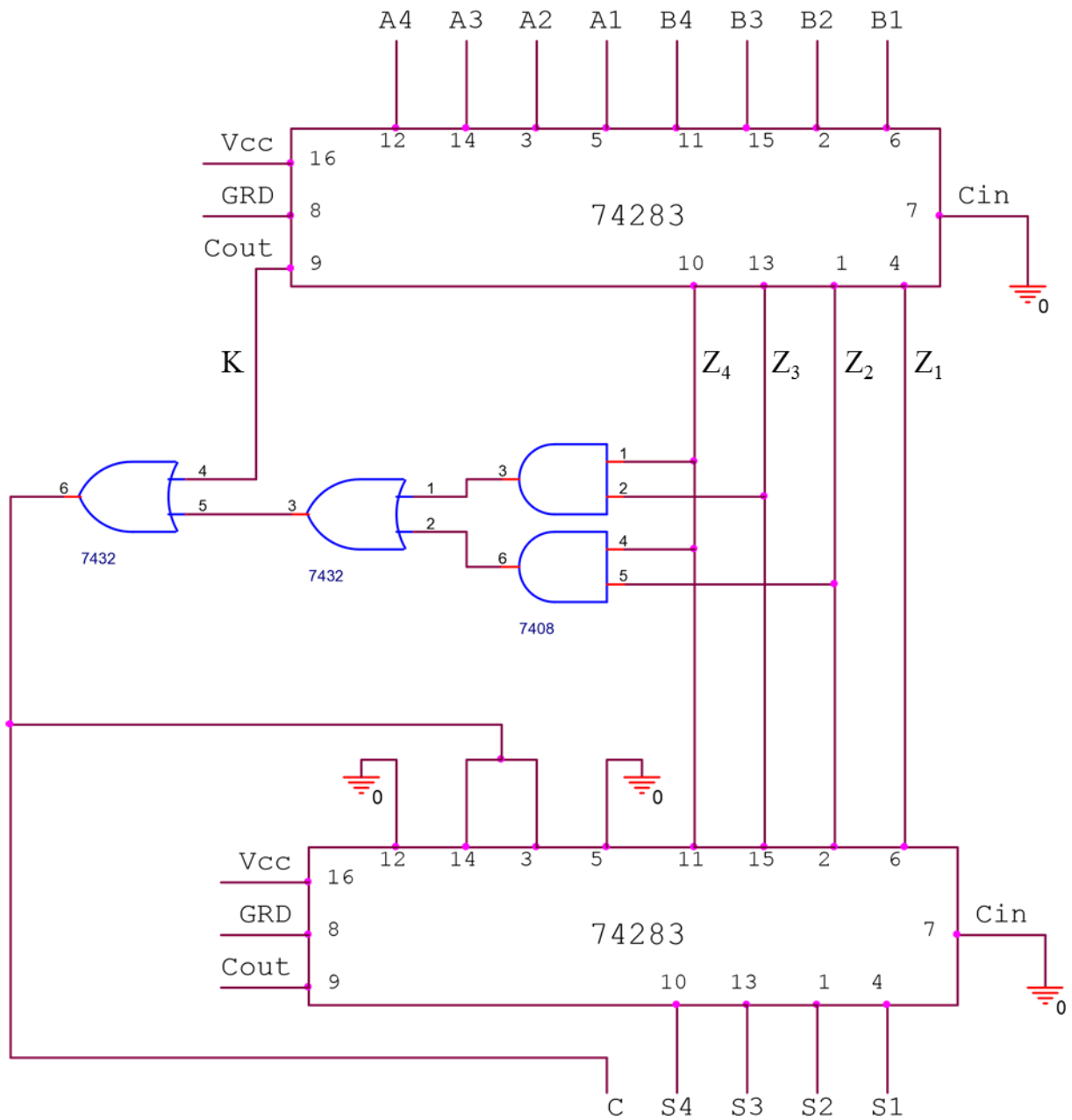


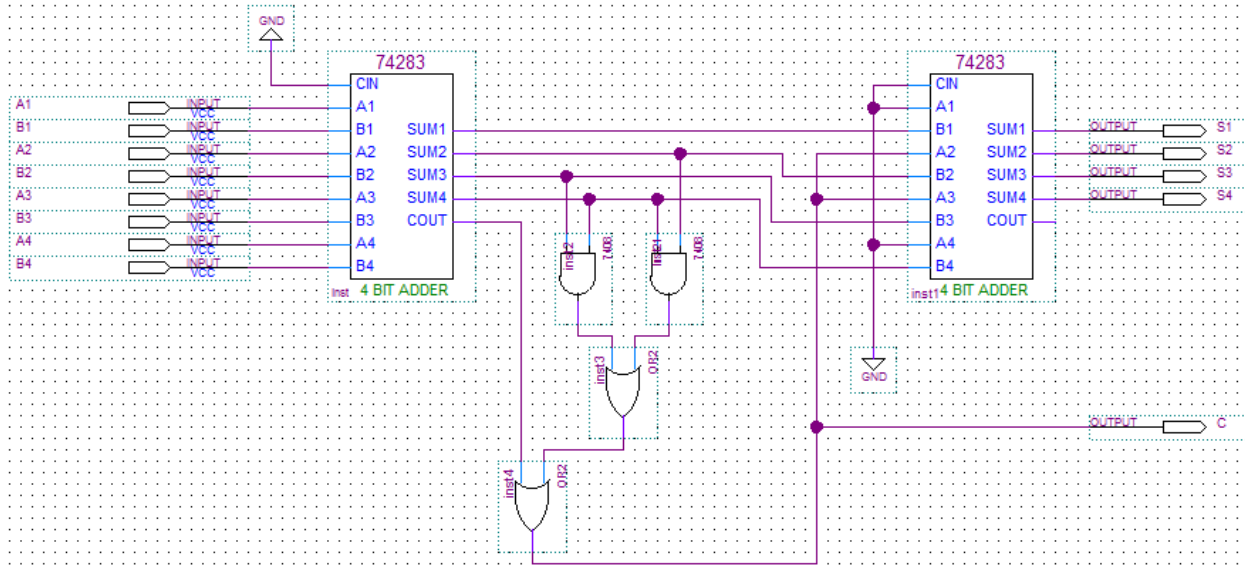
Figure: BCD Adder

<i>Decimal</i>	<i>Binary Sum</i>					<i>BCD Sum</i>				
	<i>K</i>	<i>Z₄</i>	<i>Z₃</i>	<i>Z₂</i>	<i>Z₁</i>	<i>C</i>	<i>S₄</i>	<i>S₃</i>	<i>S₂</i>	<i>S₁</i>
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. If everything is ok, assign pins as follows:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
A1	SW4	PIN_AF14	C	LEDR4	PIN_AD22
A2	SW5	PIN_AD13	S1	LEDR0	PIN_AE23
A3	SW6	PIN_AC13	S2	LEDR1	PIN_AF23
A4	SW7	PIN_C13	S3	LEDR2	PIN_AB21
B1	SW0	PIN_N25	S4	LEDR3	PIN_AC22
B2	SW1	PIN_N26			
B3	SW2	PIN_P25			
B4	SW3	PIN_AE14			



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

<i>A</i> (Decimal)	A (Binary)				<i>B</i> (Decimal)	B (Binary)				<i>BCD</i> <i>SUM</i> (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					
7	0	1	1	1	4	0	1	0	0	1 1					
2	0	0	1	0	6	0	1	1	0	0 8					
9	1	0	0	1	8	1	0	0	0	1 7					
1	0	0	0	1	3	0	0	1	1	0 4					

BCD Adder

```
module BcdAdd (A, B, S, C);  
    input [4:1] A, B;  
    output reg [4:1] S;  
    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
```


Experiment: 5

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.

A 4 to 1 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.

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.

Caution:

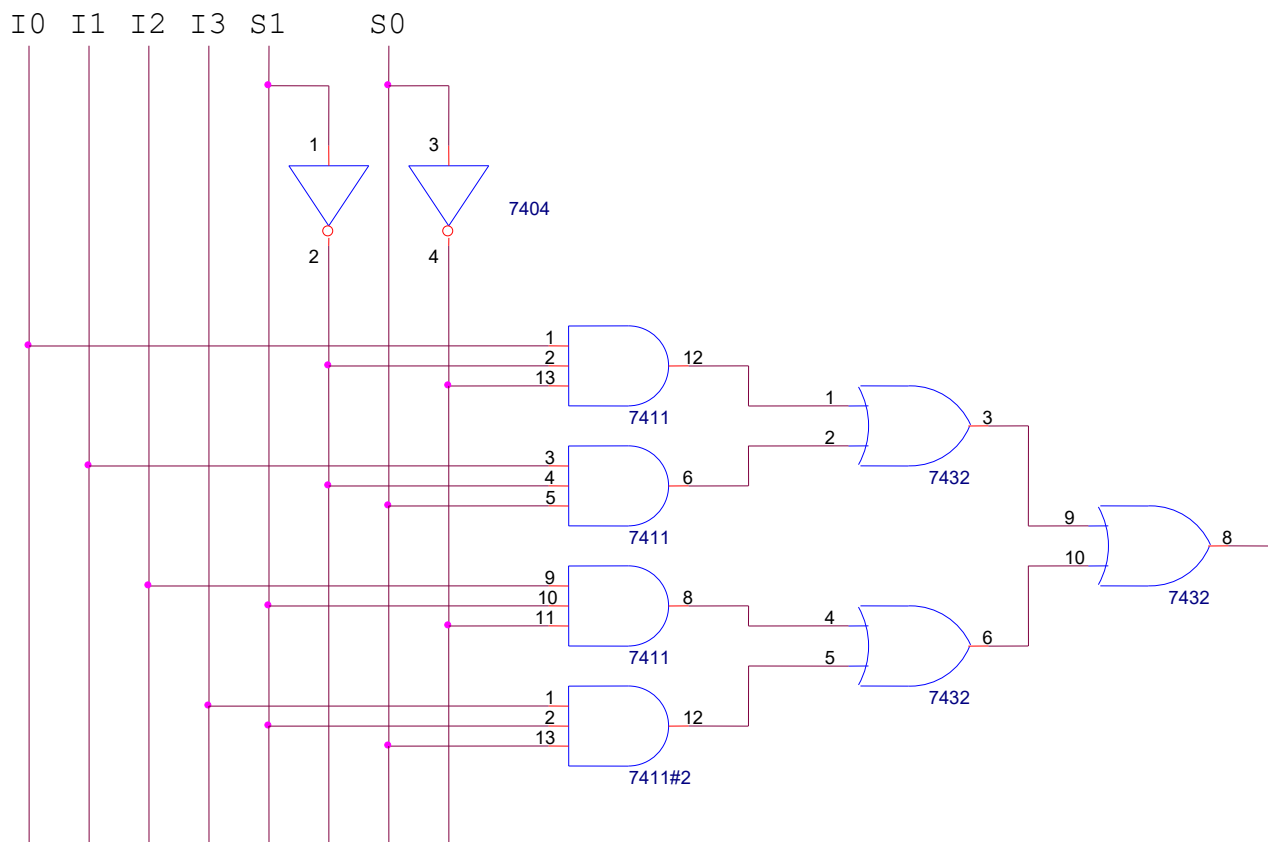
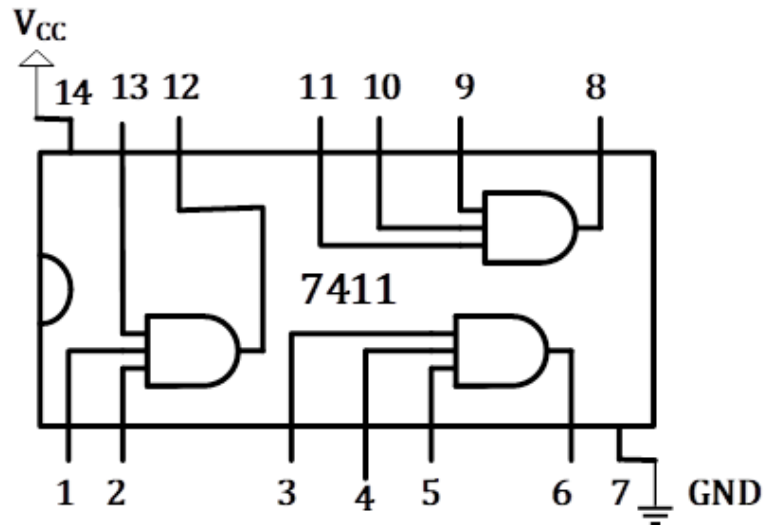
1. Remember to properly identify the pin numbers so that no accidents occur.
2. Properly bias the ICs with appropriate voltages to appropriate pins.

