



## Experiment 2

### Basic Logic Gates Implementation Using Breadboards and Discrete Gates

#### Introduction:

Logic functions can be implemented in several ways. In the past, vacuum tube and relay circuits performed logic functions. Presently logic functions are performed by tiny integrated circuits (ICs). These ICs are small silicon semiconductor sheets called chips, containing the electronic components for the logic gates. The chip is mounted in a plastic container, and connections are welded to external pins may range from 14 in a small IC package to 64 or more in a large one.

#### Objectives:

- Understand how to use the breadboard to patch up, test your logic design and debug it.
- Wire and operate logic gates such as AND, OR, NOT, NAND, NOR, XOR.
- Understand how to implement simple circuits based on a schematic diagram using logic gates.

#### Theoretical Background:

##### Types of integrated circuits (ICs)

The different sizes of integration of IC chips are usually defined in terms of the number of logic gates in a single IC or package. They are classified in one of the following categories:

1. Small-scale integration (SSI) device: contains less than 10 gates in a single package, such as logic gates.
2. Medium scale integration (MSI) device: contains 10 -100 gates in a single package, such as adders and decoders.
3. Large-scale integration (LSI) device: contains 100 to 10000 gates in a single package, such as processors.
4. Very large-scale integration (VLSI) device: contains more than 10000 gates in a single package, such as complex microprocessors chips.

##### Logic Families

The types of logic devices are classified in "families", of which the most important are TTL and CMOS. The main families are:

- TTL (Transistor-Transistor Logic) made of bipolar transistors.
- CMOS (Complementary Metal Oxide Semiconductor) made from MOSFETs
- ECL (Emitter Coupled Logic) for extremely high speeds
- NMOS, PMOS for VLSI large scale integrated circuits.

##### Subfamilies of TTL Family

There are subfamilies or series of the TTL. Commercial TTL ICs has a number designation that starts with 74 and follows with a suffix that identifies the series type. These subfamilies are:

Standard: 74xx, High speed 74hxx, low power 74Lxx, Shottky TTL 74Sxx, Low power shottky 74LSxx, Advanced shottky 74ASxx, Advanced low-power shottky 74ALSxx.

All TTL IC's are designed to operate from 5V power supply. The input and output logic levels are illustrated in the Figure 1.

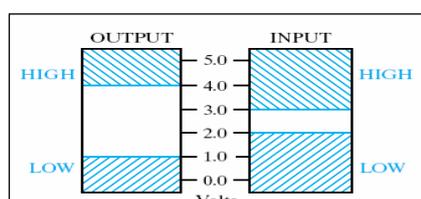


Figure 1

## Some characteristics of the TTL family

1. Power dissipation: It is the amount of power needed by the gate delivered from the power supply. It is equal to 20 mw per gate. Power dissipation is useful to estimate the total power consumption of a system, as an example it will help in portable equipment to know what type of battery might be needed.
2. Fan-in: it is the number of inputs that the gate is designed to have, the maximum inputs is 8 inputs per gate.
3. Fan-out: it is the maximum number of inputs that can be connected to the output of the gate without affecting its normal operation. It is 12 gates.
4. Propagation/time delay: it is the amount of delay between applying the input and the response of the output of the gate. Generally, the propagation delay is in the range of 0.5 to 50 nanoseconds. The total propagation delay time of a logic system will be the delay gate multiplied by the number of gates in series. It is 10 ns per gate.

## Practical TTL Logic Gates

A popular type of IC is illustrated in Figure 2. IC manufacturers refer to this case style as a *dual-in-line package* (DIP). This particular IC is called a 14-pin DIP IC. Just counterclockwise from the notch on the IC is pin 1. A dot (optional) on the top of the IC is another method used to locate pin 1.

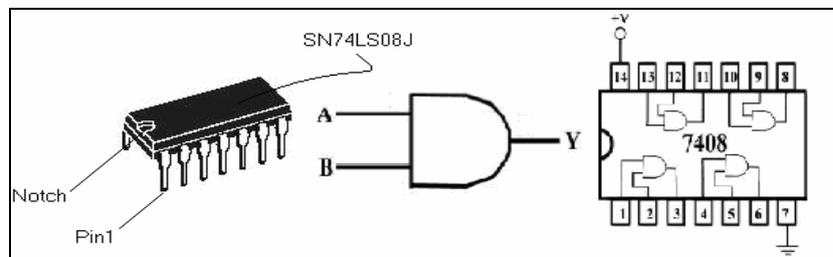


Figure 2

Part Number:

Part number is divided into three sections:

- The prefix: the manufacturer's code.
- Core part number: This determines the technology "TTL or CMOS", the device series and the function of a digital IC.
- The trailing letter(s) "the suffix" which is a code used by several manufacturers to design the DIP.

For example, the part number of:

SN74LS08J

SN: stands for the manufacturer "Texas Instruments"

74: 7400 TTL series

LS: low shottky type

08: function of a digital IC

J: Ceramic dual-in-line Package

## Breadboard

A breadboard is used to build and test circuits quickly before finalizing any circuit design. The breadboard has series of holes into which ICs can be inserted.

- Breadboard Construction:
  - The breadboard has a series of holes, each containing an electrical contact.
  - Holes in the same row (examples highlighted in yellow (1) in Figure3) are electrically connected(they are the same node),holes in other row (highlighted in green (2)) are different node, when you insert a wire into one hole then all the holes in the same node are electrically connected.
  - The gap (highlighted in pink (3)) marks a boundary between the electrical connections. A wire inserted in one of the green holes would not be connected to a wire inserted in one of the yellow holes.
  - The two top rows of holes at the top highlighted in red and blue are used for power supply connections. The first row (highlighted in blue (4)) is connected to ground, all the holes in this row are electrically connected.
  - The second row (highlighted in red (5) ) must connected to 5V , there are 40 holes in this row, each 10 holes are grouped together and electrically connected.
- Using a Breadboard
  1. Before building a circuit, connect 5V from the power supply to V1 (or V2 or V3) in the bread board and 0V the ground of the bread board as shown in the figure below.

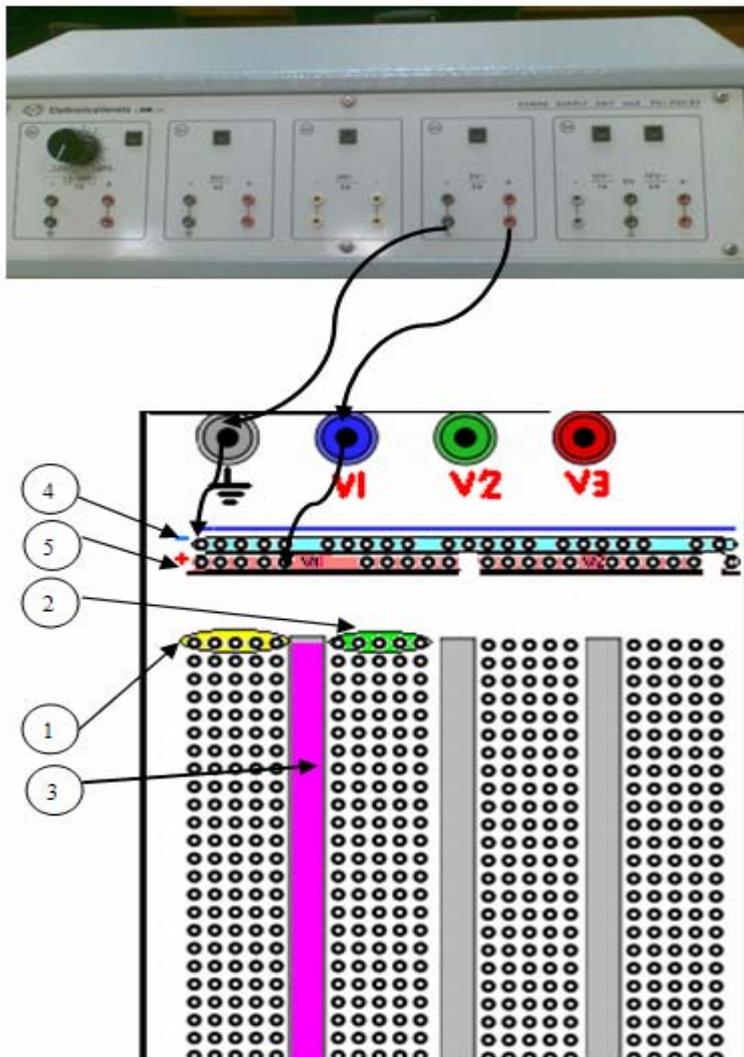


Figure 3

1. Use wires to connect V1 in the bread board to the red terminal(+), and ground of the bread board to the blue terminal(-).
2. Place the IC in the board so that pin 1 should be on the upper left of the board. Half of the legs should be on one side of the pink gap and half on the other.
3. Connect pin 14 of the IC chip to V<sub>cc</sub> and pin 7 to ground.
4. Connect pin 1 and 2 of the IC chip to the input (you can take the input from the two top rows that are connected to the power supply, holes in the first row for logic 0 and holes in the second row for logic 1).

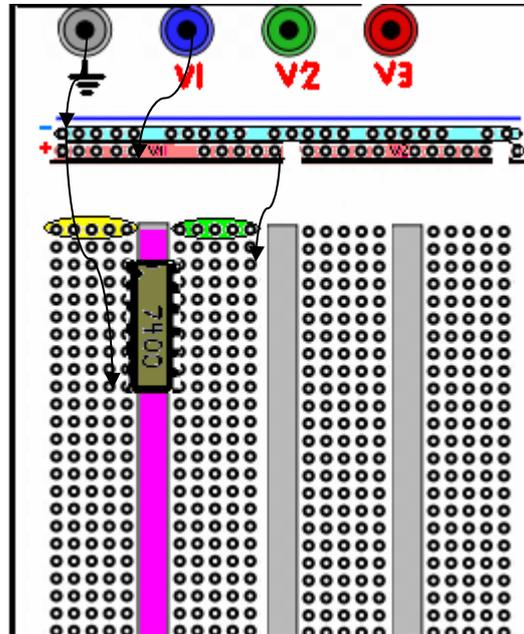


Figure 4

5. You can determine the output using the logic probe, logic probe as shown in (Figure5) is a hand-held pen-like probe used for analyzing and troubleshooting the logical states (Boolean 0 or 1) of a digital circuit. It can be used on either TTL or CMOS integrated circuit devices.
  - a. Attach red alligator clip to positive side of the power supply.
  - b. Attach black alligator clip to a negative side of the power supply.
  - c. Place the tip of the probe on the point you want to test. **Make sure that the switch is in TTL position.**

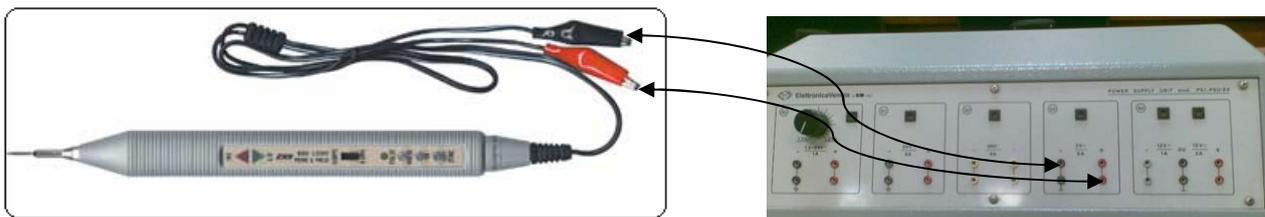


Figure 5

### Wiring Guidelines:

- Arrange the IC chips on the breadboard so that only short wire connections are needed.
- Try to keep the wire as short as possible to avoid a jungle of wires.
- Try to maintain a low wiring profile so that the pins of the chips can be reached and the chip replaced, if necessary. The best connections are those that lie flat on the board.

**Pay extra attention to power and ground. If you find your chips are getting super hot then there is probably a short circuit. Turn off power immediately and wire them correctly.**



## Introduction:

This tutorial introduces the basic features of the Quartus II software. It shows how the software can be used to design and implement a circuit specified by using the means of a schematic diagram. It makes use of the graphical user interface to invoke the Quartus II commands.

## Objectives:

- Creating a project.
- Design entry using schematic diagram.
- Assigning the circuit inputs and outputs to specific pins on the FPGA.
- Simulating the designed circuit.
- Programming and configuring the FPGA device.

## 1- Getting Started:

Each logic circuit, or sub circuit, being designed with Quartus II software is called a project. The software works on one project at a time and keeps all information for that project in a single directory (folder) in the file system.

To begin a new logic circuit design, the first step is to create a directory to hold its files. To hold the design files for this lab, we will use a directory Exp2. The running example for this Experiment is a simple circuit for Xor gate ( $A \text{ XOR } B = (A \& (\sim B)) \mid ((\sim A) \& B)$ ).

Start the Quartus II software. You should see a display similar to the one in Figure 1. This display consists of several windows that provide access to all the features of Quartus II software, which the user selects with the computer mouse. Most of the commands provided by Quartus II software can be accessed by using a set of menus that are located below the title bar (File, Edit, view, project...).

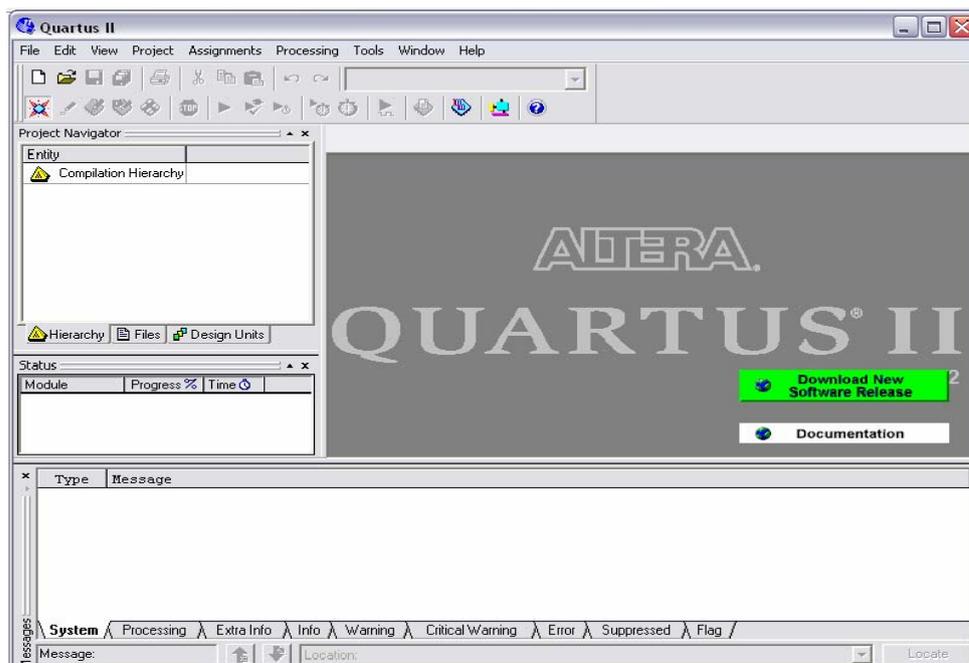


Figure 1. The main Quartus II display.

## 1.1 Quartus II Online Help

Quartus II software provides comprehensive online documentation that answers many of the questions that may arise when using the software. The documentation is accessed from the Help menu. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the Help menu. For instance, selecting Help > How to Use Help gives an indication of what type of help is provided. The user can quickly search through the Help topics by selecting Help > Search, which opens a dialog box into which keywords can be entered. Another method, context-sensitive help, is provided for quickly finding documentation for specific topics. While using most applications, pressing the F1 function key on the keyboard opens a Help display that shows the commands available for the application.

## 2- Starting a New Project

To start working on a new design we first have to define a new *design project*. Quartus II software makes the designer's task easy by providing support in the form of a *wizard*.

1. Create a new project; select **File > New Project Wizard** to reach the window in Figure 2b, which asks for the name and directory of the project.