Equipment:

1. Trainer Board
2. IC 74151, 7432, 7408, 7404
3. Microprocessor Data handbook.

Job 1:

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



Procedure:

1. Write the truth table for four to one way MUX.

S_1	S_0	Y

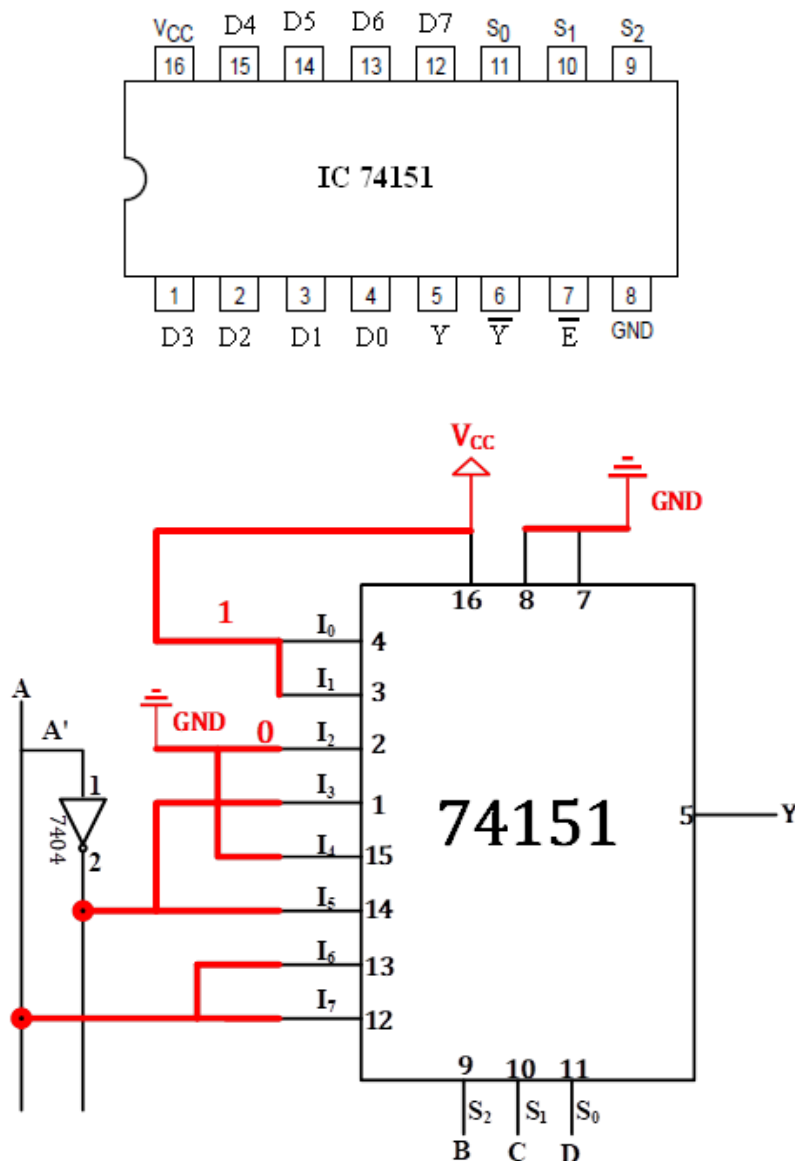
2. Write the Boolean function for the output logic.
3. Draw the logic diagram to implement the Boolean function.
4. Select ICs from the equipment list.
5. Observe and note the output logic for all combinations of inputs.

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 to 1 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.

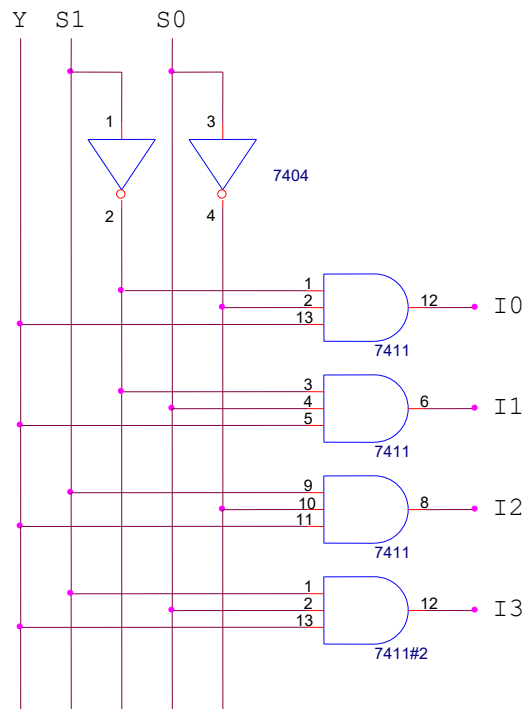
Job 3:

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

Procedure:

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

S_1	S_0	I_0	I_1	I_2	I_3



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.

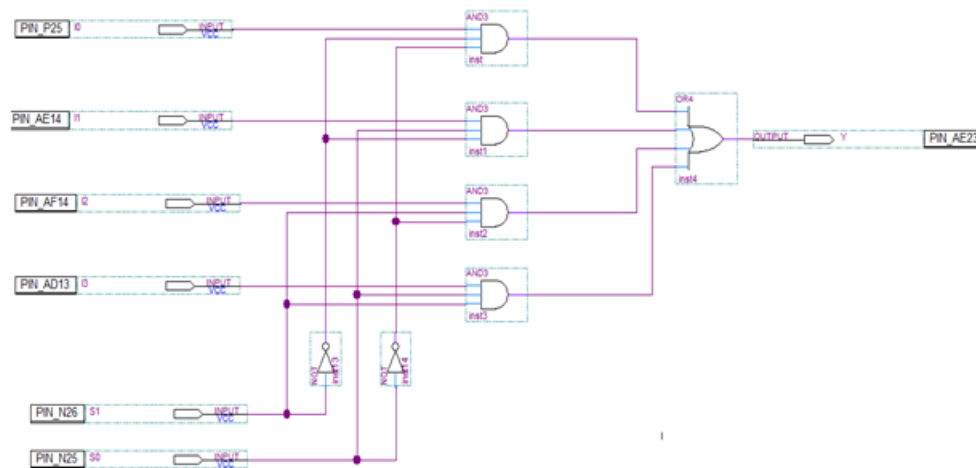
Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch	Pin No.	Signal	LED	Pin No.
I0	SW2	PIN_P25	Y	LED0	PIN_AE23
I1	SW3	PIN_AE14			
I2	SW4	PIN_AF14			
I3	SW5	PIN_AD13			
S0	SW0	PIN_N25			
S1	SW1	PIN_N26			

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.

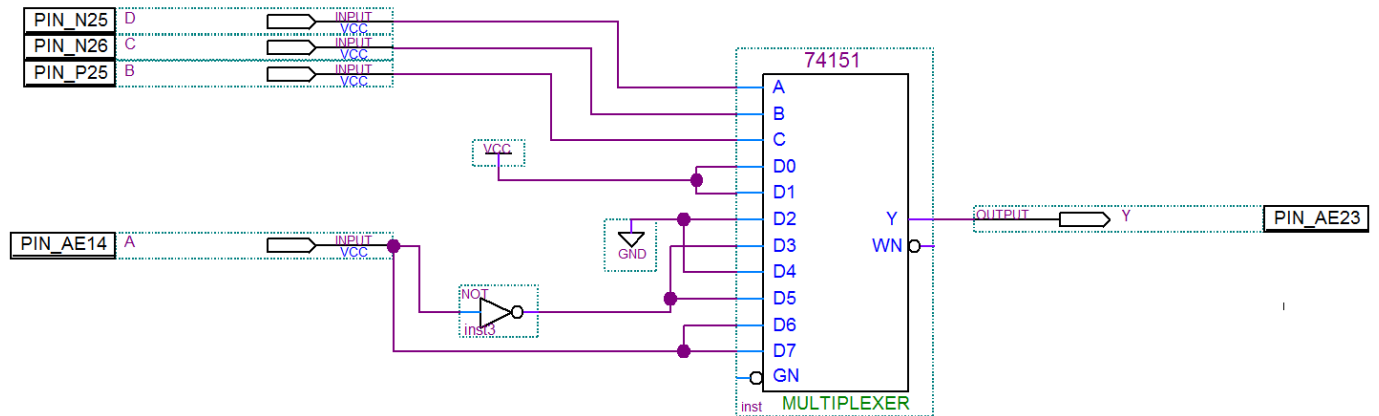
Implement the following function using an 8:1 MUX

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

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:

INPUT			OUTPUT		
Signal	Switch	Pin No.	Signal	LED	Pin No.
A	SW3	PIN_AE14	Y	LED0	PIN_AE23
B	SW2	PIN_P25			
C	SW1	PIN_N26			
D	SW0	PIN_N25			

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



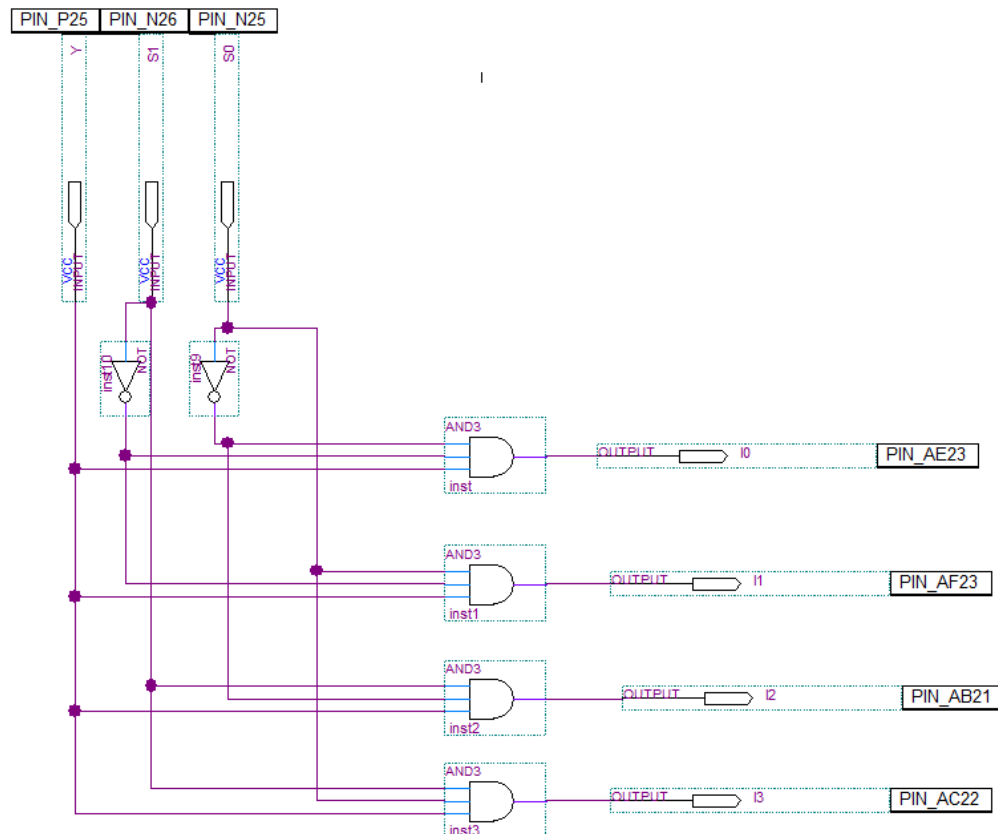
4. Test the functionality of the designed circuit using switches and LEDs on the FPGA board.

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
S0	SW0	PIN_N25	I0	LEDR0	PIN_AE23
S1	SW1	PIN_N26	I1	LEDR1	PIN_AF23
Y	SW2	PIN_P25	I2	LEDR2	PIN_AB21
			I3	LEDR3	PIN_AC22

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.

Report:

1. Implement a Full Adder using an 8:1 MUX.
2. Repeat 1 using two 4:1 MUX and basic gates.
3. Implement a 4:1 MUX using three 2:1 MUX.
4. Implement an 8:1 MUX using two 4:1 MUX & a 2:1 MUX.

MUX (2 to 1)

Conditional (Ternary) Operator

```
module mux2to1 (I0, I1, S, Y);  
    input I0, I1, S;  
    output reg Y;  
  
    always @ (I0, I1, S)  
        Y = S ? I1:I0;  
  
endmodule
```

If-Else

```
module mux2to1 (I0, I1, S, Y);  
    input I0, I1, S;  
    output reg Y;  
  
    always @ (I0, I1, S)  
        if (S == 0)  
            Y = I0;  
        else  
            Y = I1;  
  
endmodule
```

Case

```
module mux2to1 (I0, I1, S, Y);  
    input I0, I1, S;  
    output reg Y;  
  
    always @ (I0, I1, S)  
        case (S)  
            0: Y = I0;  
            1: Y = I1;  
        endcase  
  
endmodule
```

MUX (4 to 1)

If-Else If-Else

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

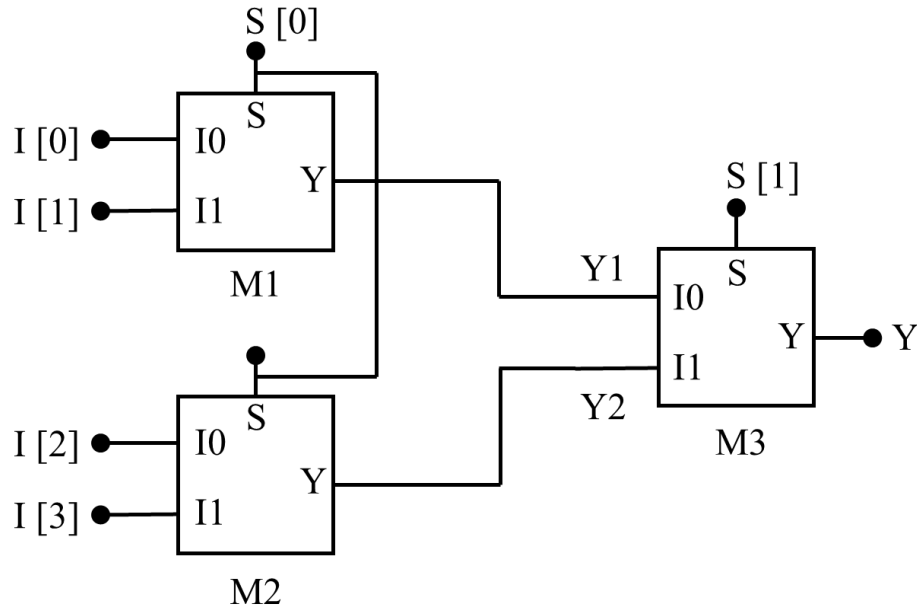
    always @ (I, S)
        if (S == 0)
            Y = I [0];
        else if (S == 1)
            Y = I [1];
        else if (S == 2)
            Y = I [2];
        else
            Y = I [3];
endmodule
```

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
```