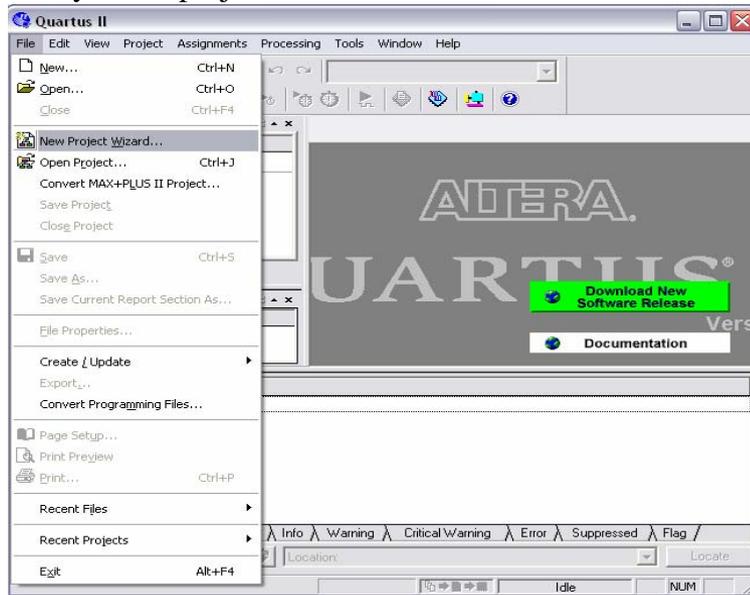


Figure 2a. Creation of a new project.

2. Set the working directory to be *Exp2*, The project must have a name, which is usually the same as the top-level design entity (schematic circuit) that will be included in the project. Choose *Xor1* as the name for both the project and the top-level entity, as shown in Figure 2b.

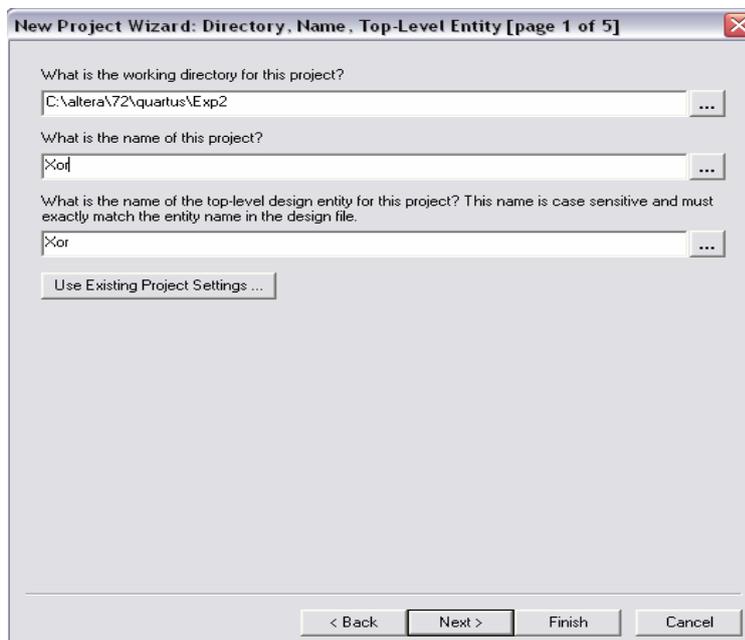


Figure 2b. Creation of a new project.

Press **Next**. Since we have not yet created the directory *Exp2*, Quartus II software displays the pop-up box in Figure 3 asking if it should create the desired directory. Click **Yes**, which leads to the window in Figure 4.

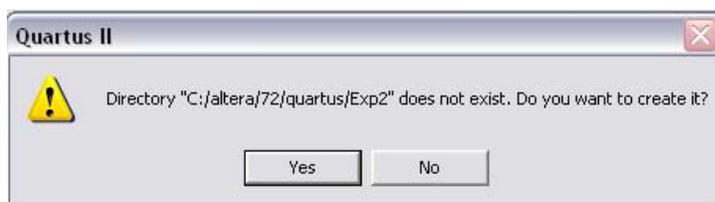


Figure 3. Quartus II software can create a new directory for the project.

3. The wizard makes it easy to specify which existing files (if any) should be included in the project. Assuming that we do not have any existing files, click **Next**, which leads to the window in Figure 5.

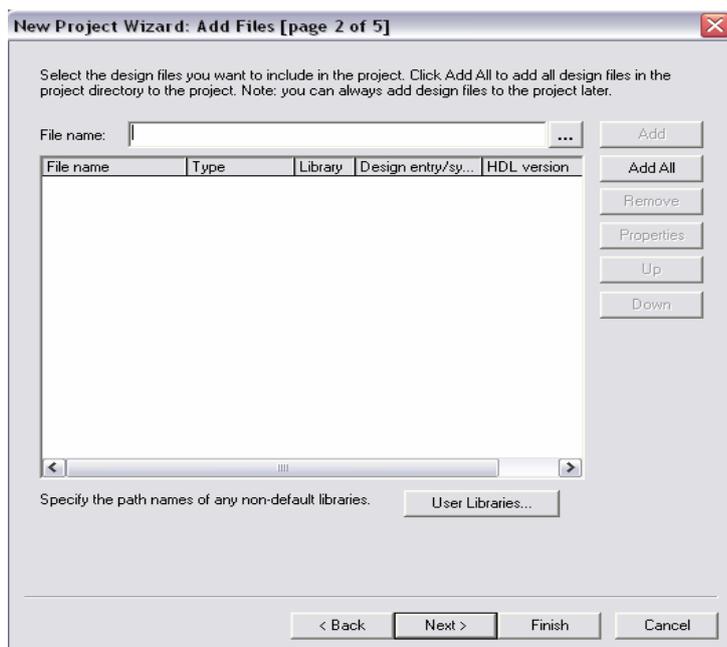


Figure 4. The wizard can include user-specified design files.

4. We have to specify the type of device in which the designed circuit will be implemented. Choose **CycloneII** as the target device family. We can let Quartus II software select a specific device in the family, or we can choose the device explicitly. We will take the latter approach. From the list of available devices, choose the device called **EP2C70F896C6** which is the FPGA used on Altera's DE2 board. Press **Next**, which opens the window in Figure 6.

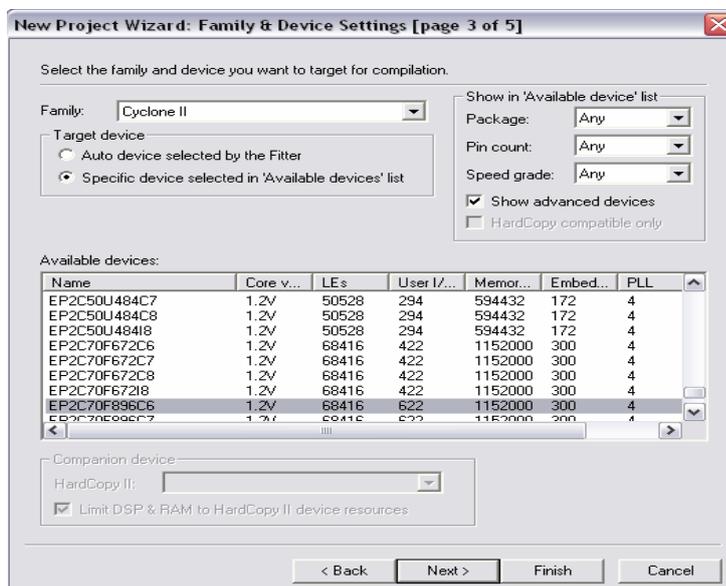


Figure 5. Choose the device family and a specific device

5. The user can specify any third-party tools that should be used. A commonly used term for CAD software for electronic circuits is EDA tools, where the acronym stands for Electronic Design Automation. This term is used in Quartus II messages that refer to third-party tools, which are the tools developed and marketed by companies other than Altera. Since we will rely solely on Quartus II tools, we will not choose any other tools. Press **Next**.

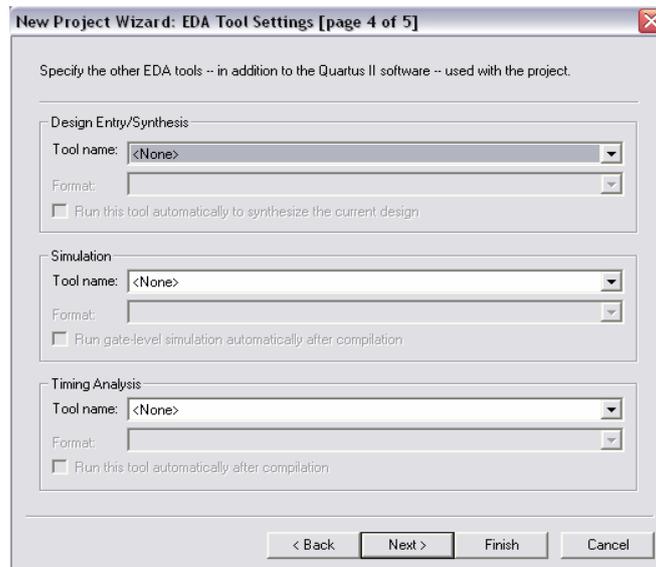


Figure 6. Other EDA tools can be specified.

6. A summary of the chosen settings appears in the screen shown in Figure 7. Press **Finish**, which returns to the main Quartus II window, but with Xor specified as the new project, in the display title bar, as indicated in Figure 8.

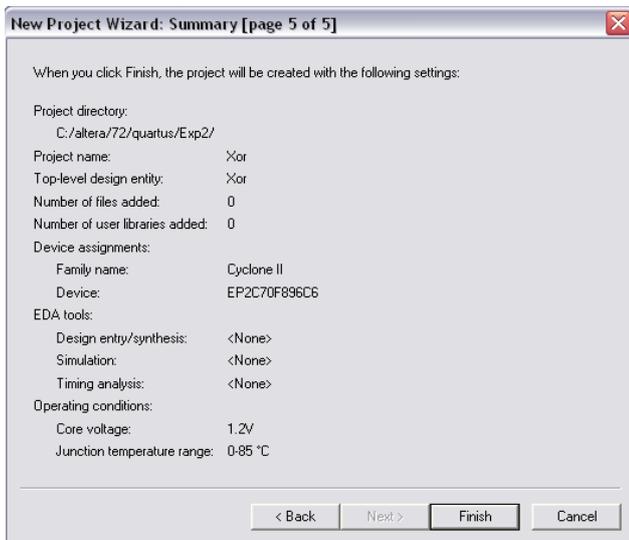


Figure 7. Summary of the project settings.

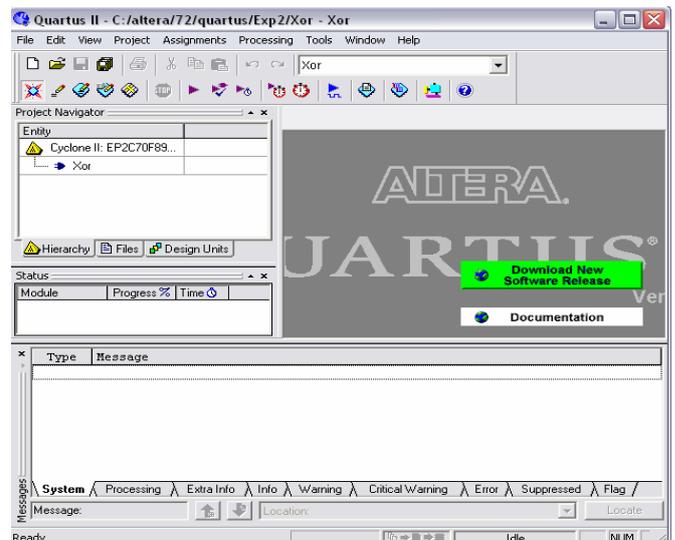


Figure 8. The Quartus II display for the created project.

### 3- Design Entry Using the Graphic Editor

As a design example, we will use the Xor circuit shown in Figure 9. The circuit has two input switches  $x1$  and  $x2$ , where a closed switch corresponds to the logic value 1. The truth table for the circuit is also given in the figure.

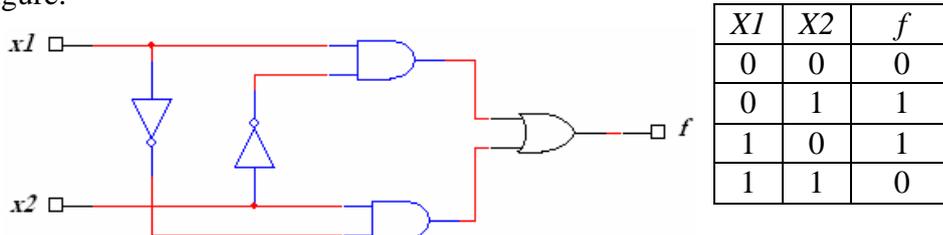


Figure 9. The Xor function circuit.

The Quartus II Graphic Editor can be used to specify a circuit in the form of a block diagram. Select **File > New** to get the window in Figure 10, choose **Block Diagram/Schematic File**, and click **OK**. This opens the Graphic Editor window.

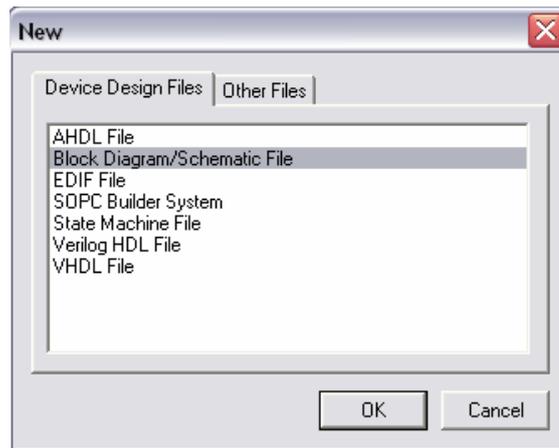


Figure 10. Choose to prepare a block diagram

The first step is to specify a name for the file that will be created. Select **File > Save As** to open the pop-up box depicted in Figure 11. In the box labeled **Save as type** choose **Block Diagram/Schematic File (\*.bdf)**. In the box labeled File name type Xor, to match the name given in Figure 2b, which was specified when the project was created. Put a checkmark in the box **Add file to current project**. Click **Save**, which puts the file into the directory Exp2 and leads to the Graphic Editor window displayed in Figure 12.

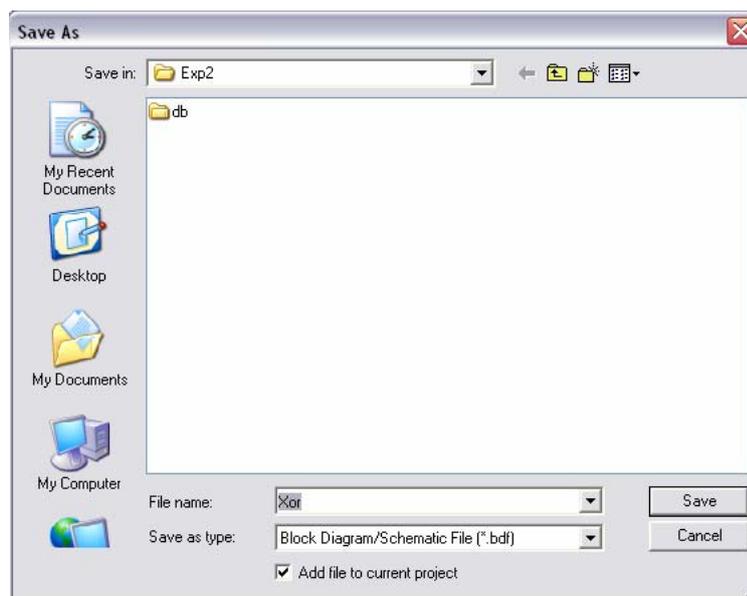


Figure 11. Name the file.

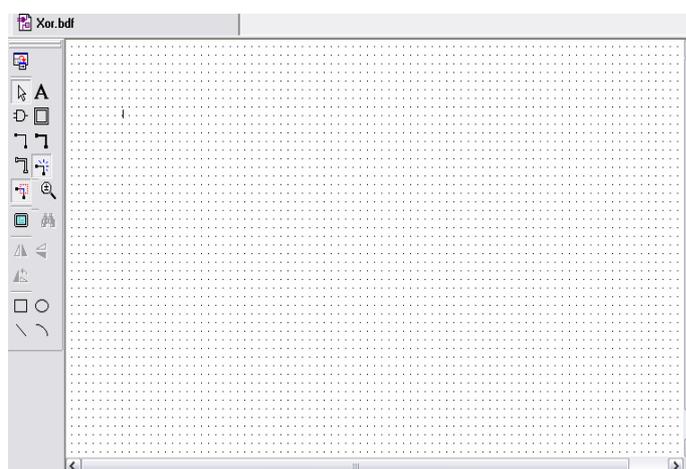


Figure 12. Graphic Editor window.

### 3.1 Importing Logic-Gate Symbols

The Graphic Editor provides a number of libraries which include circuit elements that can be imported into a schematic. **Double-click** on the blank space in the Graphic Editor window, or click on the icon in the toolbar that looks like an AND gate. A pop-up box in Figure 13 will appear.

**Expand** the hierarchy in the Libraries box as shown in the figure. First **expand libraries**, and then **expand the library primitives**, followed by **expanding the library logic** which comprises the logic gates. **Select and2**, which is a two-input AND gate, and click **OK**. Now, the AND gate symbol will appear in the Graphic Editor window. Using the mouse, move the symbol to a desirable location and **click** to place it there.