4 to 1 MUX from 2 to 1 MUX



// 4 to 1 Mux (Top Level Hierarchy)

```

module mux4to1 (I, S, Y);
    input [3:0] I;
    input [1:0] S;
    output Y;
    wire Y1, Y2;

    mux2to1 M1(I [0], I [1], S [0], Y1);
    mux2to1 M2(I [2], I [3], S [0], Y2);
    mux2to1 M3(Y1, Y2, S [1], Y);

endmodule
  
```

// 2 to 1 Mux

```

module mux2to1 (I0, I1, S, Y);
    input I0, I1, S;
    output reg Y;

    always @ (I0, I1, S)
        Y = S ? I1 : I0;

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'b0000  
        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
```

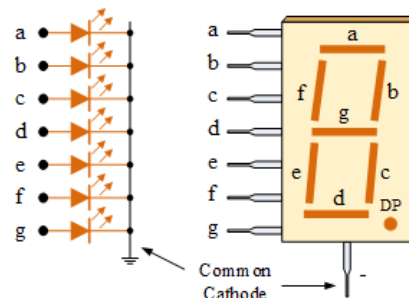
Experiment: 6**Experiment name:** *Introduction to Encoders and Decoders.***Introduction:****Priority Encoder**

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 D₃ 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). D₂ has the next priority level. The output is 10 if D₂ = 1 provided that D₃ = 0, regardless of the values of the other two lower-priority inputs. The output for D₁ 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
0	0	0	0		X	X
0	0	0	1		0	0
0	0	1	X		0	1
0	1	X	X		1	0
1	X	X	X		1	1

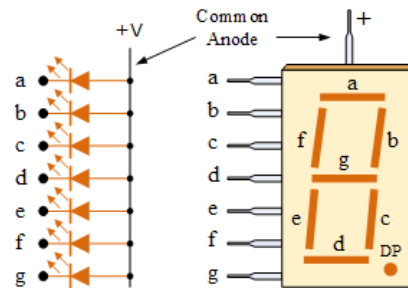
Seven Segment Decoder

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. The Common Cathode (CC) – In the common cathode display, all the cathode connections of the LED segments are joined together to logic “0” or ground. The individual segments are illuminated by application of a “HIGH”, or logic “1” signal via a current limiting resistor to forward bias the individual Anode terminals (a-g).

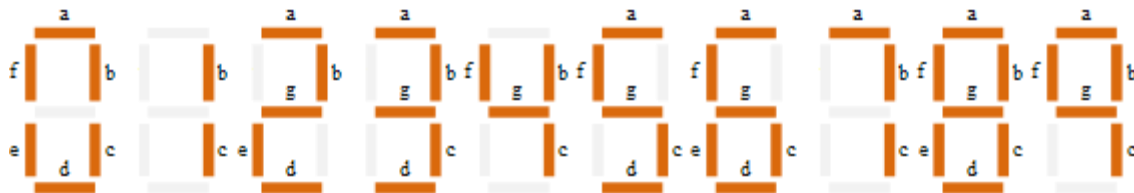
Common Cathode 7-segment Display

2. The Common Anode (CA) – 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).

Common Anode 7-segment Display



7-Segment Display Segments for all Numbers.



Then for a 7-segment display, we can produce a truth table giving the individual segments

Caution:

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

Equipment:

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

Priority encoder:

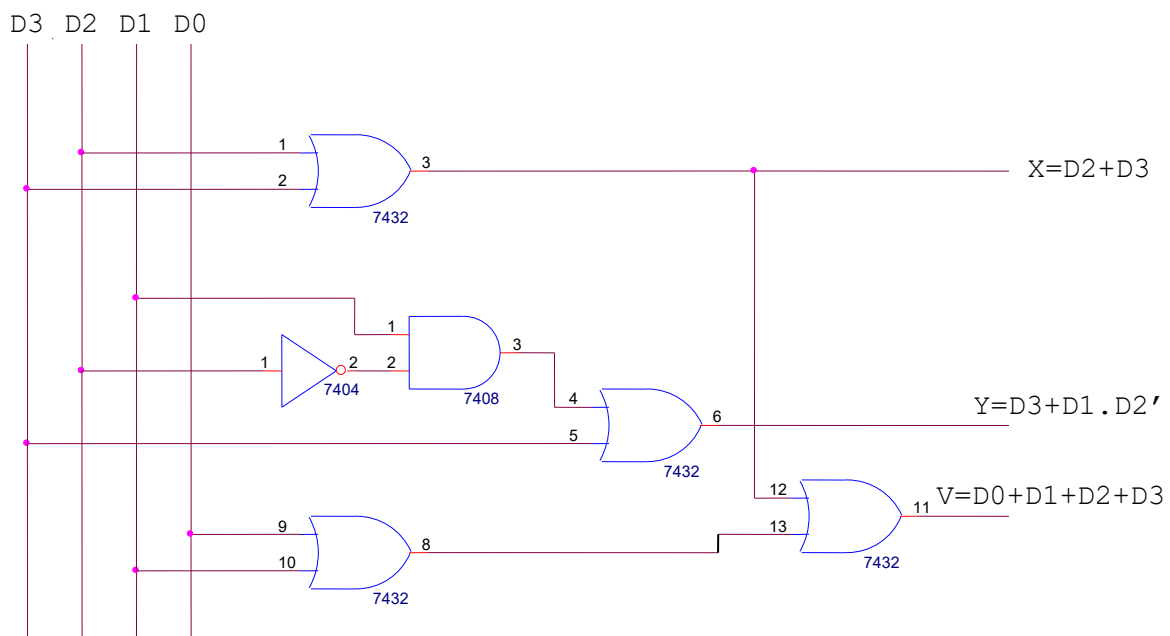
Implement a 4×2 priority encoder with basic gates.

Procedure:

1. Write the truth table for 4×2 priority encoder.

[illegible]

2. Write the Boolean function for the output logic.
3. Simplify the Boolean function using K-map.
4. Draw the logic diagram to implement the simplified Boolean function.



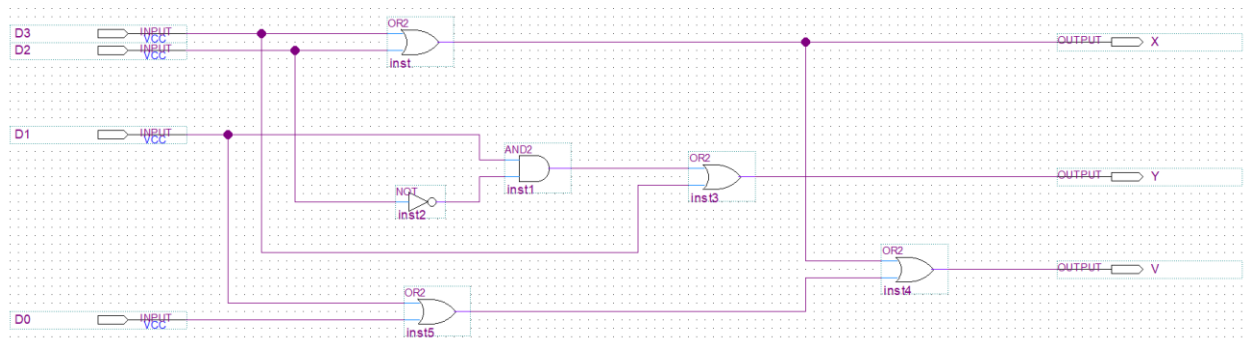
Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch	Pin No.	Signal	LED	Pin No.
D0	SW0	PIN_N25	V	LEDR0	PIN_AE23
D1	SW1	PIN_N26	Y	LEDR1	PIN_AF23
D2	SW2	PIN_P25	X	LEDR2	PIN_AB21
D3	SW3	PIN_AE14			

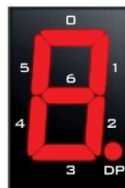
3. After assigning pins, the final schematic should look like the following one:



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

Seven Segment Display

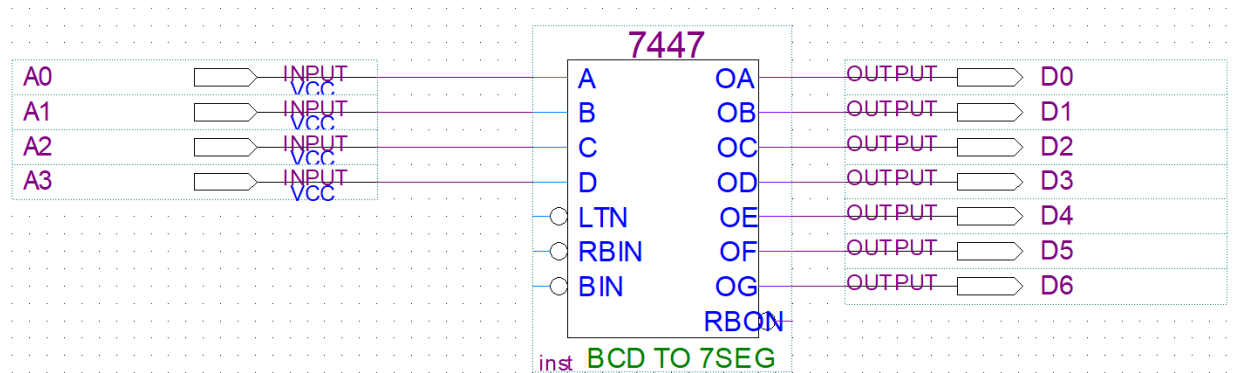
The DE2 Board has eight 7-segment displays. These displays are arranged into two pairs and a group of four, with the intent of displaying numbers of various sizes the seven segments are connected to pins on the Cyclone II FPGA. Applying a low logic level to a segment causes it to light up, and applying a high logic level turns it off. Each segment in a display is identified by an index from 0 to 6, Note that the dot in each display is unconnected and cannot be used.



Procedure for FPGA:

BCD Decoder

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
A0	SW0	PIN_N25	D0	HEX0[0]	PIN_AF10
A1	SW1	PIN_N26	D1	HEX0[1]	PIN_AB12
A2	SW2	PIN_P25	D2	HEX0[2]	PIN_AC12
A3	SW3	PIN_AE14	D3	HEX0[3]	PIN_AD11
			D4	HEX0[4]	PIN_AE11
			D5	HEX0[5]	PIN_V14
			D6	HEX0[6]	PIN_V13

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.

Priority Encoder

```

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

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

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

    output V; // Validity Indicator (NOT reg)


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


    always @ (D)

        case (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'bx

        endcase

    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;  
            2: D = 7'b0010010;  
            3: D = 7'b0000110;  
            4: D = 7'b1001100;  
            5: D = 7'b0100100;  
            6: D = 7'b1100000;  
            7: D = 7'b0001111;  
            8: D = 7'b0000000;  
            9: D = 7'b0001100;  
  
            default: D = 7'bx;  
  
        endcase  
  
endmodule
```

Experiment: 7**Experiment name:** *Introduction to Sequential Logic Circuits.***Caution:**

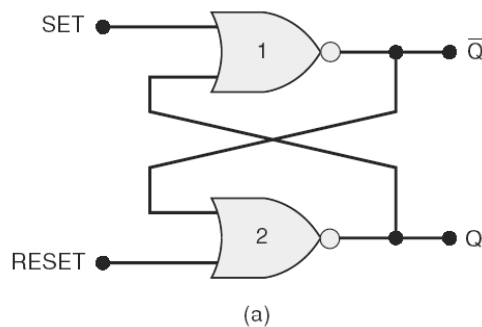
1. Remember to properly identify the pin numbers so that no accidents occur.
2. Properly bias the ICs with appropriate voltages to appropriate pins.

Equipment:

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

Nor Gate Latch

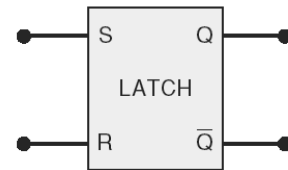
Two cross-coupled NOR gates can be used as a **NOR gate latch**. The arrangement is similar to the NAND latch except that the Q and Q' outputs have reversed positions.



Set	Reset	Output
0	0	No change
1	0	$Q = 1$
0	1	$Q = 0$
1	1	Invalid*

*Produces $Q = \bar{Q} = 0$.