Import the second AND gate, which can be done simply by positioning the mouse pointer over the existing AND-gate symbol, right-clicking, and dragging to make a copy of the symbol. A symbol in the Graphic Editor window can be moved by clicking on it and dragging it to a new location with the mouse button pressed. Next, select or2 from the library and import the OR gate into the diagram. Then, select not and import two instances of the NOT gate. Rotate the NOT gates into proper position by using the "Rotate left 90" icon. Arrange the gates as shown in Figure 14.

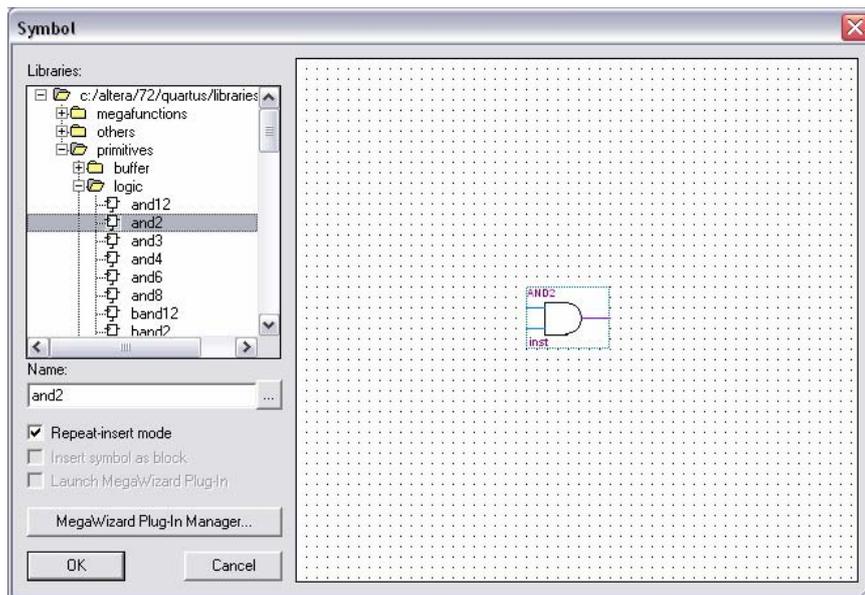


Figure 13. Choose a symbol from the library.

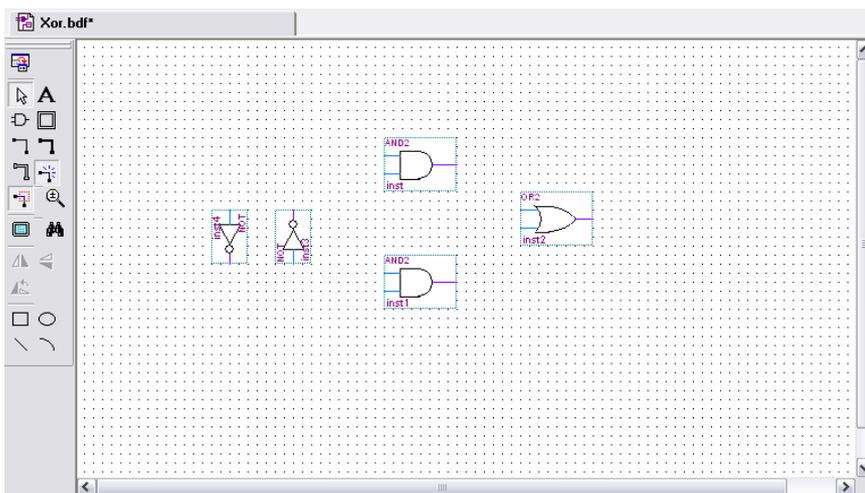


Figure 14. Import the gate symbols into the Graphic Editor window.

### 3.2 Importing Input and Output Symbols

Having entered the logic-gate symbols, it is now necessary to enter the symbols that represent the input and output ports of the circuit. Use the same procedure as for importing the gates, but choose the port symbols from the library primitives/pin. Import two instances of the input port and one instance of the output port, to obtain the image in Figure 15.

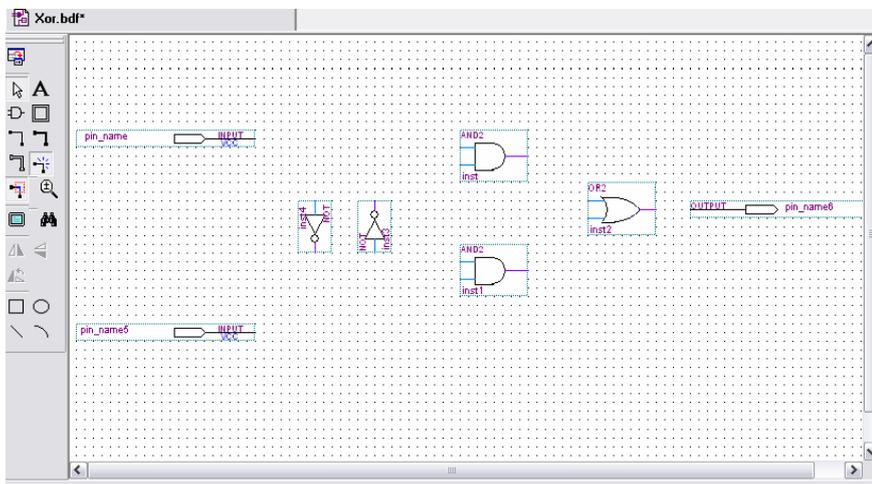


Figure 15. Import the input and output pins.

**Assign names** to the input and output symbols as follows. Make sure nothing is selected by clicking on an empty spot in the Graphic Editor window. **Point** to the word `pin_name` on the top input symbol and **double-click** the mouse. The dialog box in Figure 16 will appear. **Type the pin name**, `x1`, and click **OK**. Similarly, assign the name `x2` to the other input and `f` to the output. Alternatively, it is possible to change the name of an element by selecting it first, and then double-clicking on the name and typing a new one directly.

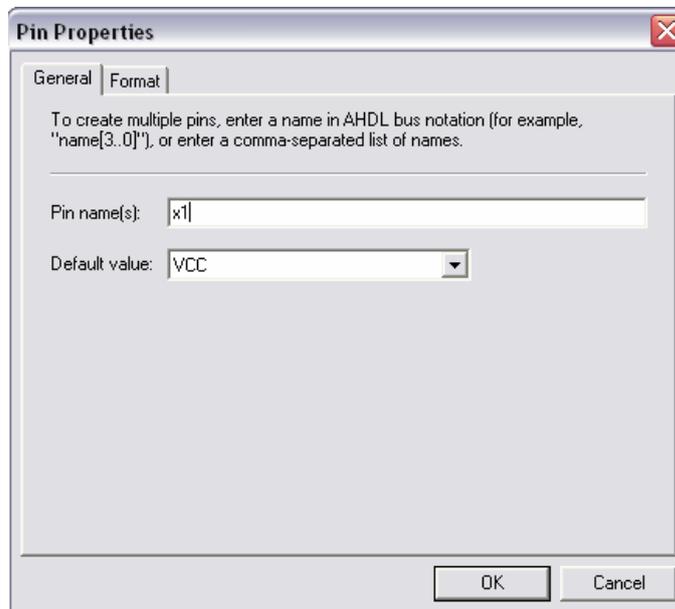


Figure 16. Naming of a pin.

### 3.3 Connecting Nodes with Wires

The symbols in the diagram have to be connected by drawing lines (wires). **Click on the icon** in the toolbar to activate the Orthogonal Node Tool. **Position the mouse** pointer over the right edge of the `x1` input pin. **Click and hold** the mouse button and **drag** the mouse to the right until the drawn line reaches the pinstub on the top input of the AND gate. **Release** the mouse button, which leaves the line connecting the two pinstubs. Next, draw a wire from the input pinstub of the leftmost NOT gate to touch the wire that was drawn above it. Note that a dot will appear indicating a connection between the two wires.

Use the same procedure to draw the remaining wires in the circuit. If a mistake is made, a wire can be selected by clicking on it, and removed by pressing the Delete key on the keyboard. Upon completing the diagram, click on the icon , to activate the Selection Tool. Now, changes in the appearance of the diagram can be made by selecting a particular symbol or wire and either moving it to a different location or deleting it. The final diagram is shown in Figure 17; **save it**.

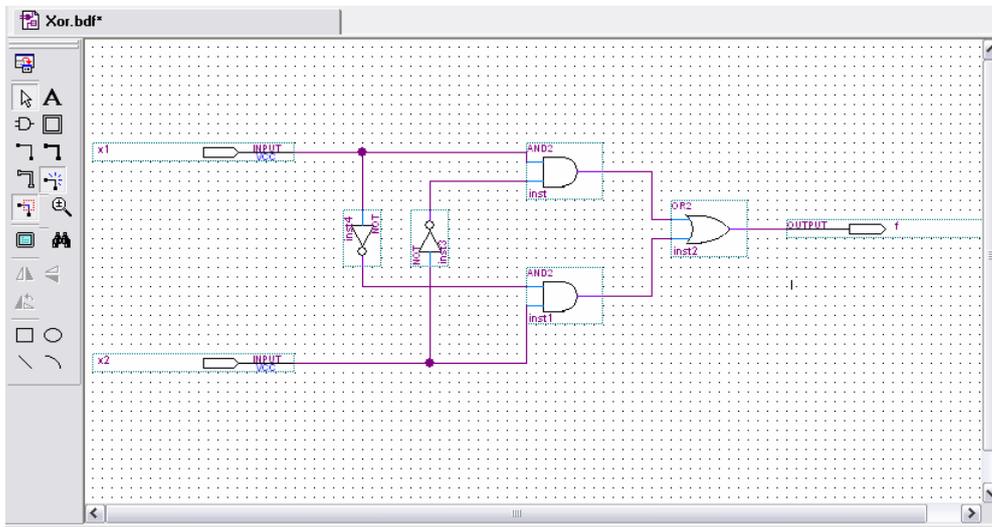


Figure 17. The completed schematic diagram.

## 4- Compiling the Designed Circuit

The entered schematic diagram file, Xor.bdf, is processed by several Quartus II tools that analyze the file, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the Compiler.

Run the Compiler by selecting **Processing > Start Compilation**, or by clicking on the toolbar icon that looks like a **purple triangle**. As the compilation moves through various stages, its progress is reported in a window on the left side of the Quartus II display. Successful (or unsuccessful) compilation is indicated in a pop-up box.

Acknowledge it by clicking **OK**, which leads to the Quartus II display in Figure 18. In the message window, at the bottom of the figure, various messages are displayed. In case of errors, there will be appropriate messages given.

When the compilation is finished, a compilation report is produced. A window showing this report is opened automatically, as seen in Figure 18. The window can be resized, maximized, or closed in the normal way, and it can be opened at any time either by selecting **Processing > Compilation Report** or by clicking on the icon.

The report includes a number of sections listed on the left side of its window. Figure 18 displays the Compiler Flow Summary section, which indicates that only one logic element and three pins are needed to implement this tiny circuit on the selected FPGA chip.

Flow Summary	
Flow Status	Successful - Sun Jan 24 11:20:25 2010
Quartus II Version	7.2 Build 151 03/26/2007 SJ Web Edition
Revision Name	Xor1
Top-level Entity Name	Xor1
Family	Cyclone II
Device	EP2C70F696C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	1 / 68,416 (< 1 %)
Total combinational functions	1 / 68,416 (< 1 %)
Dedicated logic registers	0 / 68,416 (0 %)
Total registers	0
Total pins	3 / 622 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 1,152,000 (0 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

```

Info: Running Quartus II Classic Timing Analyzer
Info: Command: quartus_tan --read_settings_files=off --write_settings_files=off Xor1 -c Xor1 --timing_analysis_only
Info: Longest tpd from source pin "x1" to destination pin "r" is 12.189 ns
Info: Quartus II Classic Timing Analyzer was successful. 0 errors, 0 warnings
Info: Quartus II Full Compilation was successful. 0 errors, 6 warnings
  
```

Figure 18. Display after a successful compilation.

In the case of unsuccessful compilation, Figure 19 shows the compilation report (Flow Summary).

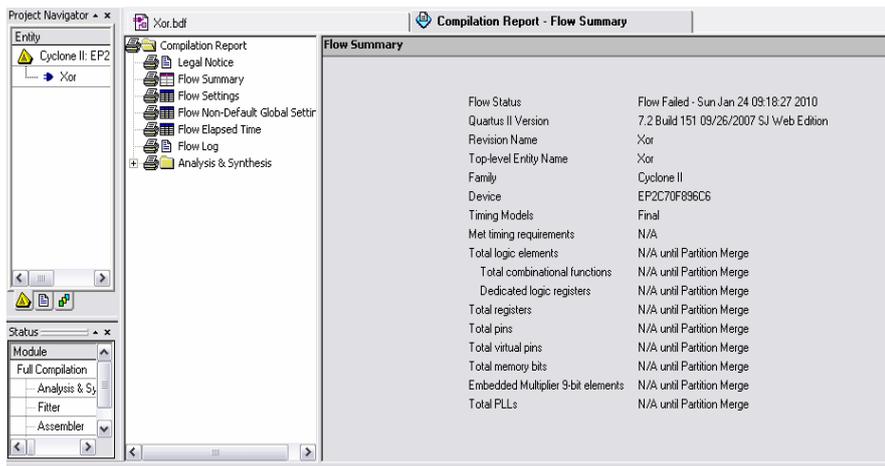


Figure 19. Compilation report for the failed design.

In the message tab, all errors will be shown, see Figure 20

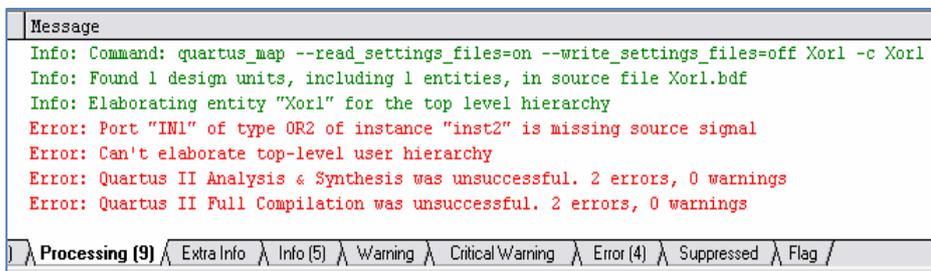


Figure 20. Error messages.

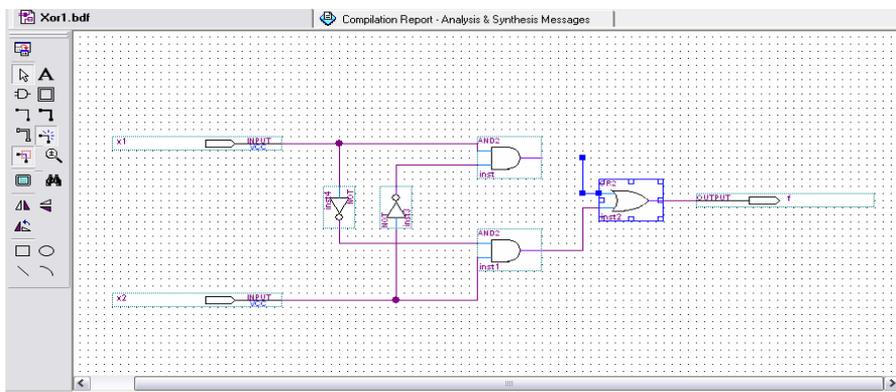


Figure 21. Identifying the location of the error.

After correcting all errors, recompile the circuit.

## 5- Pin Assignment

During the compilation above, the Quartus II Compiler was free to choose any pins on the selected FPGA to serve as inputs and outputs. However, the DE2 board has hardwired connections between the FPGA pins and the other components on the board. We will use two toggle switches, labeled **SW10** and **SW11**, to provide the external inputs, x1 and x2, to our example circuit. These switches are connected to the FPGA pins **W5** and **V10**, respectively. We will connect the output f to the red light-emitting diode labeled **LED10**, which is hardwired to the FPGA pin **AC13**.

Pin assignments are made by using the Assignment Editor. Select **Assignments > Assignment Editor** to reach the window in Figure 22.

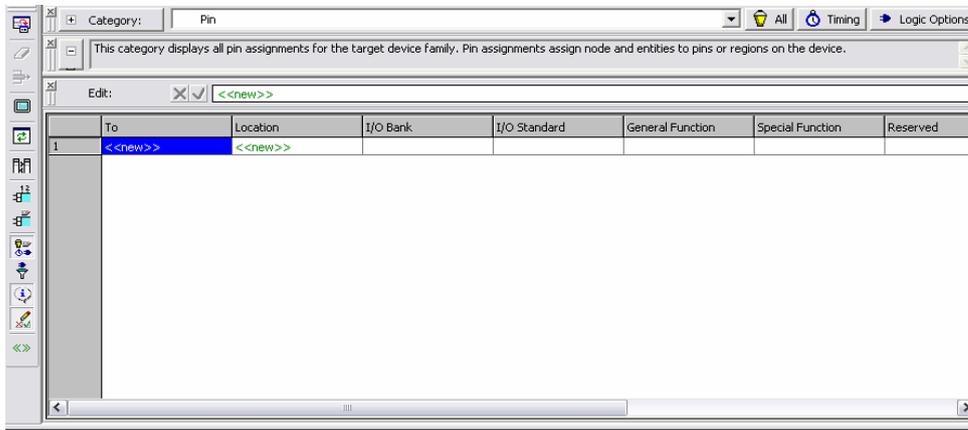


Figure 22. The Assignment Editor window.

Under **Category** select **Pin**. **Double-click on the entry <<new>>** which is highlighted in blue in the column labeled **To**. The drop-down menu in Figure 23 will appear. **Click on x1** as the first pin to be assigned; this will enter x1 in the displayed table. Follow this by **double-clicking** on the box to the right of this new x1 entry, in the column labeled **Location**. Now, the drop-down menu in Figure 24 appears.



Figure 23. The drop-down menu displays the input and output names.

Scroll down and select **PIN\_W5**. Instead of scrolling down the menu to find the desired pin, you can just **type the name of the pin (W5)** in the Location box. Use the same procedure to assign input x2 to pin V10 and output f to pin AC13, which results in the image in Figure 25.

	To	Location	I/O Bank	I/O Standard	General Function	Special Func
1	x1	PIN_W5		3.3-V LVTTTL		
2	<<new>>	PIN_W5	I/O Bank 1	Row I/O		
		PIN_W6	I/O Bank 1	Row I/O		
		PIN_W7	I/O Bank 1	Row I/O	LVDS17n	
		PIN_W8	I/O Bank 1	Row I/O	LVDS17p, CDPCLK1/DQ53L/CQ3L#	
		PIN_W9	I/O Bank 1	Row I/O	LVDS18p	
		PIN_W10	I/O Bank 1	Row I/O	LVDS18n	
		PIN_W21	I/O Bank 6	Row I/O	LVDS177p	
		PIN_W22	I/O Bank 6	Row I/O	LVDS177n	
		PIN_W23	I/O Bank 6	Row I/O	LVDS179n	
		PIN_W24	I/O Bank 6	Row I/O	LVDS179p	
		PIN_W25	I/O Bank 6	Row I/O	LVDS175n	
		PIN_W26	I/O Bank 6	Row I/O		
		PIN_W27	I/O Bank 6	Row I/O	LVDS171n	
		PIN_W28	I/O Bank 6	Row I/O	LVDS171p	
		PIN_W29	I/O Bank 6	Row I/O	LVDS169n	

Figure 24. The available pins.

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
1	x1	PIN_W5	1	3.3-V LVTTTL	Row I/O		
2	x2	PIN_V10	1	3.3-V LVTTTL	Row I/O	LVDS23p	
3	f	PIN_AC13	8	3.3-V LVTTTL	Column I/O		
4	<<new>>	<<new>>					

Figure 25. The complete assignment.

To save the assignments made, choose **File > Save**. You can also simply close the Assignment Editor window, in which case a pop-up box will ask if you want to save the changes to assignments; click **Yes**. **Recompile the circuit**, so that it will be compiled with the correct pin assignments.

## 6- Simulating the Designed Circuit

Before implementing the designed circuit in the FPGA chip on the DE2 board, it is prudent to simulate it to ascertain its correctness. Quartus II software includes a simulation tool that can be used to simulate the behavior of a designed circuit. Before the circuit can be simulated, it is necessary to create the desired waveforms, called **test vectors**, to represent the input signals. It is also necessary to specify which outputs, as well as possible internal points in the circuit, the designer wishes to observe. The simulator applies the test vectors to a model of the implemented circuit and determines the expected response. We will use the Quartus II Waveform Editor to draw the test vectors, as follows:

1. Open the Waveform Editor window by selecting **File > New>Other Files** tab, which gives the window shown in Figure 26. Choose **Vector Waveform File** and click **OK**.

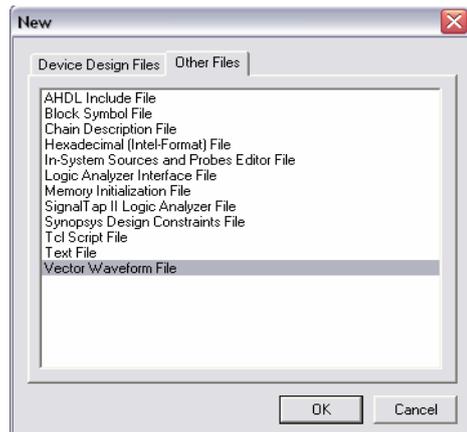


Figure 26. Need to prepare a new file.

2. The Waveform Editor window is depicted in Figure 27. **Save the file** under the name Xor.vwf; note that this changes the name in the displayed window. Set the desired simulation to run from 0 to 200 ns by selecting **Edit > End Time** and entering 200 ns in the dialog box that pops up. Selecting **View > Fit in Window** displays the entire simulation range of 0 to 200 ns in the window, as shown in Figure 28. You may wish to resize the window to its maximum size.

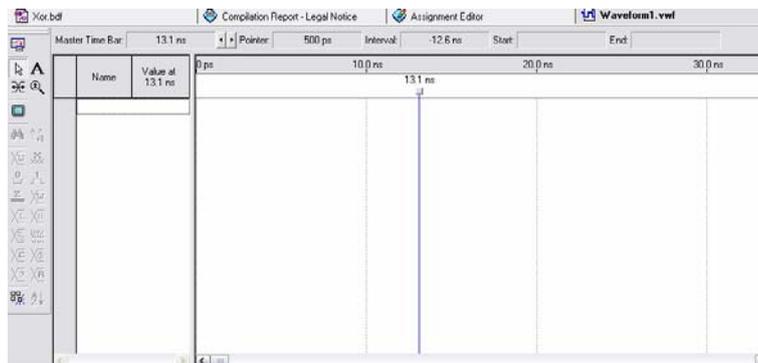


Figure 27. The Waveform Editor window.

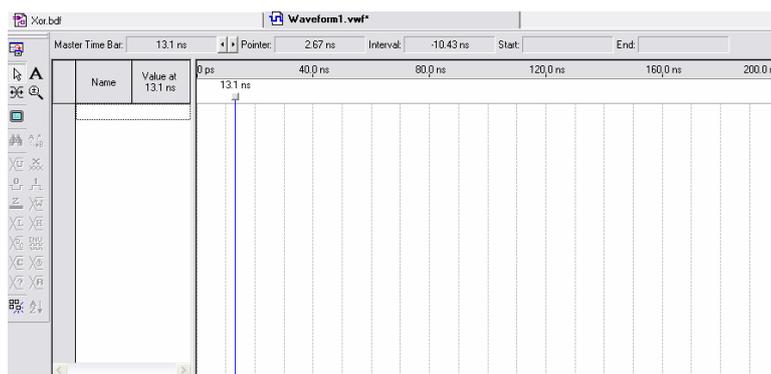


Figure 28. The augmented Waveform Editor window.

3. Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert > Insert Node or Bus** to open the window in Figure 29.

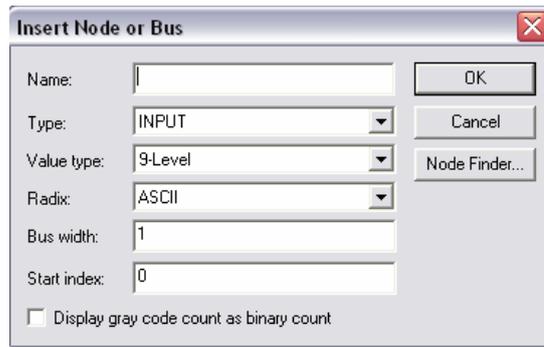


Figure 29. The Insert Node or Bus dialogue.

It is possible to type the name of a signal (pin) into the Name box, but it is easier to click on the **Node Finder** button to open the window in Figure 30. The Node Finder utility has a filter used to indicate what type of nodes are to be found. Since we are interested in input and output pins, **set the filter to Pins: all**. Click the **List** button to find the input and output nodes as indicated on the left side of the figure.

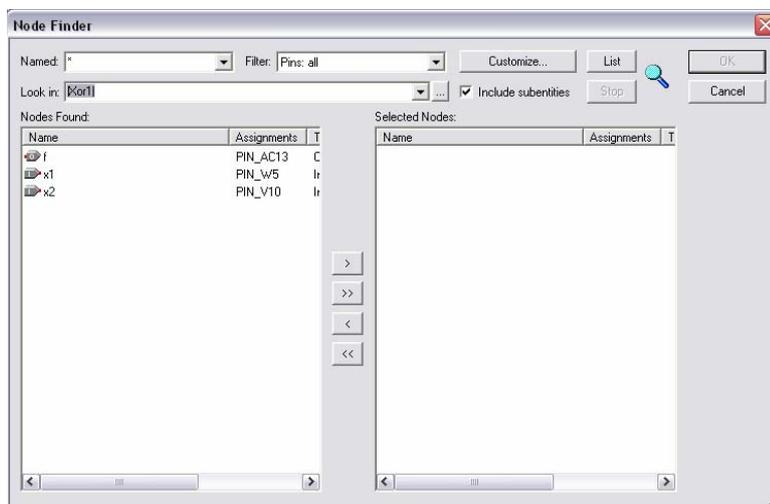


Figure 30. Selecting nodes to insert into the Waveform Editor.

Click on the x1 signal in the Nodes Found box in Figure 30, and then **click the > sign** to add it to the Selected Nodes box on the right side of the figure. Do the same for x2 and f. Click **OK** to close the Node Finder window, and then click **OK** in the window of Figure 29. This leaves a fully displayed Waveform Editor window, as shown in Figure 31. If you did not select the nodes in the same order as displayed in Figure 31, it is possible to rearrange them. To move a waveform up or down in the Waveform Editor window, click on the node name (in the Name column) and release the mouse button. The waveform is now highlighted to show the selection. Click again on the waveform and drag it up or down in the Waveform Editor.

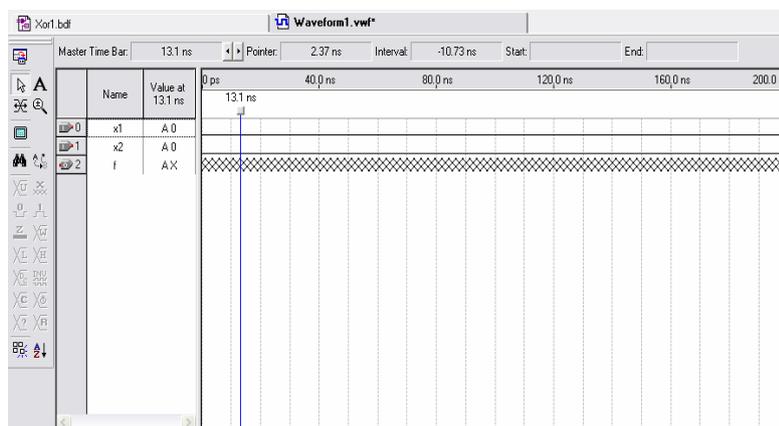


Figure 31. The nodes needed for simulation.

4. We will now specify the logic values to be used for the input signals x1 and x2 during simulation. The logic values at the output f will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides a drawing feature that snaps on these lines (which can otherwise be invoked by choosing **View > Snap to Grid**). Observe also a solid vertical line, which can be moved by pointing to its top and dragging it horizontally. This reference line is used in analyzing the timing of a circuit; move it to the time = 0 position. The waveforms can be drawn using the **Selection Tool**, which is activated by selecting the icon  in the toolbar, or the **Waveform Editing Tool**, which is activated by the icon .

To simulate the behavior of a large circuit, it is necessary to apply a sufficient number of input valuations and observe the expected values of the outputs. In a large circuit the number of possible input valuations may be huge, so in practice we choose a relatively small (but representative) sample of these input valuations.

However, for our tiny circuit we can simulate all different combinations (00, 01, 10, 11) given in Figure 9. We will use four 50-ns time intervals to apply the four test vectors. We can generate the desired input waveforms as follows. **Click on the waveform name for the x1 node.** Once a waveform is selected, the editing commands in the Waveform Editor can be used to draw the desired waveforms. Commands are available for setting a selected signal to 0, 1, unknown (X), high impedance (Z), don't care (DC), inverting its existing value (INV), or defining a clock waveform. Each command can be activated by using the **Edit > Value** command or via the toolbar for the Waveform Editor. The Edit menu can also be opened by **right-clicking on a waveform name.**

Set x1 to 0 in the time interval 0 to 100 ns, which is probably already set by default. Next, set x1 to 1 in the time interval 100 to 200 ns. Do this by **pressing the mouse at the start of the interval and dragging it to its end**, which highlights the selected interval, and **choosing the logic value 1 in the toolbar.** Make x2 = 1 from 50 to 100 ns and also from 150 to 200 ns, which corresponds to the truth table in Figure 9. This should produce the image in Figure 32. Observe that the output f is displayed as having an unknown value at this time, which is indicated by a hashed pattern; its value will be determined during simulation. **Save the file as Xor1.vwf.**

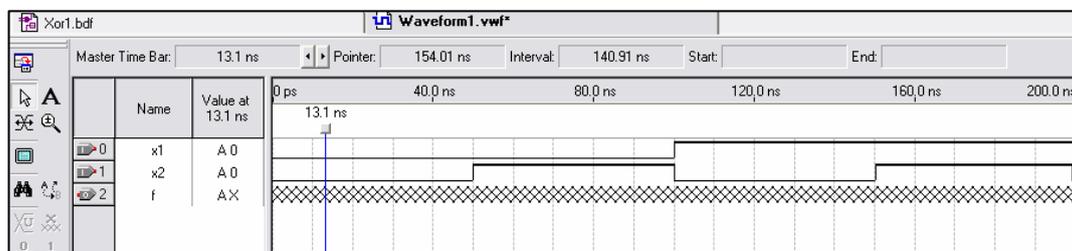


Figure 32. Setting of test values.

## 6.1 Performing the Simulation

A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and interconnection wires in the FPGA are perfect, thus causing no delay in propagation of signals through the circuit. This is called **Functional simulation**. A more complex alternative is to take all propagation delays into account, which leads to **Timing simulation**. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed. This takes much less time, because the simulation can be performed simply by using the logic expressions that define the circuit.

### 6.1.1 Functional Simulation

To perform the functional simulation select **Assignments > Settings** to open the Settings window, on the left side of this window click on **Simulator Settings** to display the window in Figure 33, **choose Functional as the simulation mode, choose Xor1.vwf as the simulation input**, and click **OK**. The Quartus II simulator takes the inputs and generates the outputs defined in the Xor.vwf file.

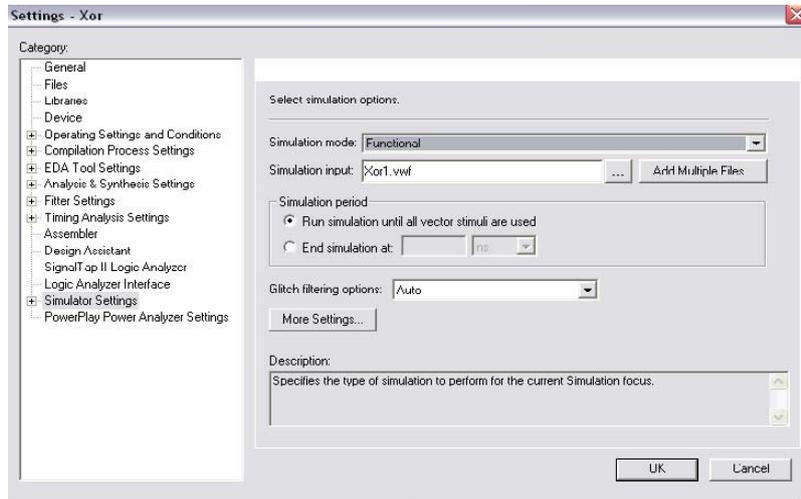


Figure 33. Specifying the simulation mode.