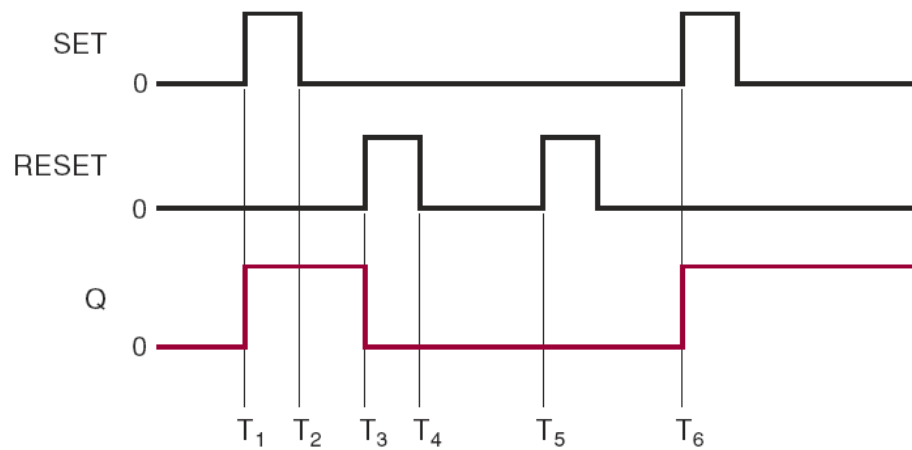
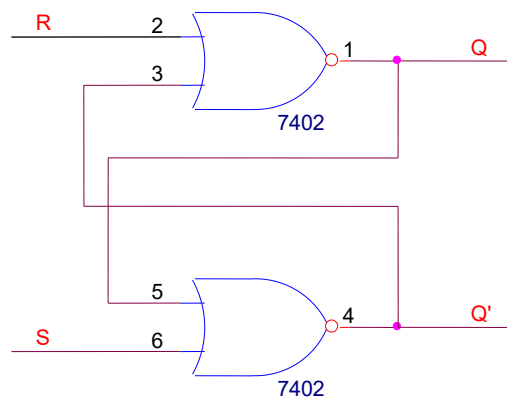
(b)



The analysis of the operation of the NOR latch can be performed in the same manner as for the NAND latch. The results are given in the function table and are summarized as follows:

1. SET = RESET = 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.
2. SET = 1, RESET = 0. This will always set $Q = 1$, where it will remain even after SET returns to 0.
3. RESET = 1, SET = 0, this will always clear $Q = 0$, where it will remain even after RESET returns to 0.
4. SET = 1, RESET = 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.

The NOR gate latch operates exactly like the NAND latch except that the SET and RESET inputs are active-HIGH rather than active-LOW, and the normal resting state is Q will be set HIGH by a HIGH pulse on the SET input, and it will be cleared LOW by a HIGH pulse on the RESET input.

Timing Diagram**Logic Diagram****Procedure:**

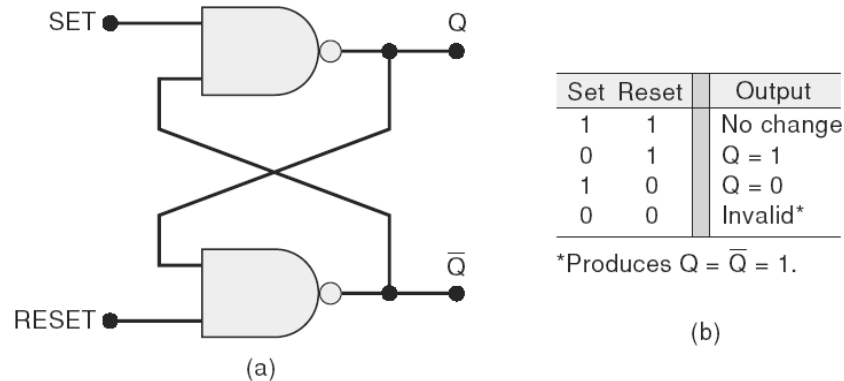
1. Draw the logic diagram to implement SR Latch.
2. Fill up the table with different combinations of inputs.

S	R	Q	Q'
1	0		
0	0		
0	1		
0	0		
1	1		

3. Observe the combination for which no change and invalid or race conditions arise.

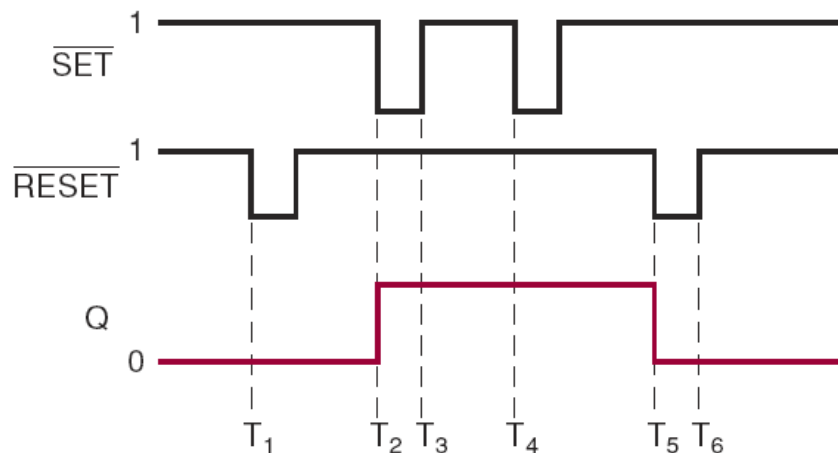
NAND Gate Latch

The most basic FF circuit can be constructed from either two NAND gates or two NOR gates. The NAND gate version, called a **NAND gate latch** or simply a **latch**, is shown in following Figure. The two NAND gates are cross-coupled so that the output of NAND-1 is connected to one of the inputs of NAND-2, and vice versa. The gate outputs, labeled Q and Q' , respectively, are the latch outputs.

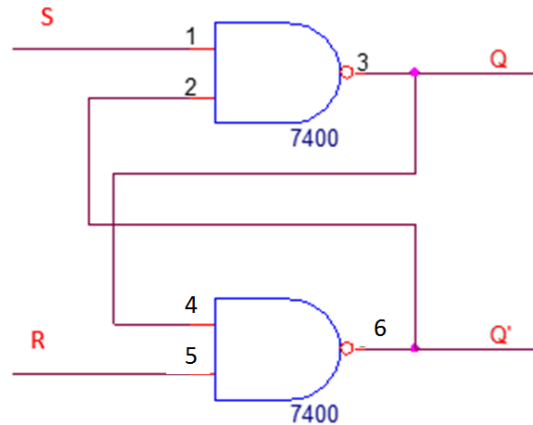


1. SET = RESET = 1. This condition is the normal resting state, and it has no effect on the output state. The Q and Q' outputs will remain in whatever state they were in prior to this input condition
2. SET = 0, RESET = 1. This will always set $Q = 1$, where it will remain even after RESET returns to 0.
3. SET = 1, RESET = 0, this will always clear $Q = 0$, where the output will remain even after RESET returns HIGH. This is called clearing or resetting the latch.
4. SET = 0, RESET = 0, this condition tries to set and reset the latch at the same time, and it produces $Q = Q' = 1$. If the inputs are returned to 1 simultaneously, the resulting output state is unpredictable. This input condition should not be used.

Timing Diagram



Logic Diagram



Procedure:

1. Draw the logic diagram to implement SR Latch.
2. Fill up the table with different combinations of inputs.

S	R	Q	Q'
0	1		
1	1		
1	0		
1	1		
0	0		

3. Observe the combination for which no change and invalid or race conditions arise.

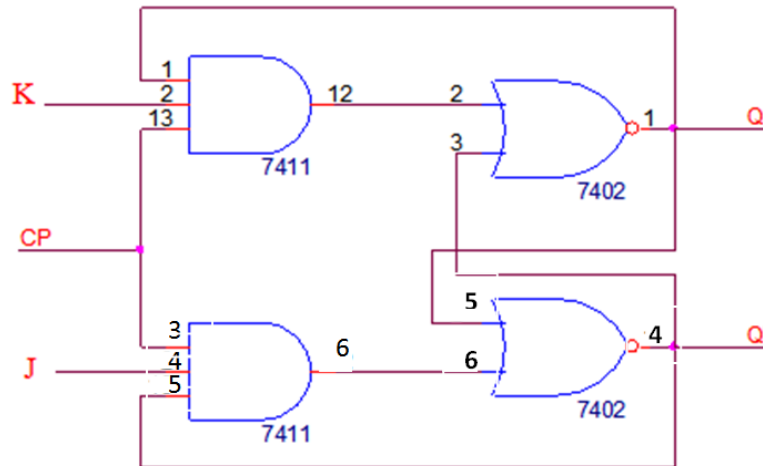
JK Latch

A JK Latch is a refinement of the RS Latch in that the indeterminate state of the RS type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the Latch, respectively. The input marked J is for set and the input marked K is for reset. When both inputs J and K are equal to 1, the Latch switches to its complement state, that is, if $Q = 1$, it switches to $Q = 0$, and vice versa.

A JK Latch constructed with two cross-coupled NOR gates and two AND gates is shown in Figure. Output Q is ANDed with K and CP inputs so that the Latch is cleared during a clock pulse only if Q was previously 1. Similarly, output Q' is ANDed with J and CP inputs so that the flop-flop is set with a clock pulse only when Q' was previously 1. When both J and K are 1, the input pulse is transmitted through one AND gate only: the one whose input is connected to the Latch output that is presently equal to 1. Thus, if $Q = 1$, the output of the upper AND gate becomes 1 upon application of the clock pulse, and the Latch is cleared. If $Q' = 1$, the output of the lower AND gate becomes 1 and the Latch is set. In either case, the output state of the Latch is complemented. The behavior of the JK Latch is demonstrated in the characteristic table.

It is very important to realize that because of the feedback connection in the JK Latch, a CP pulse that remains in the 1 state while both J and K are equal to 1 will cause the output to complement again and repeat complementing until the pulse goes back to 0. To avoid this

undesirable operation, the clock pulse must have a time duration that is shorter than the propagation delay time of the Latch. This is a restrictive requirement, since the operation of the circuit depends on the width of the pulse. For this reason, JK Latches are never constructed as shown in Figure. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction, as discussed in the next section. The same reasoning applies to the T Latch.



Procedure:

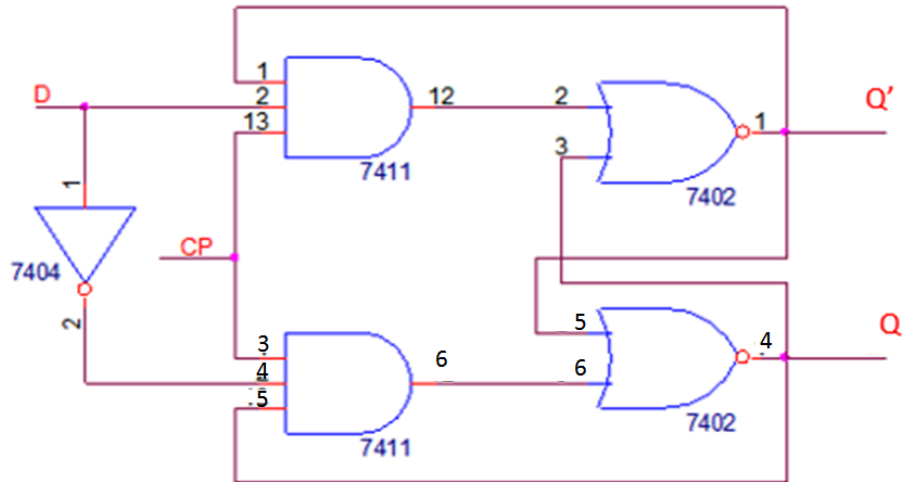
1. Draw the logic diagram to implement J-K Latch.
2. Fill up the table with different combinations of inputs.

Q	J	K	$Q(t+1)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

3. Observe the combination for which no change and invalid or race conditions arise.

D Latch

Design of a D Latch from a J-K Latch.



Procedure:

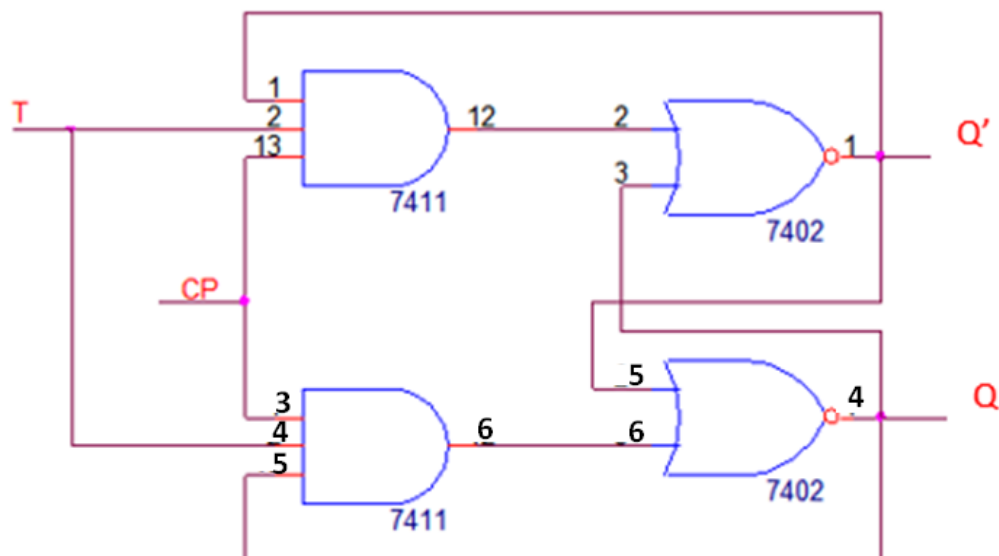
1. Draw the logic diagram to implement D Latch.
2. Fill up the table with different combinations of inputs.

Q	D	$Q(t+1)$
0	0	
0	1	
1	0	
1	1	

3. Observe the combination for which no change and invalid or race conditions arise.

T Latch

Design of a T Latch from a J-K Latch.



Procedure:

1. Draw the logic diagram to implement T Latch.
2. Fill up the table with different combinations of inputs.

Q	T	$Q(t+1)$
0	0	
0	1	
1	0	
1	1	

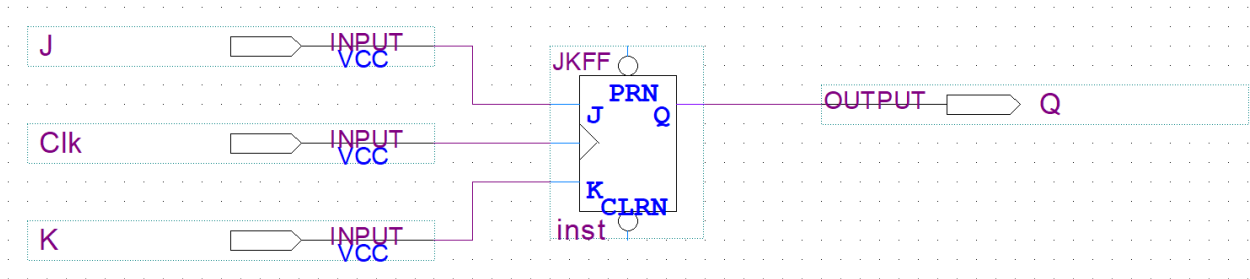
3. Observe the output logic.

Procedure for FPGA:

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.

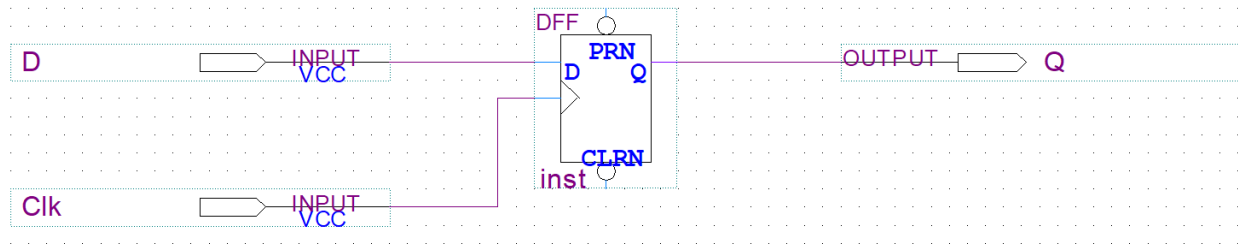
INPUT			OUTPUT		
Signal	Switch No	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	Q	LEDR0	PIN_AE23
J	SW1	PIN_N26			
K	SW0	PIN_N25			

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.

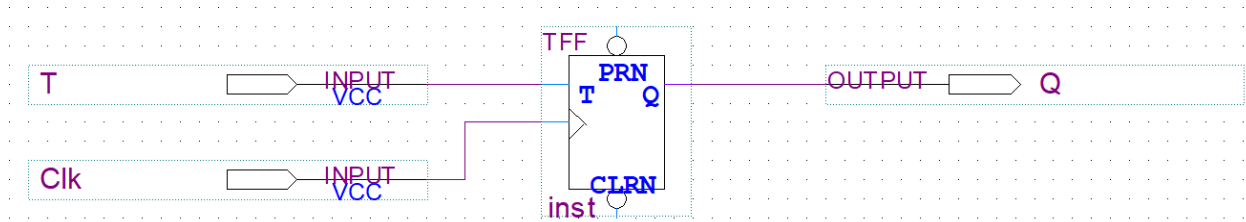
INPUT			OUTPUT		
Signal	Switch No	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	Q	LEDR0	PIN_AE23
D	SW0	PIN_N25			

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.

INPUT			OUTPUT		
Signal	Switch No	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	Q	LEDR0	PIN_AE23
T	SW0	PIN_N25			

5. Verify the functionality of your schematic.

Experiment: 8**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.

There are many types of shift registers. Here we have described two types –

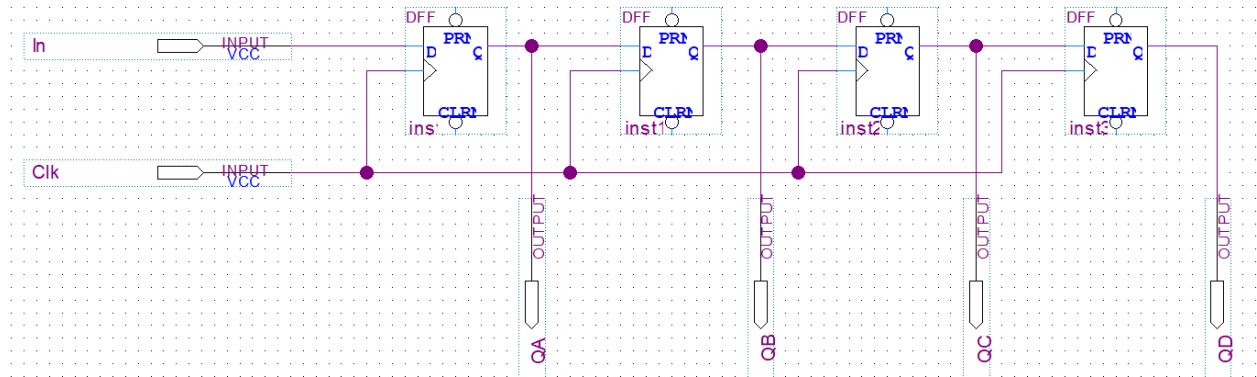
- Serial Register
 - Serial in Serial out Shift register
- Parallel Register
 - Parallel in parallel out shift register

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.

Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	QA	LEDR3	PIN_AC22
In	SW0	PIN_N25	QB	LEDR2	PIN_AB21
			QC	LEDR1	PIN_AF23
			QD	LEDR0	PIN_AE23

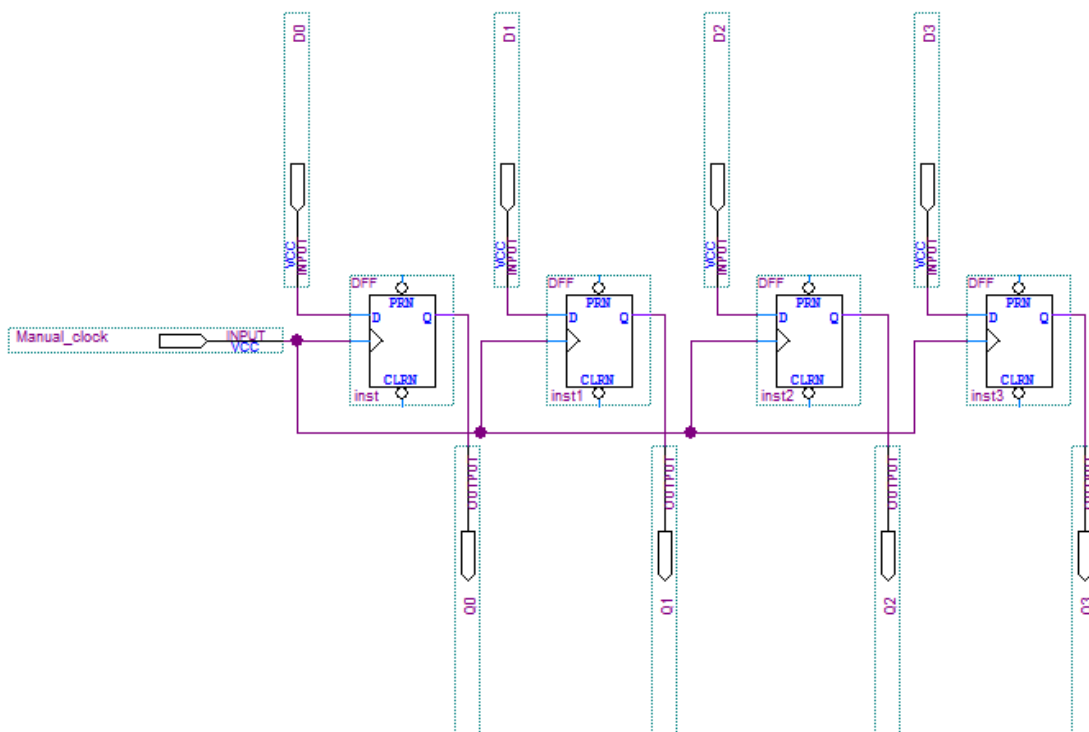
3. Test the functionality of your designed circuit.

Parallel in parallel out shift register

For a register with parallel input, the bits are entered simultaneously in to 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.

Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
D0	SW0	PIN_N25	Q0	LEDR0	PIN_AE23
D1	SW1	PIN_N26	Q1	LEDR1	PIN_AF23
D2	SW2	PIN_P25	Q2	LEDR2	PIN_AB21
D3	SW3	PIN_AE14	Q3	LEDR3	PIN_AC22
Manual_clock	KEY3	PIN_W26			

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. Two of most common types of shift register counters, the Johnson counter and the ring counter, are discussed here.

JOHNSON COUNTER

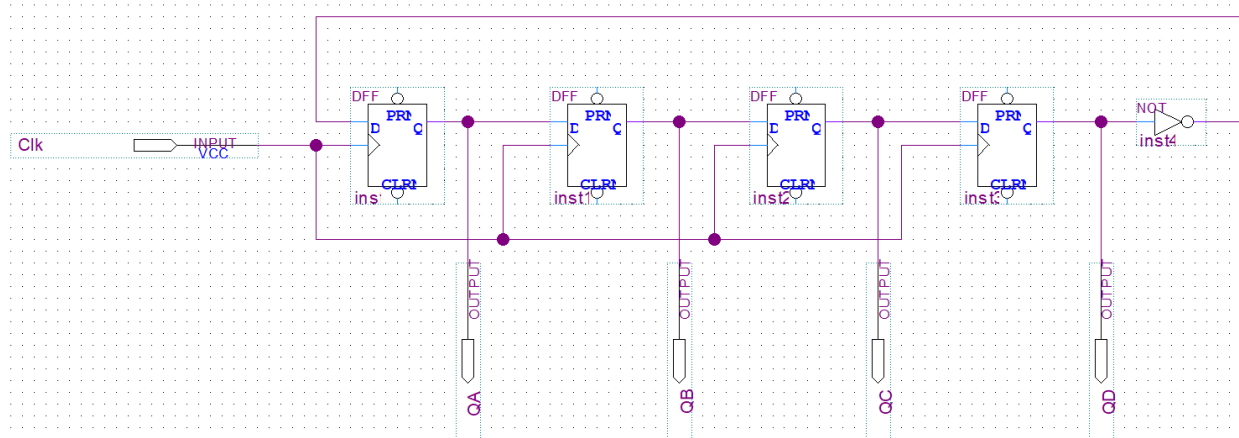
In a Johnson counter the complement of the last flipflop is connected back to the D input of the first flipflop. This feedback arrangement produces a characteristic sequence of states which is shown in the following table for a 4-bit device. 4-bit sequence has a total of eight states or bit patterns.

Clock Pulse	QA	QB	QC	QD
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

Table: 4-bit Johnson counter sequences.

Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	QA	LEDR3	PIN_AC22
			QB	LEDR2	PIN_AB21
			QC	LEDR1	PIN_AF23
			QD	LEDR0	PIN_AE23

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

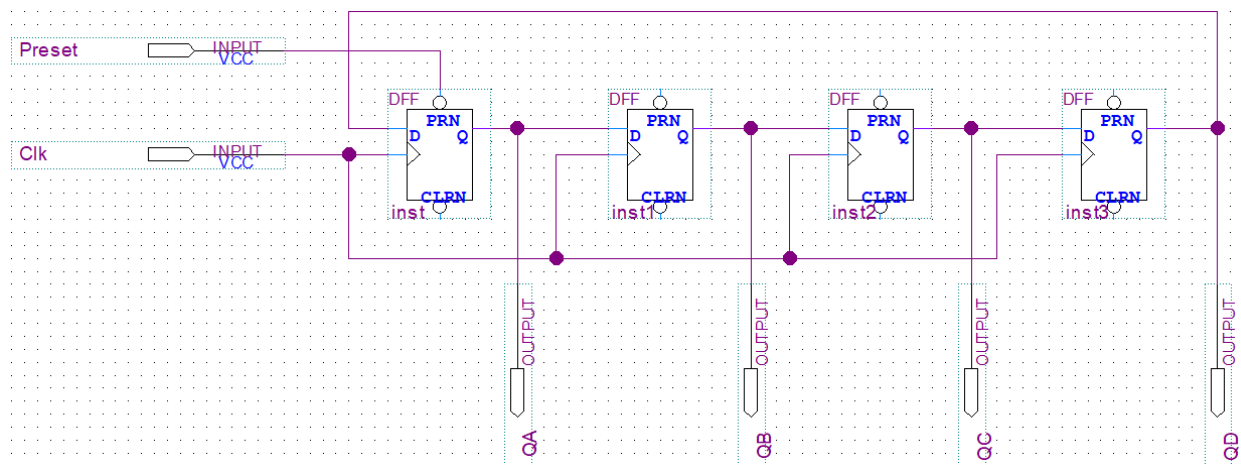
RING COUNTER

The ring counter utilizes one flipflop for each state in the sequence. It has the advantage that decoding gates are not required.

Clock Pulse	QA	QB	QC	QD
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Procedure for FPGA:

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:

INPUT			OUTPUT		
Signal	Switch No.	Pin No.	Signal	LED No.	Pin No.
Clk	KEY3	PIN_W26	QA	LEDR3	PIN_AC22
Preset	SW0	PIN_N25	QB	LEDR2	PIN_AB21
			QC	LEDR1	PIN_AF23
			QD	LEDR0	PIN_AE23

Experiment: 9**Experiment name:** *Introduction to CMOS INVERTER***Objectives:** Verify NOT, NAND, NOR gate using CMOS**Equipment:** Power supply, digital multimeter, potentiometer, and CMOS.**Background:**

CMOS is currently the most popular digital circuit technology. CMOS logic circuits are available as standard SSI and MSI packages for use in conventional digital system design. CMOS is also used in general-purpose VLSI circuits such as memory and microprocessors.