Before running the functional simulation it is necessary to create the required netlist; select **Processing > Generate Functional Simulation Netlist**. A simulation run is started by **Processing > Start Simulation**. At the end of the simulation, Quartus II software indicates its Successful completion and displays a **Simulation Report** illustrated in Figure 34. If your report window does not show the entire simulation time range, **click on the report window** to select it and choose **View > Fit in Window**. Observe that the output f is as specified in the truth table of Figure 9.

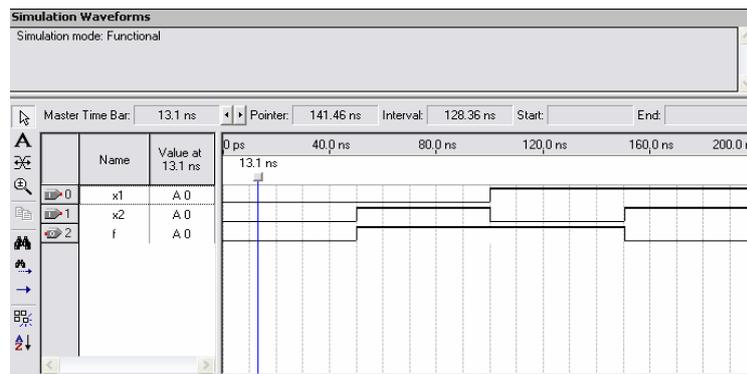


Figure 34. The result of functional simulation.

### 6.1.2 Timing Simulation

Having ascertained that the designed circuit is functionally correct, we should now perform the timing simulation to see how it will behave when it is actually implemented in the chosen FPGA device. Select **Assignments > Settings > Simulator Settings** to get to the window in Figure 33, **choose Timing as the simulation mode, choose Xor1.vwf as the simulation input**, and click **OK**. Run the simulator, which should produce the waveforms in Figure 35. Observe that there is a delay of about 6 ns in producing a change in the signal f from the time when the input signals, x1 and x2, change their values. This delay is due to the propagation delays in the logic element and the wires in the FPGA device.

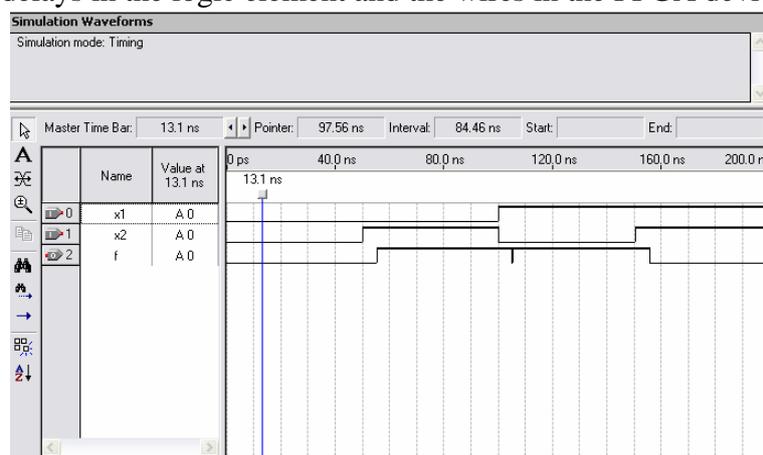


Figure 35. The result of timing simulation.

## 7- Programming and Configuring the FPGA Device

The FPGA device must be programmed and configured to implement the designed circuit. The required configuration file is generated by the Quartus II Compiler's Assembler module. Altera's DE2 board allows the configuration to be done in two different ways, known as JTAG and AS modes. The configuration data is transferred from the host computer (which runs the Quartus II software) to the board by means of a cable that connects a USB port on the host computer to the leftmost USB connector on the board. To use this connection, it is necessary to have the USB-Blaster driver installed. Before using the board, make sure that the USB cable is properly connected and turn on the power supply switch on the board.

In the JTAG mode, the configuration data is loaded directly into the FPGA device. The acronym JTAG stands for Joint Test Action Group. This group defined a simple way for testing digital circuits and loading data into them, which became an IEEE standard. If the FPGA is configured in this manner, it will retain its configuration as long as the power remains turned on. The configuration information is lost when the power is turned off. The second possibility is to use the Active Serial (AS) mode. In this case, a configuration device that includes some flash memory is used to store the configuration data. Quartus II software places the configuration data into the configuration device on the DE2 board. Then, this data is loaded into the FPGA upon power-up or reconfiguration.

Thus, the FPGA need not be configured by the Quartus II software if the power is turned off and on. *The choice between the two modes is made by the RUN/PROG switch on the DE2 board. The RUN position selects the JTAG mode, while the PROG position selects the AS mode.*

### 7.1 JTAG Programming

The programming and configuration task is performed as follows. **Flip the RUN/PROG switch into the RUN position.** Select **Tools > Programmer** to reach the window in Figure 36. Here it is necessary to specify the programming hardware and the mode that should be used. If not already chosen by default, **select JTAG in the Mode box.**

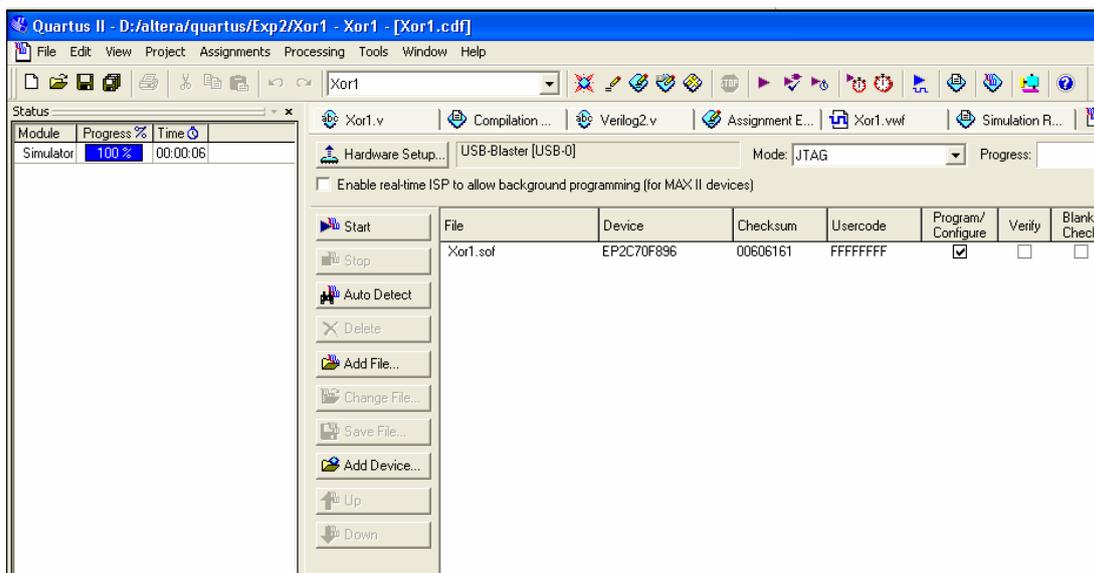


Figure 36. The Programmer window.

Also, if the USB-Blaster is not chosen by default, **press the Hardware Setup...** button and **select the USB-Blaster** in the window that pops up, as shown in Figure 37.

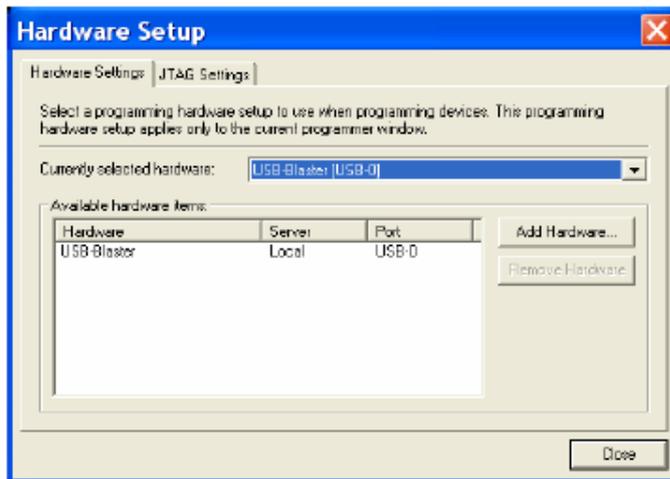


Figure 37. The Hardware Setup window.

In Figure 36, observe that the configuration file Xor1.sof is listed in the window. If the file is not already listed, then click **Add File** and select it. This is a binary file produced by the Compiler's Assembler module, which contains the data needed to configure the FPGA device. The extension .sof stands for SRAM Object File. Note also that the device selected is **EP2C70F896C6**, which is the FPGA device used on the DE2 board. **Click on the Program/Configure check box**. Now, **press Start button**. A LED on the board will light up when the configuration data has been downloaded successfully. If you see an error reported by Quartus II software indicating that programming failed, check to ensure that the board is properly powered on.



## Introduction:

Verilog HDL is a hardware description language used to design electronic systems. Verilog HDL allows designers to design at various levels of abstraction. It is the most widely used HDL with a user community of more than 50,000 active designers.

This tutorial shows how the Quartus II software can be used to design and implement a circuit specified by using the Verilog hardware description language.

## Objectives:

- Creating a project using Quartus II software.
- Design entry using Verilog code.
- Assigning the circuit inputs and outputs to specific pins on the FPGA.
- Simulating the designed circuit.
- Programming and configuring the FPGA device.

## ➤ What is Verilog?

Verilog is one of the two major Hardware Description Languages (HDL) used by hardware designers in industry and academia. VHDL is the other one.

The Verilog language describes a digital system as a set of modules. Each of these modules has input(s) and output(s). Usually we place one module per file but that is not a requirement.

**Note:** Verilog is **case sensitive**

## The structure of a module is the following:

```
module <module name> (<port list>);  
<declares>  
<module items>  
endmodule
```

- The **<module name>** is an identifier that uniquely names the module.
- The **<port list>** is a list of input and output ports.
- The **<declares>** section specifies data objects as inputs, outputs, or wires.
- The **<module items>** may be assignments or instances of modules.

Modules can represent pieces of hardware ranging from simple gates to complete systems. Modules can either be specified **behaviorally or structurally** (or a combination of the two).

- A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs, e. g., **ifs, whiles, assignment statements**.

## **Example:**

```
// Behavioral Model of a Nand gate  
module NAND(in1,in2, out);  
input in1, in2;  
output out;  
// continuous assign statement  
assign out = ~(in1 & in2);  
endmodule
```

**In the above example:**

```
<module name>: NAND
<port list>: in1, in2, out
<Declares>: input in1, in2;
            output out;
<module items>: assign out = ~(in1 & in2);
```

**Note:** The continuous assignment **assign** continuously watches for changes to variables in its right hand side and whenever that happened the right hand side is re-evaluated and the result immediately propagated to the left hand side (**out**).

- **Structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules.

Here is a structural specification of a module **AND** obtained by connecting the output of one **NAND** to both inputs of another one.

```
module AND(in1, in2, out);
// Structural model of AND gate from two NANDS
input in1, in2;
output out;
wire w1;
// two instantiations of the module NAND
NAND NAND1 (in1, in2, w1);
NAND NAND2 (w1, w1, out);
endmodule
```

This module has two instances of the **NAND** module called **NAND1** and **NAND2** connected together by an internal wire **w1**.

The general form to invoke an instance of a module is:

```
<module name> <instance name> (<port list>);
```

## ➤ Quartus II Introduction Using Verilog Design:

The following example makes use of the Verilog design entry method, in which the user specifies the desired circuit in the Verilog hardware description language.

### 1-Getting Started:

Follow the steps in the previous experiment to create new project, and name it Xor1.

### 2-Using the Quartus II Text Editor:

1-Select **File > New** to get the window in Figure 1, then choose **Verilog HDL File** and click **OK**. This opens the Text Editor window in Figure 2.

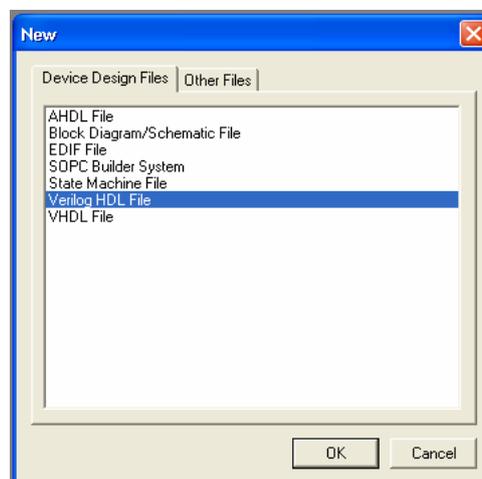


Figure (1)

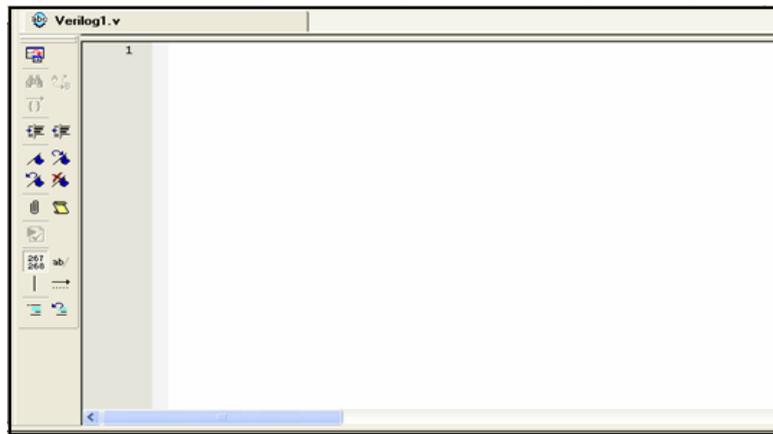


Figure (2)

2- Specify the name for the file that will be created. Select **File > Save As** to open the pop-up box depicted in Figure 3.

In the box labeled **Save as type** choose **Verilog HDL File**. In the box labeled **File name type** **Xor1**. Put a checkmark in the box **Add file to current project**. Click **Save**, which puts the file into the directory Exp2 and leads to the Text Editor window shown in Figure (4).

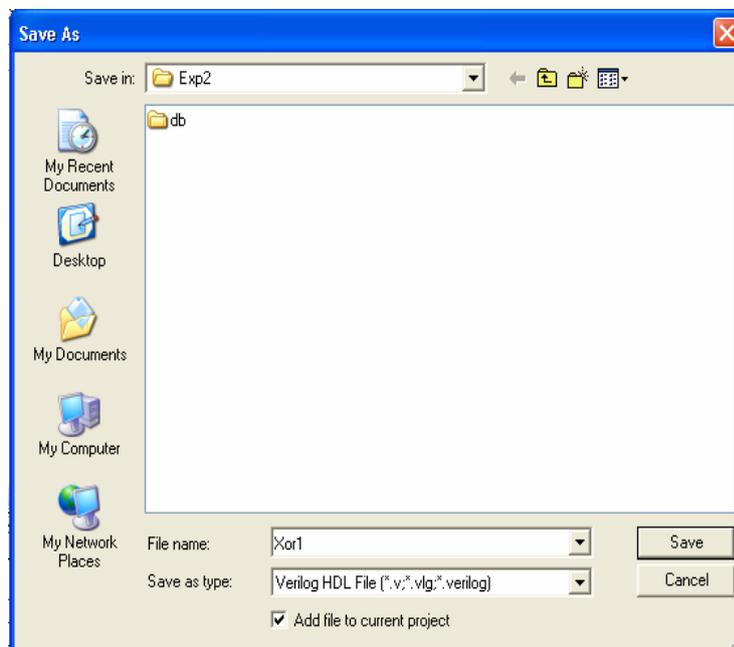


Figure (3)

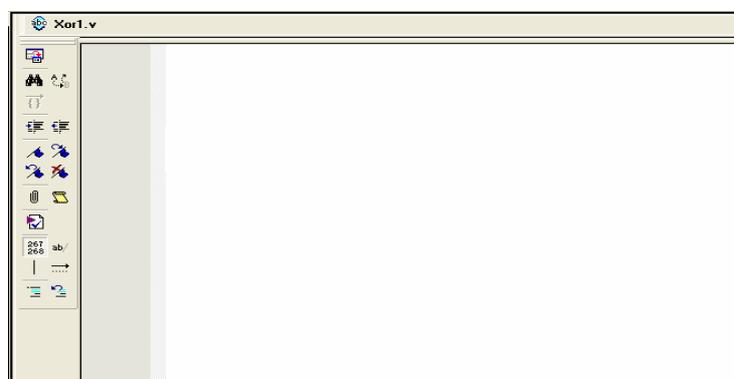


Figure (4)

3- Enter the Verilog code as shown in Figure 5. Then save the file by choosing **File > Save**, or by typing the shortcut Ctrl-s.