The CMOS Inverter is shown in figure 1. It consists of an N-channel MOSFET and a P-channel MOSFET. The input is applied to the two gates. The substrate of each transistor is connected to the source, and therefore no body effect for both transistors. When V_i is high, Q_N is ON and Q_P is OFF. The output is low. If V_i is low, Q_N is OFF and Q_P is ON. The output is high with V_{OH} .

Procedure:

1. Connect the circuit shown in Figure 1. Set the supply to V_{DD} 5V.
2. Verify the table for different values of inputs.
3. Connect the circuit shown in Figure 2. Set the supply to V_{DD} 5V.
4. Complete table 2 for different values of inputs.

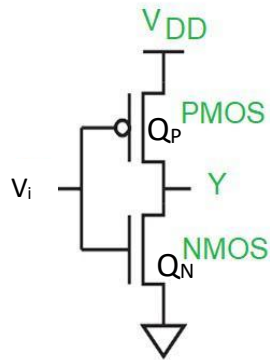
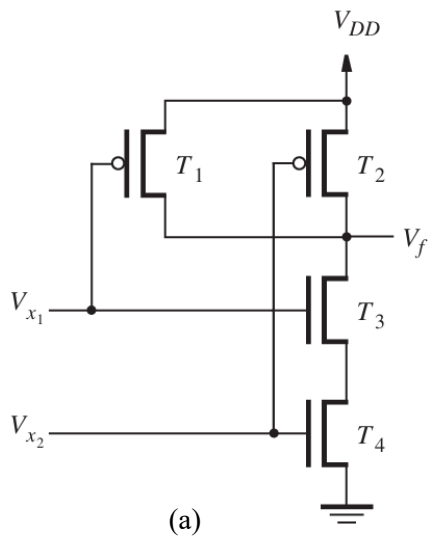


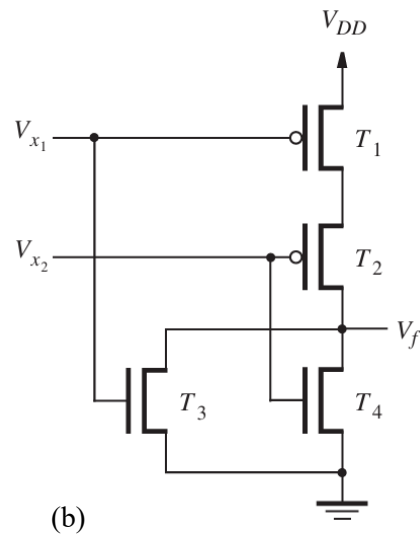
Figure 1: CMOS as an Inverter

V_i	Y

Table 1: Truth Table verification for NAND and NOR



(a)



(b)

Figure 2: CMOS as a) NAND b) NOR

CMOS as NAND Gate		
V_{X1}	V_{X2}	V_f
0	0	
0	1	
1	0	
1	1	

CMOS as NOR Gate		
V_{X1}	V_{X2}	V_f
0	0	
0	1	
1	0	
1	1	

Table 2: Truth Table verification for NAND and NOR

Report:

Solve the following exercises on separate sheets of paper and submit your solution

3. Analyze the circuit of Fig.1.
4. Discuss the working principle of CMOS and NAND and NOR gate
5. Design Ex-OR and Ex-NOR using CMOS.

Experiment: 10**Experiment name:** *Introduction to EMITTER-COUPLED LOGIC (ECL)***Objective:**

To demonstrate the operation of a simplified version of the ECL gate made by using discrete components.

Equipments: Power supply, digital multimeter, potentiometer (100 k Ω), diodes, transistors and resistors.

Background:

The first part of this experiment deals with the reference voltage used in the ECL circuit. The experimentally measured value will be compared with the theoretically calculated value. In the second part, a two-input ECL gate will be made excluding the emitter-follower output stages.

Procedure:

1. Construct the circuit as shown in figure1. Adjust the potentiometer to get -5V at VEE.
2. Measure the reference voltage V_R .
3. Construct the simplified ECL gate excluding the output stages as shown in Figure 2.
4. Let $V(1) = -0.75$ V and $V(0) = -1.75$ V. For different combinations of voltages, measure the two output voltages and record their values in Table I. Also, measure the voltage at point E in the circuit shown in Figure 2.

Table I: Measurements of voltages for the ECL gate

V_A (Volts)	V_B (Volts)	V_{O1} (Volts)	V_{O2} (Volts)	V_E (Volts)
- 1.75	- 1.75			
- 1.75	- 0.75			
- 0.75	- 1.75			
- 0.75	- 0.75			

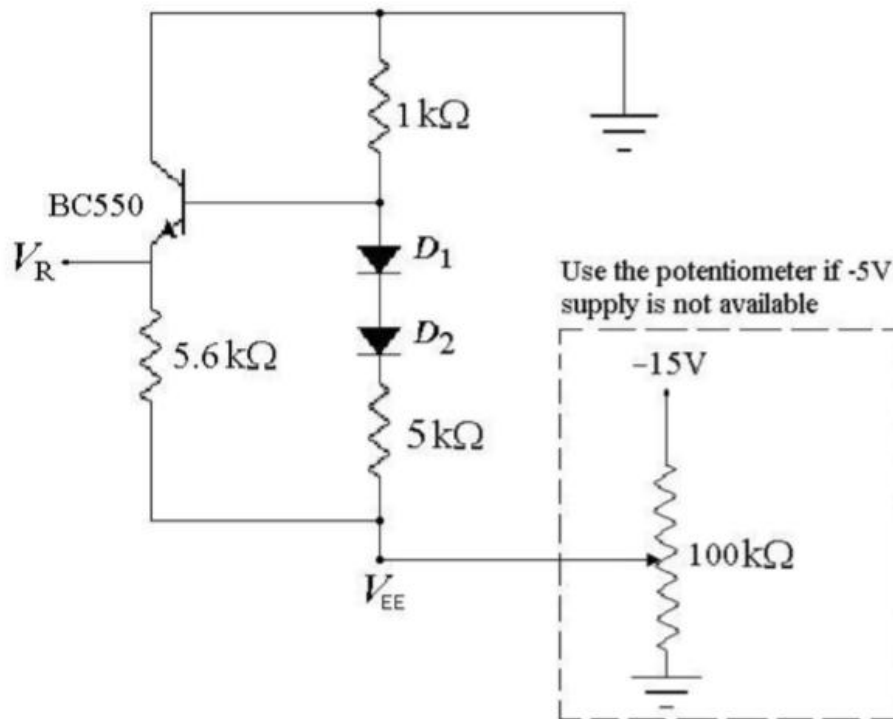


Figure 1 : Circuit for obtaining reference voltage.

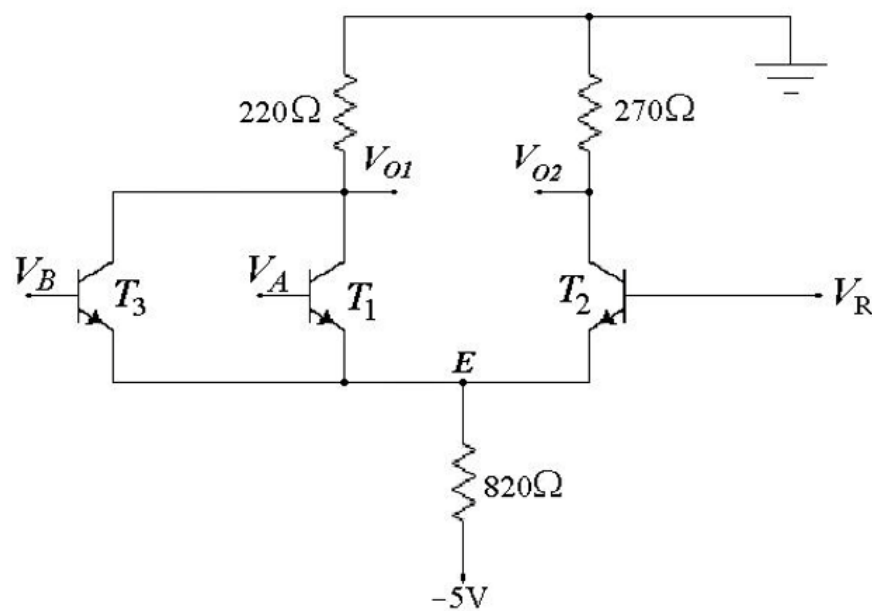


Figure 2 : Simplified ECL gate.

Table II : Output voltage V_{O2} versus the input voltage V_A for V_B set to logic '0'

V_A (V)	0.0	- 0.2	- 0.3	- 0.4	- 0.5	- 0.6	- 0.7	- 0.8	- 0.9
V_{O2} (V)									
V_A (V)	- 1.0	- 1.1	- 1.2	- 1.3	- 1.4	- 1.5	- 1.6	- 1.8	- 2.0
V_{O2} (V)									

6. Using the results obtained in step 4 above , plot the voltage transfer characteristics (V_{O2} versus V_A). From these characteristics determine the noise margins by completing the entries in Table III.

Table III: Determination of the noise margins for the ECL gate.

V_{OH} (V)	V_{IH} (V)	NM_H (V)	V_{IL} (V)	V_{OL} (V)	NM_L (V)

7. Disconnect the circuit and measure the resistances of all the resistors used in the experiment. Record their values.

Discussion:

1. Compare the experimentally obtained value of reference voltage V_R with the theoretically calculated value. Explain the difference between the two values.
2. On the basis of measured voltages in Table I, identify which output is for OR operation and which output is for NOR operation .
3. Using the measured voltages in Table I, determine the mode of operation of each transistor for various combinations of input voltages. Compare your results with theoretically expected modes of operation for these transistors.
4. Why the two levels of output voltage are not the same as the logic '0' and logic '1' voltages used in the experiment. Explain

Report:

.

Solve the following exercises on separate sheets of paper and submit your solution

1. Analyze the circuit of Fig.2
2. Simulate circuit of Fig. 2, using PSPICE.

ANNEXURE I

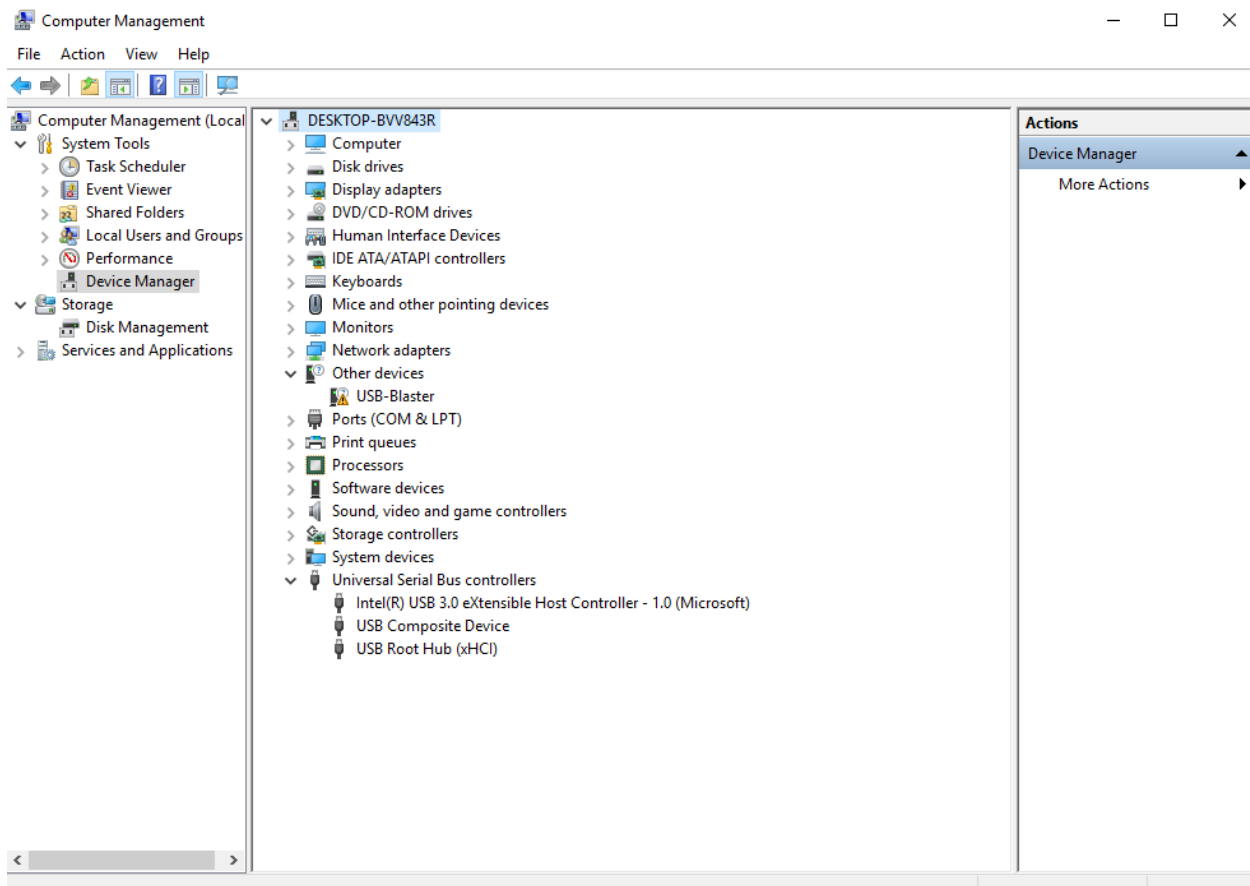
Installing USB-Blaster driver software on Windows 7

1. Set the **RUN/PROG** switch to the **RUN** position.
2. Connect the supplied USB cable to the **USB-Blaster port** of the FPGA and to a USB port of the PC. Also connect the 9V power supply adapter and turn the power switch **ON**.