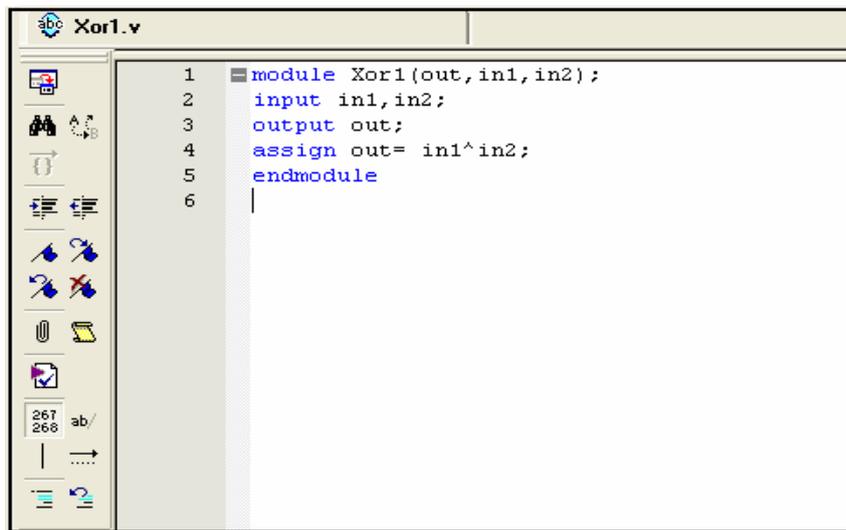


Figure (5)

### Using Verilog Templates

The syntax of Verilog code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of Verilog templates. The templates provide examples of various types of Verilog statements, such as a module declaration, an always block, and assignment statements. It is worthwhile to browse through the templates by selecting **Edit > Insert Template > Verilog HDL** to become familiar with this resource.

### 3-Compiling the Designed Circuit:

Refer to Experiment 2.

### Errors

Quartus II software displays messages produced during compilation in the Messages window. If the Verilog design file is correct, one of the messages will state that the compilation was successful and that there are no errors.

If the Compiler does not report zero errors, then there is at least one mistake in the Verilog code. In this case a message corresponding to each error found will be displayed in the Messages window. Double-clicking on an error message will highlight the offending statement in the Verilog code in the Text Editor window. Similarly, the Compiler may display some warning messages. Their details can be explored in the same way as in the case of error messages. The user can obtain more information about a specific error or warning message by selecting the message and pressing the F1 function key.

To see the effect of an error, open the file Xor1.v. Remove the semicolon in the assign statement, illustrating a typographical error that is easily made. Compile the erroneous design file by clicking on the icon. A pop-up box will ask if the changes made to the Xor1.v file should be saved; click Yes. After trying to compile the circuit, Quartus II software will display a pop-up box indicating that the compilation was not successful. Acknowledge it by clicking OK. The compilation report summary, now confirms the failed result. Expand the Analysis & Synthesis part of the report and then select Messages to have the messages displayed as shown in Figure 6.

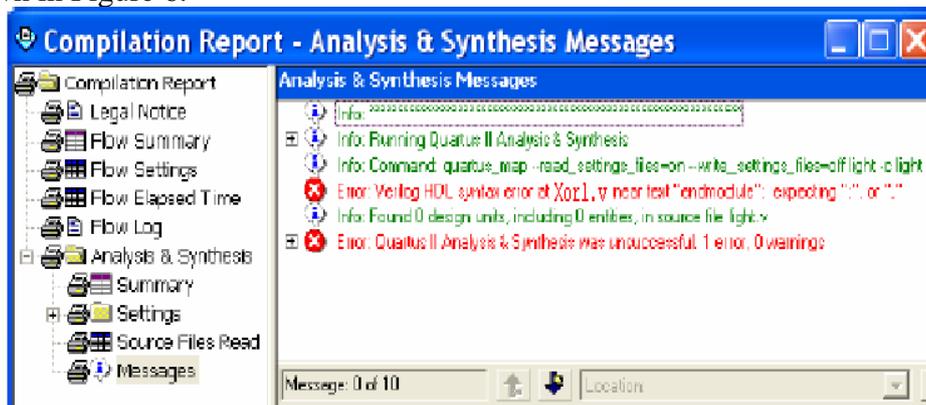


Figure 6

Double-click on the first error message. Quartus II software responds by opening the Xor1.v file and highlighting the statement which is affected by the error. Correct the error and recompile.

#### **4-Pin Assignment:**

Refer to Experiment 2.

#### **5- Simulating the Designed Circuit**

Refer to Experiment 2.

#### **6 -Programming and Configuring the FPGA Device**

Refer to Experiment 2.

#### **Note:**

*If you want to implement the Xor module using structural modeling do the following:*

1. Write the primitive needed modules (AND, OR, INV) in a separate file name it (lib.v)

```
module ANDGATE(in1, in2, out);
input in1, in2;
output out;
assign out= in1&in2;
endmodule
```

```
module ORGATE(in1, in2, out);
input in1, in2;
output out;
assign out= in1|in2;
endmodule
```

```
module INVGATE(in1, out);
input in1;
output out;
assign out= ~in1;
endmodule
```

2. Write Xor module in a separate file, name it (Xor1.v)

```
module Xor1(in1, in2, out);
input in1, in2;
output out;
wire w1,w2,w3,w4;

INVGATE inv1(in1,w1);           // w1 = ~in1
INVGATE inv2(in2,w2);           // w2 = ~in2

ANDGATE AND1(in1, w2, w3);       // w3 = in1&(~in2)
ANDGATE AND2(in2, w1, w4);       // w4 = in2&(~in1)

ORGATE or1(w3,w4,out);          // out = w3 | w4

endmodule
```

*Note that the Xor1 module should be the top level module, you can change top level module by **choosing project>set as top level.***

*If you need the file (lib.v) in another project you don't have to create it again, just make add file in project creation (step3) in Experiment 2.*



## **Experiment 4**

### **Boolean Functions Implementation**

#### **Introduction:**

Generally you will find that the basic logic functions AND, OR, NAND, NOR, and NOT are not sufficient to implement complex digital logic functions. These gates are the basis for building more complex logic circuits that are constructed using various combinations of gates, which is known as Combinational Logic. Combinational logic requires the use of two or more gates to form a useful, complex function.

#### **Objectives:**

- Draw the schematic diagram of any function based on its equation, truth table or K-map and implement this function on the FPGA.
- Minimize a logic function to the minimum number of gates that will generate an equivalent function to reduce a circuit cost and size and increase its reliability.
- Design and build real functions on the FPGA.
- Use Verilog and schematic programming of QuartusII to implement logic functions.
- Derive all logic functions using NAND or NOR universal gate.

#### **Gate level Minimization:**

A Boolean expression is composed of variables and terms, the simplification of Boolean expressions can lead to more effective and efficient circuits that allow designers to make use of fewer components, thus reducing the cost of a particular system when implemented in practice.

Simplification can be achieved by a number of methods, two well known methods are:

- 1) Algebraic Manipulation of Boolean Expressions.
- 2) Karnaugh Maps .

##### **1) Algebraic Manipulation of Boolean Expressions:**

This is an approach where you can transform one Boolean expression into an equivalent expression by applying Boolean Theorems, so we cancel out redundant terms and variables from an SOP or POS expression.

It should be noted that there are no fixed rules that can be used to minimize a given expression. It is left to an individual's ability to apply Boolean Theorems in order to Minimize a function.

In Boolean algebra simplification, Terms can be factored out of expressions, and parenthesis can be added and removed to and from grouped terms as needed.

Here is the list of rules used for the Boolean expression simplification:

- **The Idempotent Laws**  
 $AA = A$   
 $A+A = A$
- **The Commutative Laws**  
 $AB = BA$   
 $A+B = B+A$

- **The Distributive Laws**

$$A(B+C) = AB+AC$$

$$A+BC = (A+B)(A+C)$$

- **The Identity Laws**

$$A.0 = 0$$

$$A.1 = A$$

$$A+0 = A$$

$$A+1 = 1$$

- **The Complement Laws**

$$AA' = 0$$

$$A+A' = 1$$

- **The Involution Law**

$$A'' = A$$

- **DeMorgan's Law**

$$(AB)' = A' + B'$$

$$(A+B)' = A' . B'$$

**Example:**

- Simplify:  $(A + C)(AD + AD') + AC + C$ :

<u>Expression</u>	<u>Rule(s) Used</u>
$(A + C)(AD + AD') + AC + C$	Original Expression
$(A + C)A(D + D') + AC + C$	Distributive.
$(A + C)A + AC + C$	Complement, Identity.
$A((A + C) + C) + C$	Commutative, Distributive.
$A(A + C) + C$	Associative, Idempotent.
$AA + AC + C$	Distributive.
$A + (A + 1)C$	Idempotent, Identity, Distributive.
$A + C$	Identity, twice.

## 2) Karnaugh Maps:

Simplification can be obtained by applying the postulates and theorems, but this can be time consuming and easily lead to human error when considering large equations, another method of simplification can be obtained using Karnaugh maps or K-maps.

- A Karnaugh Map or (K-map for short) is a graphical representation of the truth table of a logic function.
- The K-map for an n-input logic function is an array with  $2^n$  cells or squares, one for each input combination or minterm.
- The rows and columns are labeled so that the input combination for any cell is determined from the row and column headings.
- The K-map for a function is filled by putting a '1' in the square corresponding to a minterm, and '0' otherwise .
- Group or combine as many adjacent 1-cells as possible:
  - The larger the group is, the fewer the number of literals in the resulting term.
  - Each group of combined adjacent 1-cells must have a number of cells equal to powers of two: 1, 2, 4, 8, ...
  - Grouping 2 adjacent 1-cells eliminates 1 variable, grouping 4 1-cells eliminates 2 variables, grouping 8 1-cells eliminates 3 variables, and so on. In general, grouping  $2^n$  squares eliminates n variables.
  - Select as few groups as possible to cover all the 1-cells(minterms) of the function, The fewer the groups, the fewer the number of terms in the minimized function.

### Example:

Suppose you have the function  $F(x,y)$ , represented by the table below:

x	y	F
1	1	1
1	0	1
0	1	0
0	0	0

$$F = xy + xy'$$

For every row in the truth table there corresponds a square in the K-map.

For a two variable function there are 4 different possible choices. We represent this using the 4 squares of the K-map.

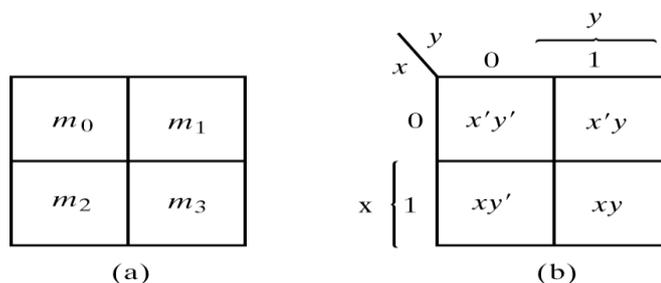


Figure1

Using a Karnaugh map, we can simplify this circuit as follows:

When 1s occur in two adjacent cells in the K-map, the terms represented by those cells can be combined into a product involving just one of the variables.

We can see from the K-map that the value of y doesn't affect the outcome of the function; in general, when there are 1s in two adjacent cells in the K-map, the minterms represented by those cells can be combined into a product involving just one of the variables.

So  $F=X$ .

We can verify this using laws of Boolean algebra:

$F= xy + xy' = x(y + y') = x(1) = x$  (using distribution, complement, and identity laws).

	y	$\bar{y}$
$\bar{x}$	0	0
x	1	1

## Universal gate:

NAND and NOR gates are sometimes referred to as universal gates, by utilizing a combination of NAND or NORs, all other logic gates can be formed so that any digital circuit can be designed by using only one type of gates either NAND or NOR gates.

Example:

- An inverter can be formed from a NAND as in figure below:

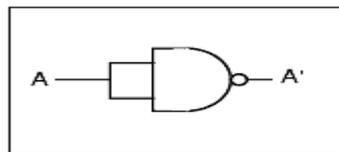


Figure 2

- AND gate can be formed from a NAND gates as in figure below:

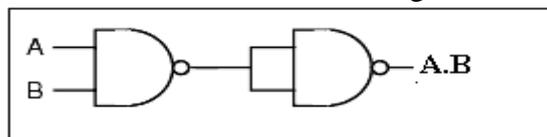


Figure 3

- OR gate can be formed from a NAND gates as in figure below:

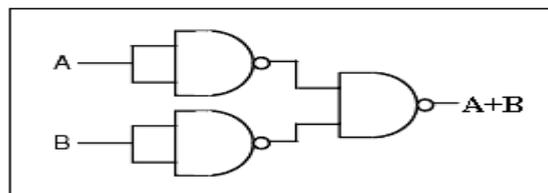


Figure 4

- NOR gate can be formed from a NAND gates as in figure below:

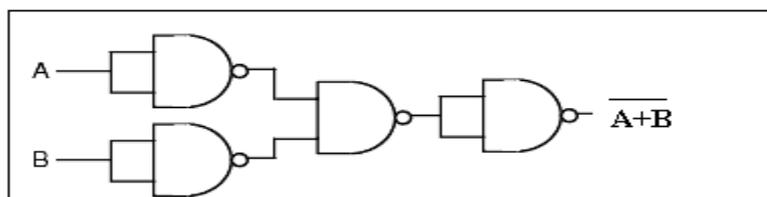


Figure 5

- XOR gate can be formed from a NAND gates as in figure below:

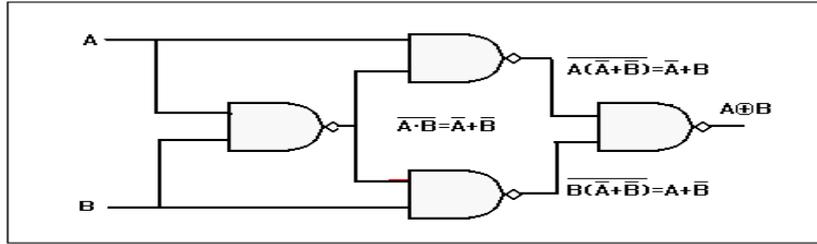


Figure 5

## Design procedure:

The design of combinational circuits starts from the specification of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained.

### **Combinational Logic Design Procedure:**

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are designed letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn

### **Example:**

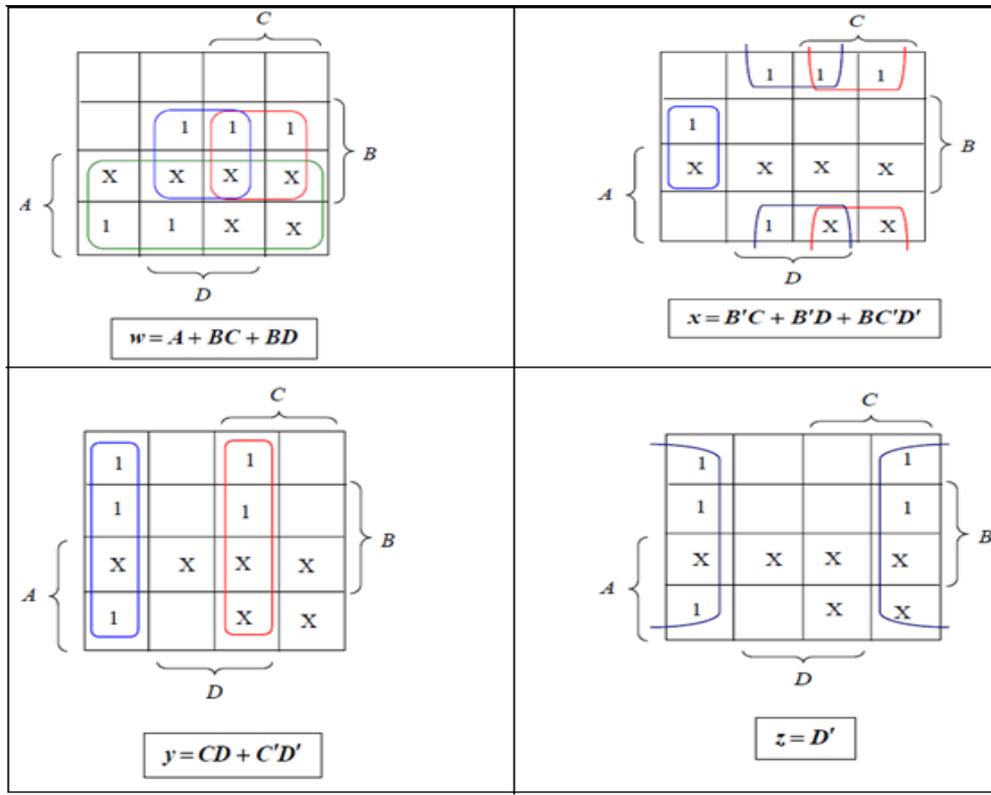
Design a combinational logic circuit that will convert from a BCD code to Excess-3 code.

- The number of available input variables is 4 inputs → A, B, C, and D.
- The number of the required output variables is 4 outputs → w, x, y, and z
- Truth Table:

Inputs (BCD Code)				Outputs(Excess-3)			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

The six unused combinations are considered don't care conditions. These correspond to 10, 11, 12, 13, 14, and 15.

- Simplification of the output functions is made using Karnaugh maps and making use of the don't care conditions.



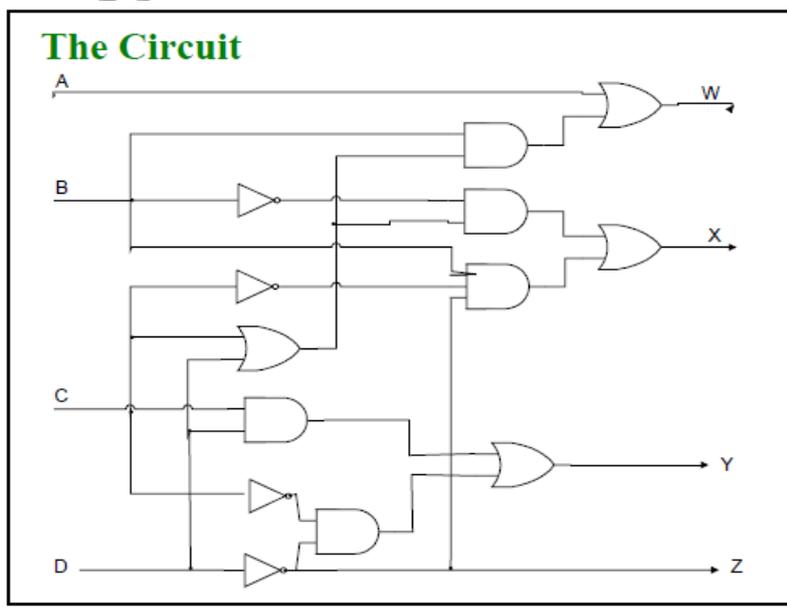
- The logic diagram is drawn according each of the following function as we Know.

$$W = A + BC + BD$$

$$X = B'C + B'D + BC'D'$$

$$Y = CD + C'D'$$

$$Z = D'$$





## Experiment 5

### Decoder/Encoder Implementation

#### Introduction:

Data communications between digital systems or computers are usually transmitted in some form of a code. A circuit that will convert a digital input into some form of a binary code is called an Encoder. A digital circuit that converts a binary code into a recognizable number or character is called a Decoder.

#### Objectives:

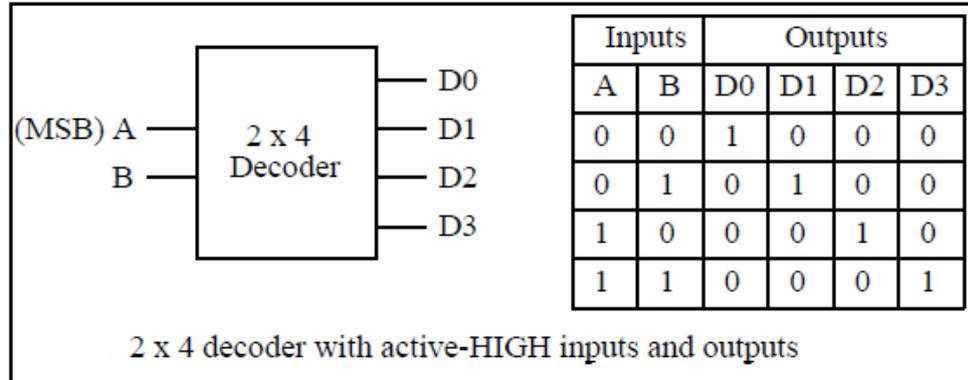
- Design, build, and test a variety of Decoders, Encoders.
- Demonstrate the operations and applications of Decoders, Encoders.
- Implement logic functions using Decoders.

#### Decoder:

A Decoder is a combinational circuit that converts binary information from  $n$  input lines to a maximum of  $2^n$  unique output lines.

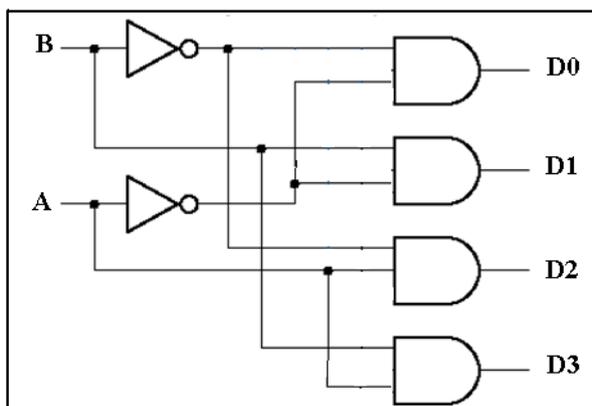
- A decoder has  $n$  inputs and  $m$  outputs, where  $m \leq 2^n$ , and is called  $n$ -to- $m$ -line decoder .
- each output represent one of the minterms of the  $n$  input variables for Active-high decoders, and represent one of the maxterms for active-low decoders .

The Figure below represents the block diagram and a truth table for a 2-line-to-4-line (or 2 x 4) decoder that has active-HIGH inputs and outputs.



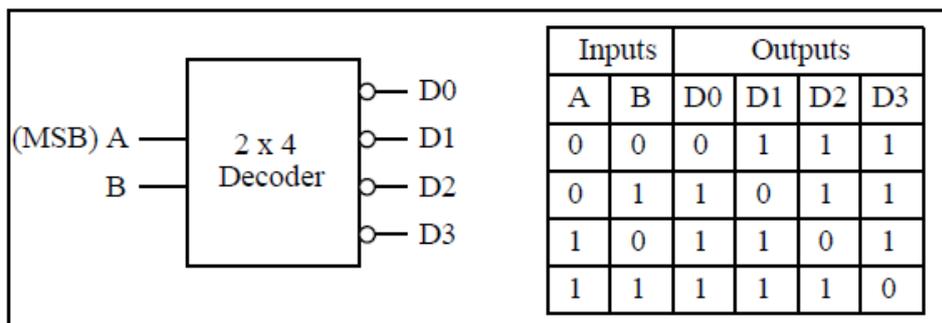
Figure(1)

Logic Diagram of 2 x 4 decoder with active-HIGH inputs and outputs:



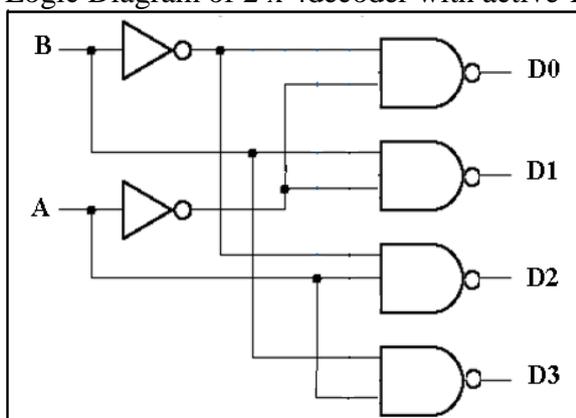
Figure(2)

Some decoders, have active-LOW outputs. Figure below shows a block diagram and a truth table for a 2 x 4 decoder with active-LOW outputs.



Figure(3)

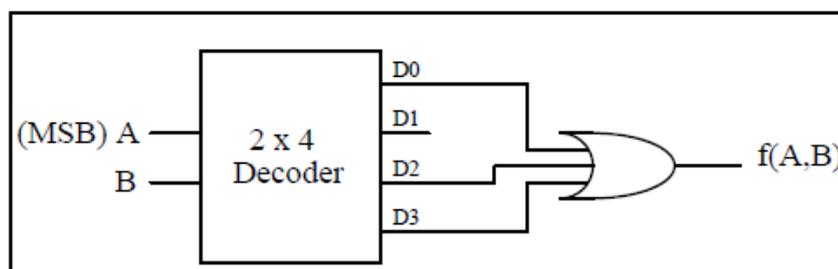
Logic Diagram of 2 x 4 decoder with active-LOW outputs:



Figure(4)

### ➤ Function Implementation using Decoder

As we mention above the outputs of the decoder correspond to minterms for the active high decoder. For example,  $D0 = m0 = A' B'$ , a combinational logic function that is expressed as a sum of minterms, therefore, can be implemented by summing decoder outputs. For example, if  $f(A,B) = \Sigma(0, 2, 3)$  then  $f(A,B) = D0 + D2 + D3$  so  $f$  can be implemented by the circuit shown in Figure below:



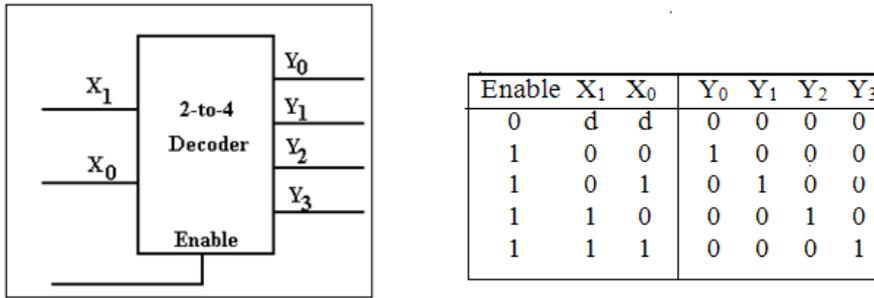
Figure(5)

### ➤ The Enable Input

Enable is an important input to the decoder chip. If the decoder enable signal is active high, then the decoder is active (enabled) when enable is 1 and not active (disabled) when enable = 0.

For an active high decoder that is enabled high we have the following:

Enable = 0    All outputs of the decoder are 0  
 Enable = 1    The selected output of the decoder is 1, all other outputs are 0.



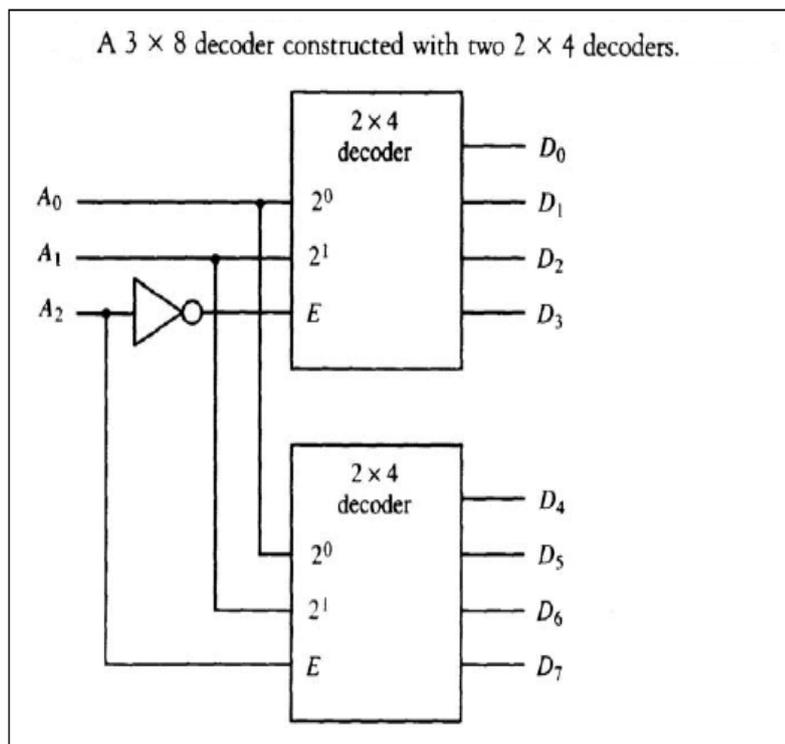
Figure(6)

If the decoder enable signal is active low, then the decoder is active (enabled) when enable is 0 and not active (disabled) when enable = 1.

### ➤ Decoder Expansion

It is possible to combine two or more decoders with enable inputs to form a larger decoder .

The enable inputs are a convenient feature for decoder expansion .

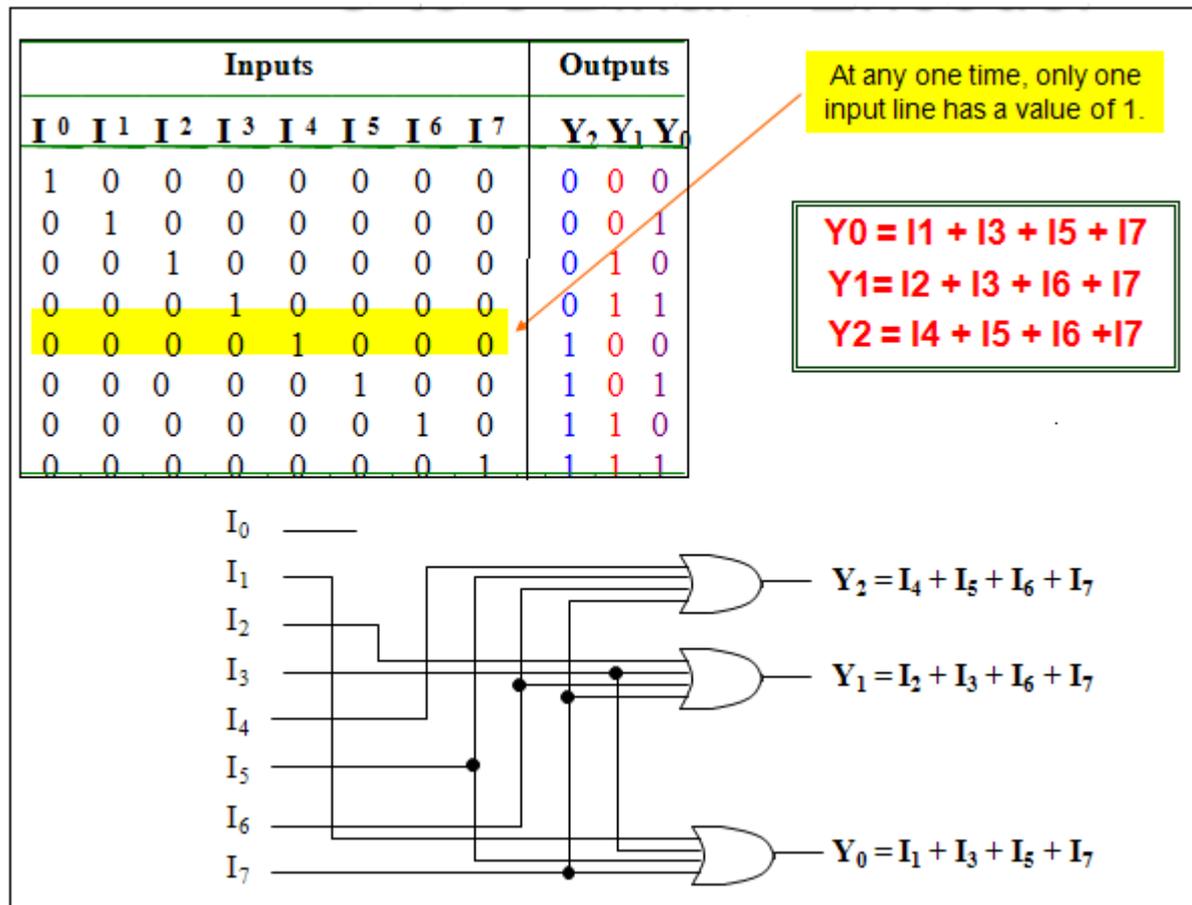


Figure(7)

## Encoder:

An encoder is a digital circuit that performs the inverse of a decoder, the encoder has  $2^n$  (or less) input lines and  $n$  output lines, the output lines generate the binary code corresponding to the input value.

Example : design 8-3 encoder.



Figure(8)

If we look carefully at the Encoder circuits that we got, we see that if more than two inputs are active simultaneously, the output is unpredictable or it is not what we expect it to be. This ambiguity is resolved if priority is established so that only one input is encoded, no matter how many inputs are active at a given point of time.

- **Priority Encoder :**

With a priority encoder, we may have more than one input with a value of 1. How do we can decide which input subscript to encode by assign a priority to each of the subscripts.

There are two common ways to do come up with a priority scheme:

- Larger subscripts have higher priorities. Thus, 0 has the lowest priority, and 7 has the highest.
- Smaller subscripts have higher priorities. Thus, 7 has the lowest priority, and 0 has the highest.