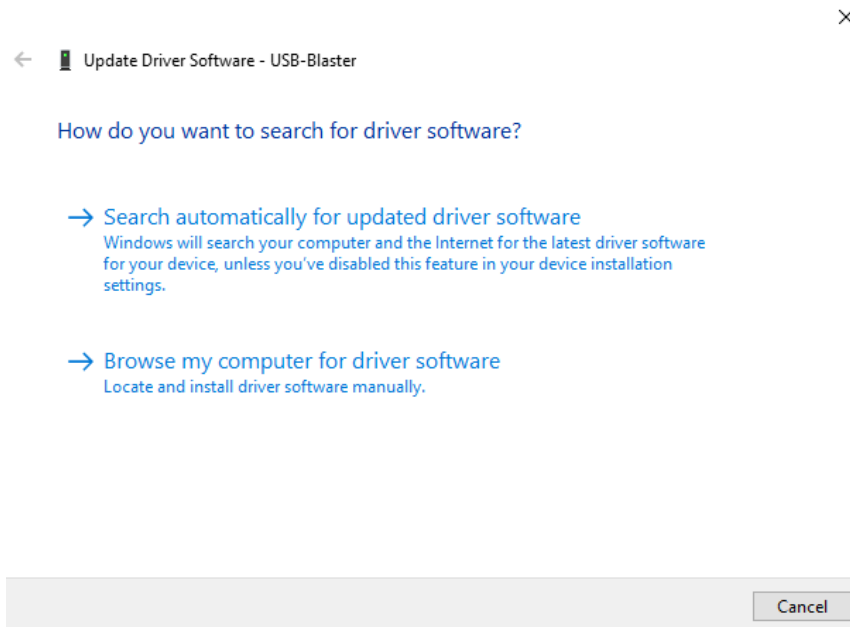
At this point you should observe the following:

- All user LEDs are flashing
- All 7-segment displays are cycling through the numbers 0 to F
- The LCD display shows Welcome to the Altera DE2 Board

3. Open **Device Manager**.



4. Note that, **USB-Blaster** is listed under **Other devices**. Right click on it and select **Update Driver Software**. **Update Driver Software –USB-Blaster** window will open up.

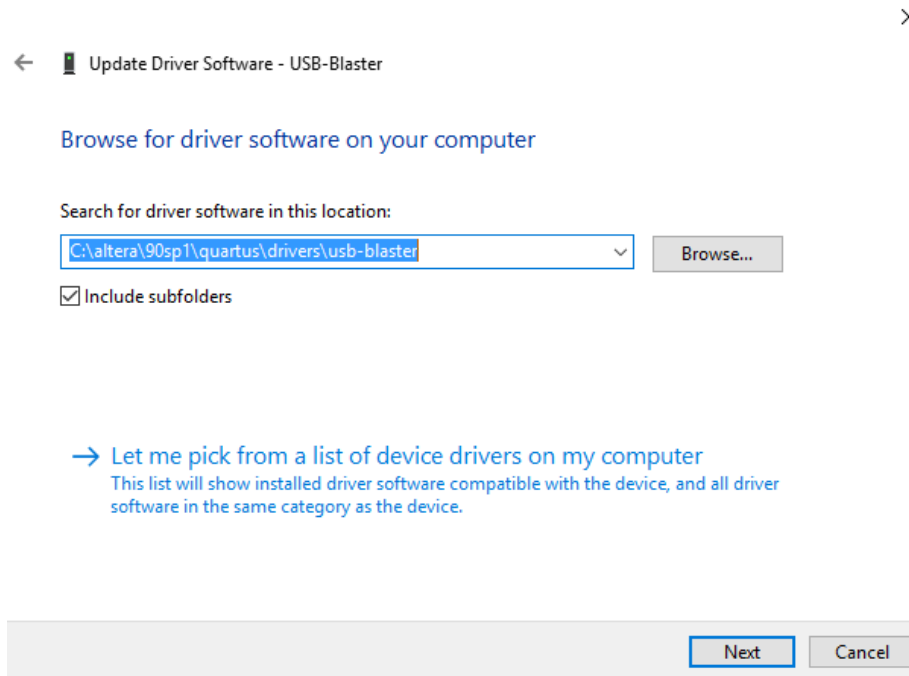


5. Select **Browse my computer for driver software**.

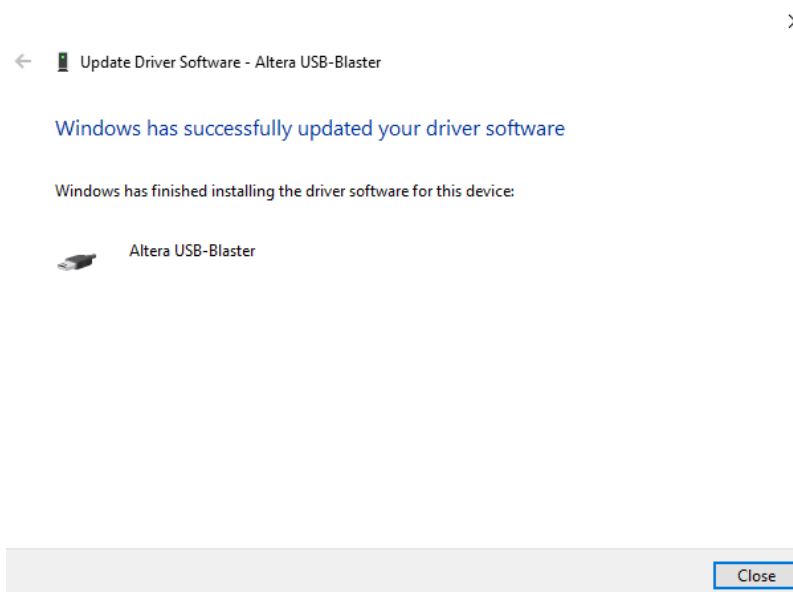
6. Find the location of **USB-Blaster driver software** from the installation directory of **Quartus II**. It will be under

<your installation directory>\altera\90sp1\quartus\drivers\usb-blaster

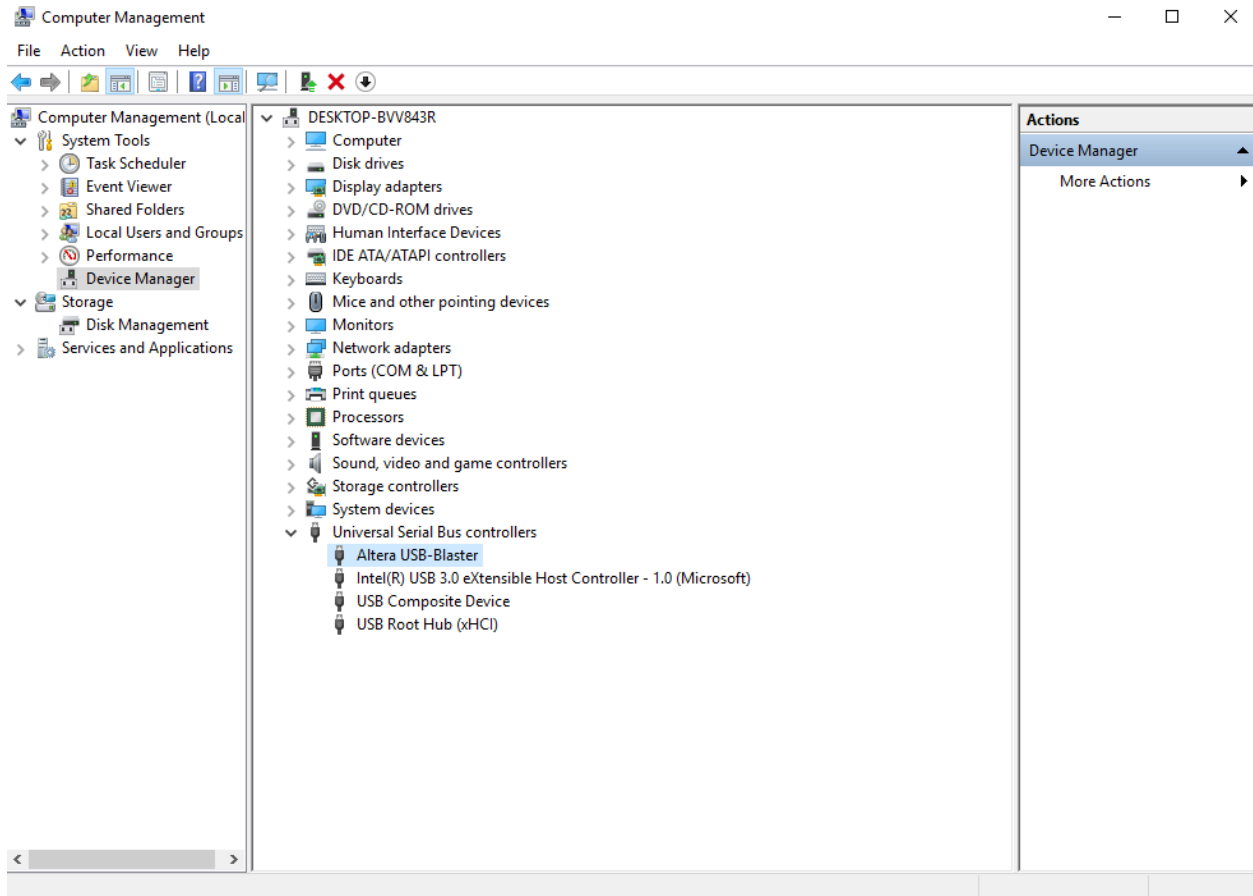
for Quartus II 9.0 sp1 web edition.



7. Select **Install this software anyway** if Windows Security prompt appears.



8. After successful installation, **Altera USB-Blaster** will appear under **Universal Serial Bus controllers** in **Device Manager** window:



ANNEXURE II

Using the LEDs and Switches

The DE2 board provides four pushbutton switches. Each of these switches is debounced using a Schmitt Trigger circuit. The four outputs called *KEY0*, ..., *KEY3* of the Schmitt Trigger device are connected directly to the Cyclone II FPGA. Each switch provides a high logic level (3.3 volts) when it is not pressed, and provides a low logic level (0 volts) when depressed. Since the pushbutton switches are debounced, they are appropriate for use as clock or reset inputs in a circuit. There are also 18 toggle switches (sliders) on the DE2 board. These switches are not debounced, and are intended for use as level-sensitive data inputs to a circuit. Each switch is connected directly to a pin on the Cyclone II FPGA. When a switch is in the DOWN position (closest to the edge of the board) it provides a low logic level (0 volts) to the FPGA, and when the switch is in the UP position it provides a high logic level (3.3 volts).

There are 27 user-controllable LEDs on the DE2 board. Eighteen red LEDs are situated above the 18 toggle switches, and eight green LEDs are found above the pushbutton switches (the 9th green LED is in the middle of the 7-segment displays). Each LED is driven directly by a pin on the Cyclone II FPGA; driving its associated pin to a high logic level turns the LED on, and driving the pin low turns it off.

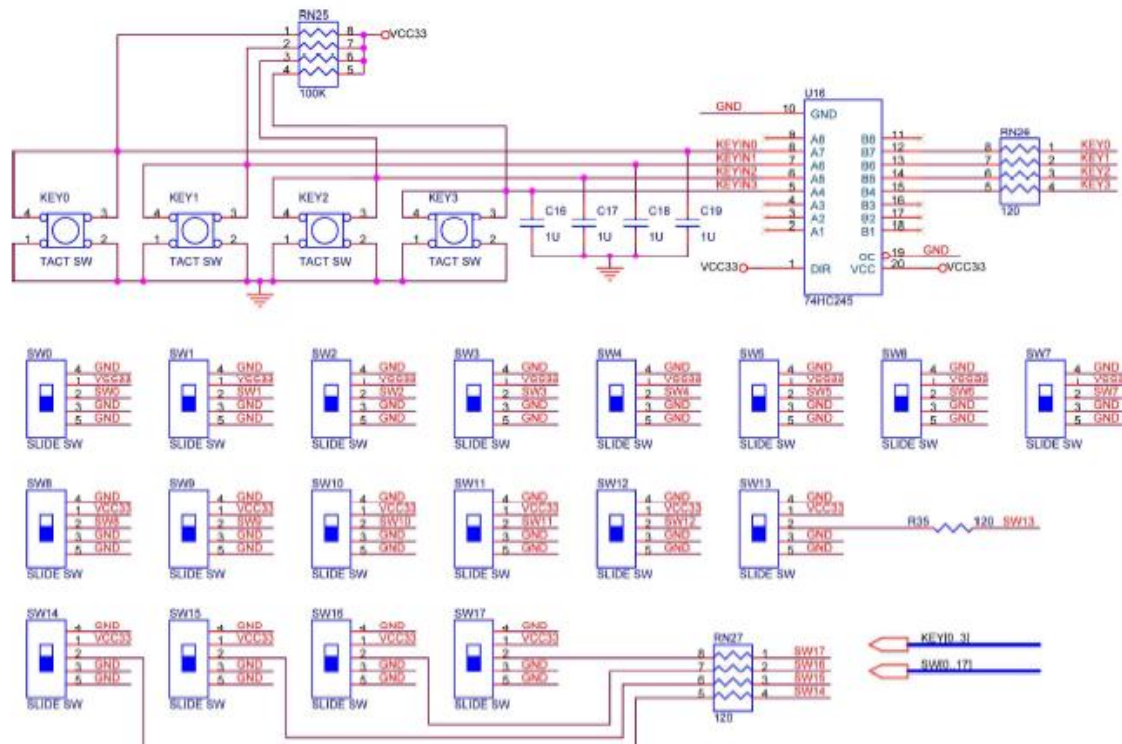


Figure1: Schematic diagram of push button and toggle switches

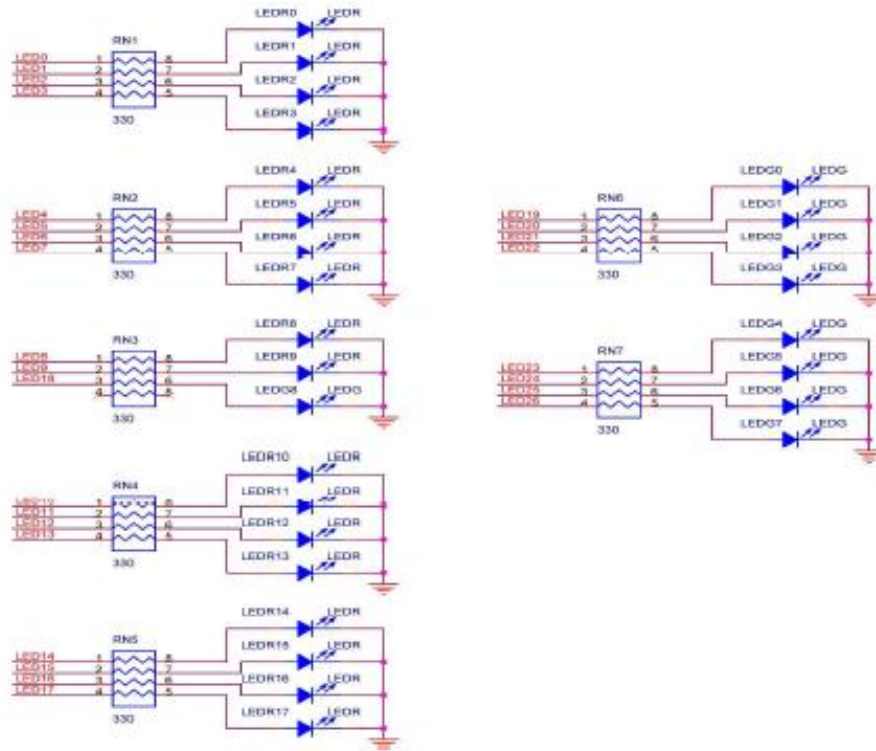


Figure 2: Schematic diagram of LEDs

Signal Name	FPGA Pin No.	Description
SW[0]	PIN_N25	Toggle Switch[0]
SW[1]	PIN_N26	Toggle Switch[1]
SW[2]	PIN_P25	Toggle Switch[2]
SW[3]	PIN_AE14	Toggle Switch[3]
SW[4]	PIN_AF14	Toggle Switch[4]
SW[5]	PIN_AD13	Toggle Switch[5]
SW[6]	PIN_AC13	Toggle Switch[6]
SW[7]	PIN_C13	Toggle Switch[7]
SW[8]	PIN_B13	Toggle Switch[8]
SW[9]	PIN_A13	Toggle Switch[9]
SW[10]	PIN_N1	Toggle Switch[10]
SW[11]	PIN_P1	Toggle Switch[11]
SW[12]	PIN_P2	Toggle Switch[12]
SW[13]	PIN_T7	Toggle Switch[13]
SW[14]	PIN_U3	Toggle Switch[14]
SW[15]	PIN_U4	Toggle Switch[15]
SW[16]	PIN_V1	Toggle Switch[16]
SW[17]	PIN_V2	Toggle Switch[17]

Table1: Pin Assignments for toggle switches

Signal Name	FPGA Pin No.	Description
KEY[0]	PIN_G26	Pushbutton[0]
KEY[1]	PIN_N23	Pushbutton[1]
KEY[2]	PIN_P23	Pushbutton[2]
KEY[3]	PIN_W26	Pushbutton[3]

Table 2: Pin Assignments for push button switches

Signal Name	FPGA Pin No.	Description
LEDR[0]	PIN_AE23	LED Red[0]
LEDR[1]	PIN_AF23	LED Red[1]
LEDR[2]	PIN_AB21	LED Red[2]
LEDR[3]	PIN_AC22	LED Red[3]
LEDR[4]	PIN_AD22	LED Red[4]
LEDR[5]	PIN_AD23	LED Red[5]
LEDR[6]	PIN_AD21	LED Red[6]
LEDR[7]	PIN_AC21	LED Red[7]
LEDR[8]	PIN_AA14	LED Red[8]
LEDR[9]	PIN_Y13	LED Red[9]
LEDR[10]	PIN_AA13	LED Red[10]
LEDR[11]	PIN_AC14	LED Red[11]
LEDR[12]	PIN_AD15	LED Red[12]
LEDR[13]	PIN_AE15	LED Red[13]
LEDR[14]	PIN_AF13	LED Red[14]
LEDR[15]	PIN_AE13	LED Red[15]
LEDR[16]	PIN_AE12	LED Red[16]
LEDR[17]	PIN_AD12	LED Red[17]
LEDG[0]	PIN_AE22	LED Green[0]
LEDG[1]	PIN_AF22	LED Green[1]
LEDG[2]	PIN_W19	LED Green[2]
LEDG[3]	PIN_V18	LED Green[3]
LEDG[4]	PIN_U18	LED Green[4]
LEDG[5]	PIN_U17	LED Green[5]
LEDG[6]	PIN_AA20	LED Green[6]
LEDG[7]	PIN_Y18	LED Green[7]
LEDG[8]	PIN_Y12	LED Green[8]

Table 3: Pin Assignments for LEDs

Using the 7-segment Displays

The DE2 Board has eight 7-segment displays. These displays are arranged into two pairs and a group of four, with the intent of displaying numbers of various sizes. As indicated in the schematic in Figure 4.6, the seven segments are connected to pins on the Cyclone II FPGA. 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, with the positions given in Figure 4.7. Note that the dot in each display is unconnected and cannot be used. Table 4.4 shows the assignments of FPGA pins to the 7-segment displays.

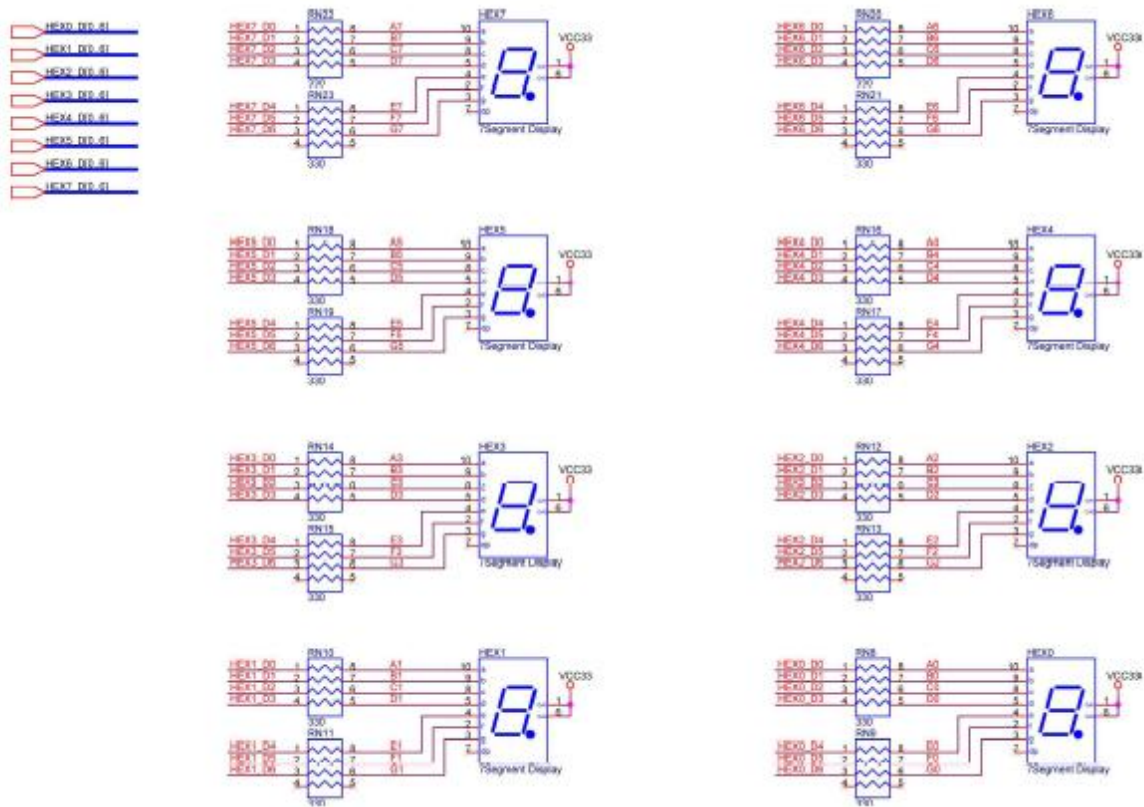


Figure 3: Schematic diagram of 7 segment displays

Signal Name	FPGA Pin No.	Description
HEX0[0]	PIN_AF10	Seven Segment Digit 0[0]
HEX0[1]	PIN_AB12	Seven Segment Digit 0[1]
HEX0[2]	PIN_AC12	Seven Segment Digit 0[2]
HEX0[3]	PIN_AD11	Seven Segment Digit 0[3]
HEX0[4]	PIN_AE11	Seven Segment Digit 0[4]
HEX0[5]	PIN_V14	Seven Segment Digit 0[5]
HEX0[6]	PIN_V13	Seven Segment Digit 0[6]
HEX1[0]	PIN_V20	Seven Segment Digit 1[0]
HEX1[1]	PIN_V21	Seven Segment Digit 1[1]
HEX1[2]	PIN_W21	Seven Segment Digit 1[2]
HEX1[3]	PIN_Y22	Seven Segment Digit 1[3]
HEX1[4]	PIN_AA24	Seven Segment Digit 1[4]
HEX1[5]	PIN_AA23	Seven Segment Digit 1[5]
HEX1[6]	PIN_AB24	Seven Segment Digit 1[6]
HEX2[0]	PIN_AB23	Seven Segment Digit 2[0]
HEX2[1]	PIN_V22	Seven Segment Digit 2[1]
HEX2[2]	PIN_AC25	Seven Segment Digit 2[2]
HEX2[3]	PIN_AC26	Seven Segment Digit 2[3]
HEX2[4]	PIN_AB26	Seven Segment Digit 2[4]
HEX2[5]	PIN_AB25	Seven Segment Digit 2[5]
HEX2[6]	PIN_Y24	Seven Segment Digit 2[6]
HEX3[0]	PIN_Y23	Seven Segment Digit 3[0]
HEX3[1]	PIN_AA25	Seven Segment Digit 3[1]
HEX3[2]	PIN_AA26	Seven Segment Digit 3[2]
HEX3[3]	PIN_Y26	Seven Segment Digit 3[3]
HEX3[4]	PIN_Y25	Seven Segment Digit 3[4]
HEX3[5]	PIN_U22	Seven Segment Digit 3[5]
HEX3[6]	PIN_W24	Seven Segment Digit 3[6]
HEX4[0]	PIN_U9	Seven Segment Digit 4[0]
HEX4[1]	PIN_U1	Seven Segment Digit 4[1]
HEX4[2]	PIN_U2	Seven Segment Digit 4[2]
HEX4[3]	PIN_T4	Seven Segment Digit 4[3]
HEX4[4]	PIN_R7	Seven Segment Digit 4[4]
HEX4[5]	PIN_R6	Seven Segment Digit 4[5]
HEX4[6]	PIN_T3	Seven Segment Digit 4[6]

HEX5[0]	PIN_T2	Seven Segment Digit 5[0]
HEX5[1]	PIN_P6	Seven Segment Digit 5[1]
HEX5[2]	PIN_P7	Seven Segment Digit 5[2]
HEX5[3]	PIN_T9	Seven Segment Digit 5[3]
HEX5[4]	PIN_R5	Seven Segment Digit 5[4]
HEX5[5]	PIN_R4	Seven Segment Digit 5[5]
HEX5[6]	PIN_R3	Seven Segment Digit 5[6]
HEX6[0]	PIN_R2	Seven Segment Digit 6[0]
HEX6[1]	PIN_P4	Seven Segment Digit 6[1]
HEX6[2]	PIN_P3	Seven Segment Digit 6[2]
HEX6[3]	PIN_M2	Seven Segment Digit 6[3]
HEX6[4]	PIN_M3	Seven Segment Digit 6[4]
HEX6[5]	PIN_M5	Seven Segment Digit 6[5]
HEX6[6]	PIN_M4	Seven Segment Digit 6[6]
HEX7[0]	PIN_L3	Seven Segment Digit 7[0]
HEX7[1]	PIN_L2	Seven Segment Digit 7[1]
HEX7[2]	PIN_L9	Seven Segment Digit 7[2]
HEX7[3]	PIN_L6	Seven Segment Digit 7[3]
HEX7[4]	PIN_L7	Seven Segment Digit 7[4]
HEX7[5]	PIN_P9	Seven Segment Digit 7[5]
HEX7[6]	PIN_N9	Seven Segment Digit 7[6]

Table 4: Pin diagram for seven segment displays

REFERENCE

1. Digital logic and computer design by M. Morris Mano.
2. Digital fundamentals by Thomas L. Floyd.
3. Fundamentals of Digital Logic with Verilog Design by Stephen Brown and Zvonko Vranesic.
4. Digital systems principles and applications by Ronald J. Tocci.