For now, let's assume that larger subscripts have higher priorities, then the following table represent 8-3 high priority encoder:

Inputs								Outputs		
I <sup>0</sup>	I <sup>1</sup>	I <sup>2</sup>	I <sup>3</sup>	I <sup>4</sup>	I <sup>5</sup>	I <sup>6</sup>	I <sup>7</sup>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>
0	0	0	0	0	0	0	0	x	x	x
x	x	x	x	x	x	x	1	1	1	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	1	0	0	0	0	0	1	1
x	x	1	0	0	0	0	0	0	1	0
x	1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0

So:

$$Y_0 = I_7 + I_5 I_6' + I_3 I_4' I_6' + I_1 I_2' I_4' I_6'$$

$$Y_1 = I_7 + I_6 + I_3 I_4' I_5' + I_2 I_4' I_5'$$

$$Y_2 = I_7 + I_6 + I_5 + I_4$$

- **The All-Zero Case**

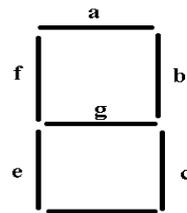
What do we do if all the inputs are 0? We might encode 000 as output, but that creates a problem. In particular, we can't distinguish between all 0's as inputs and having I<sup>0</sup> = 1. One way to solve this problem is to create a status bit (valid bit (V)). This bit is an output. We could say that this bit is 1 if the input is valid, and 0 if not. Thus, this bit is only 0 when all inputs are 0.

So I can write the equation of V as:  $V = I_0 + I_1 + I_2 + I_3 + I_4 + I_5 + I_6 + I_7$

Other hardware devices could look at the status bit to determine whether a proper encoding was performed.

## Segment Display:

The 7-segment display consist in small bars (the segments) that, set in the way indicated in Figure below, with standard letters **a** through **g**; enable the representation of the ten decimal numbers (0-9) and some other characters.

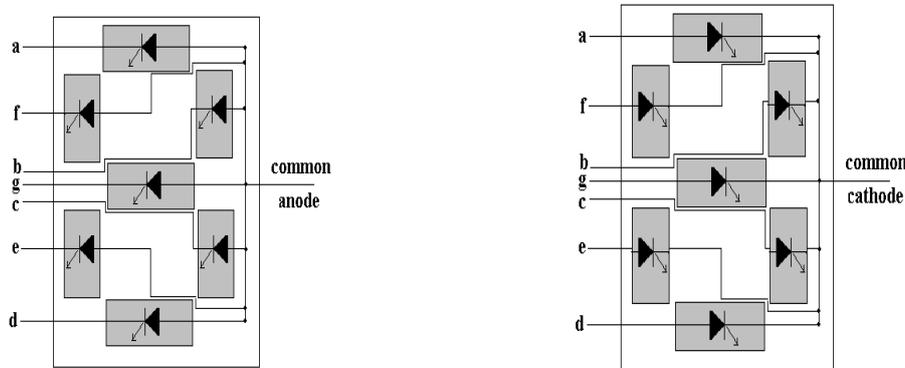


Figure(9)

Electrically the LEDs behave like standard diodes at solid state, with the only difference that there is a higher voltage between anode and cathode, in case of direct polarization.

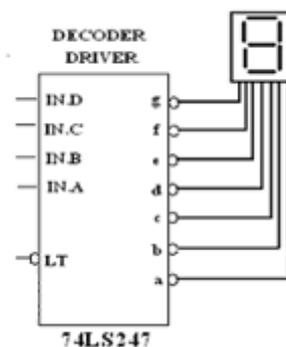
There are two main types of 7-segment displays (As shown in Figure below):

- 1-with common cathode, driven with positive logic.
- 2-with common anode, driven with negative logic.



Figure(10)

The 7-segment display device requires seven separate inputs. To use this display device, the binary code called Binary Coded Decimal (BCD) is converted to 7-segment code and supplied to the input of the display device. The circuit that performs the conversion is called a **BCD to 7-segment decoder/driver** (As shown in Figure below).The LT input used to test that all segments working.



Figure(11)



## Experiment 6

### Multiplexers Design and Implementation

#### Introduction:

A multiplexer (or data selector) is a device that is capable of taking two or more data lines and converting them into a single data line for transmission to another point.

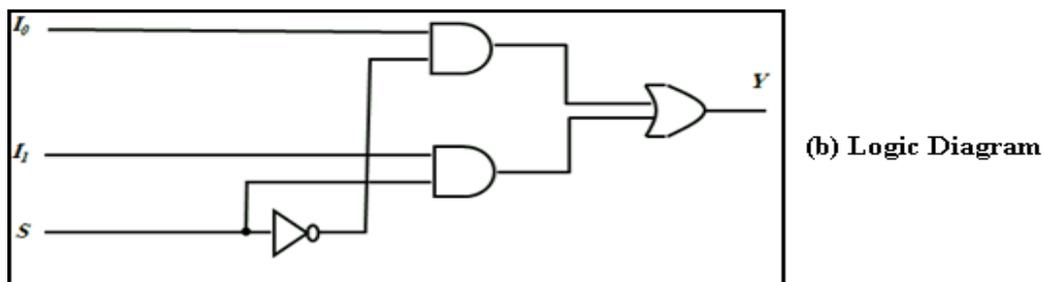
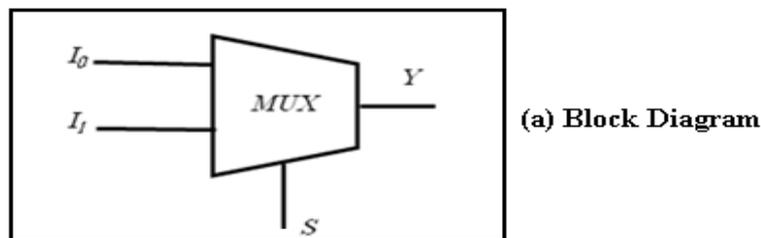
A multiplexer (MUX) performs the function of selecting the input on any one of 'n' input lines and feeding this input to one output line. Multiplexers are used as one method of reducing the number of integrated circuit packages required by a particular circuit design. This in turn reduces the cost of the system.

#### Objectives:

- Design, build, and test Multiplexers.
- Demonstrate the operations and applications of Multiplexers.
- Implement logic functions using Multiplexers.
- Use Tri-State Buffers to implement a Multiplexer.

#### Multiplexer

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output lines.
- The selection of particular input line is controlled by a set of selection lines.
- Normally, there are  $(2)^n$  input line and n selection lines whose combinations determine which input is selected.
- A 2-to-1 line multiplexer connects one of two 1-bit sources to a common destination as shown in figure below.



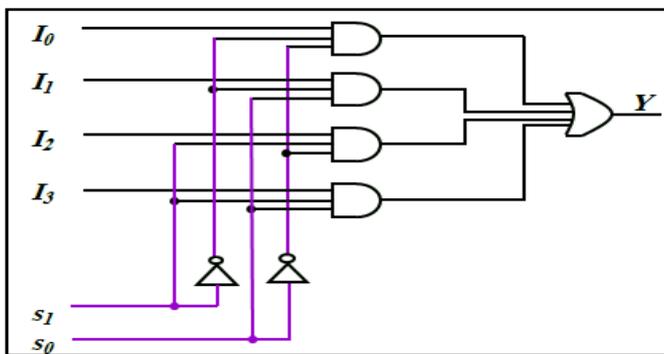
Figure(1)

Truth Table of 2-1 mux:

input			Output
S	I0	I1	
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

- The circuit has two data input lines, and one selection line S.
- When S=0, the upper AND gate is enabled and the I0 has a path to the output
- When S=1, the lower AND gate is enabled and I1 has path to the output.
- The multiplexer acts like an electric switch that selects one of two sources.

✚ A 4-to-1 line multiplexer is shown below



Figure(2)

- Each of the four inputs,  $I_0$  through  $I_3$ , 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 output of AND gates are applied to a single OR gate that provides the 1-line output.

### DeMultiplexers:

A Demultiplexer is a Combinational logic circuit that receives binary information from a single input and directs this information to one of many outputs. The selection is done by a binary value present at the select inputs. (As shown in fig 3)

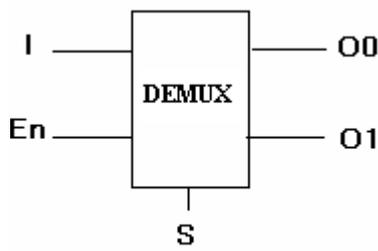
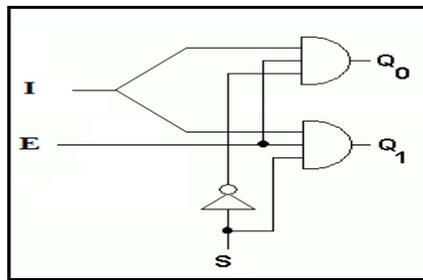


Fig 3

Input			Output	
I	En	S	O0	O1
X	0	X	X	X
0	1	0	0	X
1	1	0	1	X
0	1	1	X	0
1	1	1	X	1

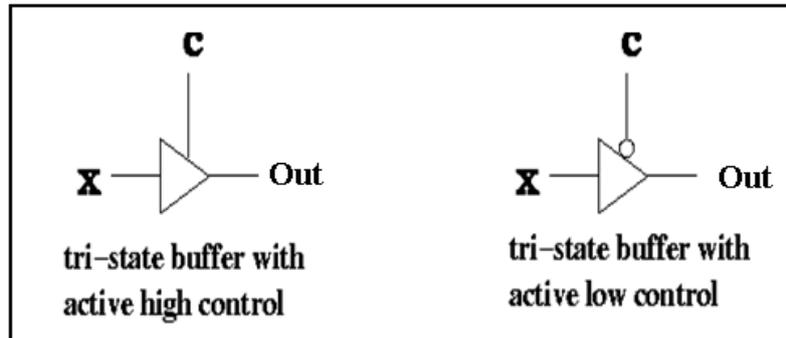


Figure(4):Demultiplexer

## Tri-state buffer

A tri-state buffer is a digital device that is capable of three different outputs, high, low and disconnected (high impedance).

Here's two diagrams of the tri-state buffer.



Figure(4)

A tri-state buffer has two inputs: a data input  $x$  and a control input  $c$ . The control input acts like a valve. When the control input is active, the output is the input. That is, it behaves just like a normal buffer. When the control input is not active no electrical current flows through, so the tri state buffer is in **high impedance state (Z)**, Thus, even if  $x$  is 0 or 1, that value does not flow through.

Here's a truth table describing the behavior of a active-high tri-state buffer.

c	x	Out
0	0	Z
0	1	Z
1	0	0
1	1	1

➔

c	Out
0	Z
1	x

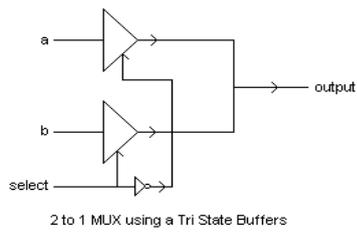
### Active-low tri-state buffers

Some tri-state buffers are active low. In an active-low tri-state buffer,  $c = 0$  turns open the valve, while  $c = 1$  turns it off. Here's the condensed truth table for an active-low tri-state buffer.

c	Out
0	x
1	Z

### Implementing 2-to-1 Multiplexe using Tristate buffer:

Given below is the circuit diagram of a 2 to 1 MUX implemented by a tristate buffer.





## Experiment 7

### Arithmetic Circuits Design and Implementation

#### Introduction:

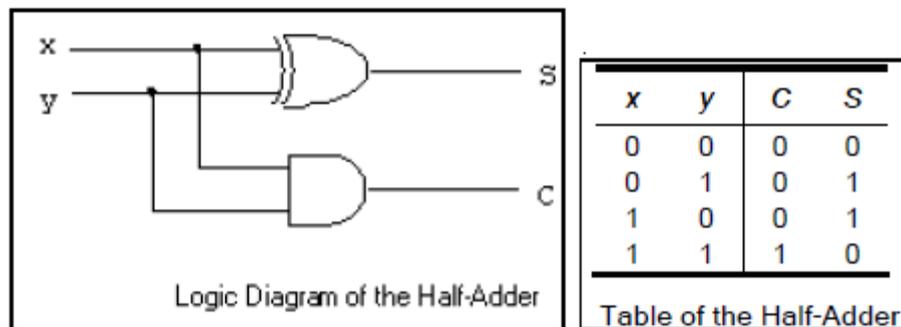
Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

#### Objectives:

- To understand the concept of Half and Full Adders.
- Design and build Ripple Carry Adder .
- Introduce 4-bit magnitude comparator.
- Design and implement binary multiplier

#### Half Adder:

Half adder is a combinational circuit that adds only two one bit numbers ,Since there are two inputs (x and y), only four possible combinations of inputs can applied . These four possibilities, and the resulting sums are shown in following truth table.



Figure(1)

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

$$C = xy$$

## Full Adder:

Full adder is a combinational circuit that adds three bits and generates a sum and carry.

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

Truth Table of the Full-Adder

From the truth table, we can obtain the Boolean expression of C & S outputs as follows :

$$S = x'y'z + x'yz' + xy'z' + xyz$$

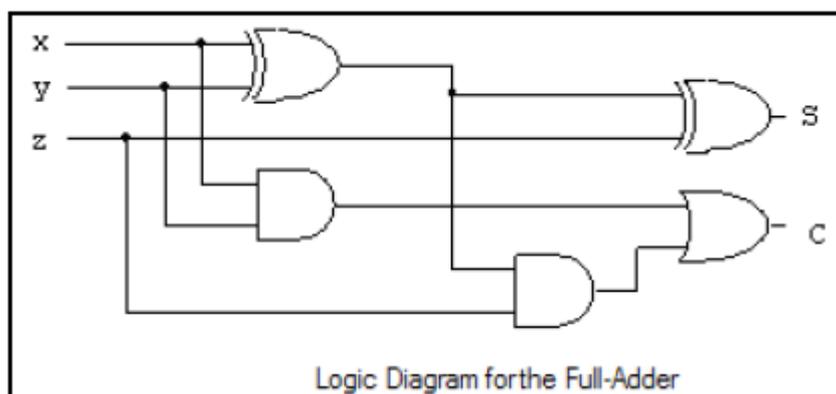
$$C = x'yz + xy'z + xyz' + xyz$$

Using Map-simplification method, we can get the simplified forms as follows :

$$S = x \oplus y \oplus z$$

$$C = xy + yz + xz$$

Now, we can construct the full-adder circuit based on the simplified Boolean expression of S and C outputs



Figure(2)

## Ripple Adder

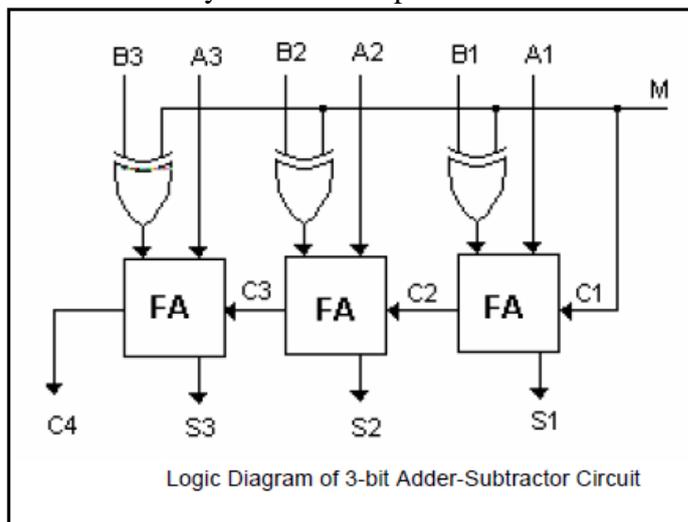
Two binary numbers, each of  $n$  bits, can be added using a ripple adder, a cascade of  $n$  full adders; each full adder handles one bit. Each Cout of a full adder is connected to the Cin of the higher full adder. The Cin of the least significant full adder is set to 0.

## Adder-Subtractor circuit

The subtraction of two binary numbers can be done by taking the 2's complement of the subtrahend and adding it to the minuend. The 2's complement can be obtained by taking the 1's complement and adding 1. To perform  $A - B$ , we complement the four bits of  $B$ , add them to the four bits of  $A$ , and add 1 to the input carry.

We may use XOR gate as an inverter if placing a logic "1" at one of the inputs. This helps in getting the 1's complement of the subtrahend; then we add "1" to get the 2's complement; which in turn is added to the minuend to get the final result of the subtraction.

Figure below shows adder-subtractor circuit; the mode input  $M$  controls the operation; when  $M=0$ , the circuit is an adder. When  $M=1$ , the circuit becomes a subtractor. This circuit can be cascaded for any number of inputs.



Figure(3)

## Multiplier:

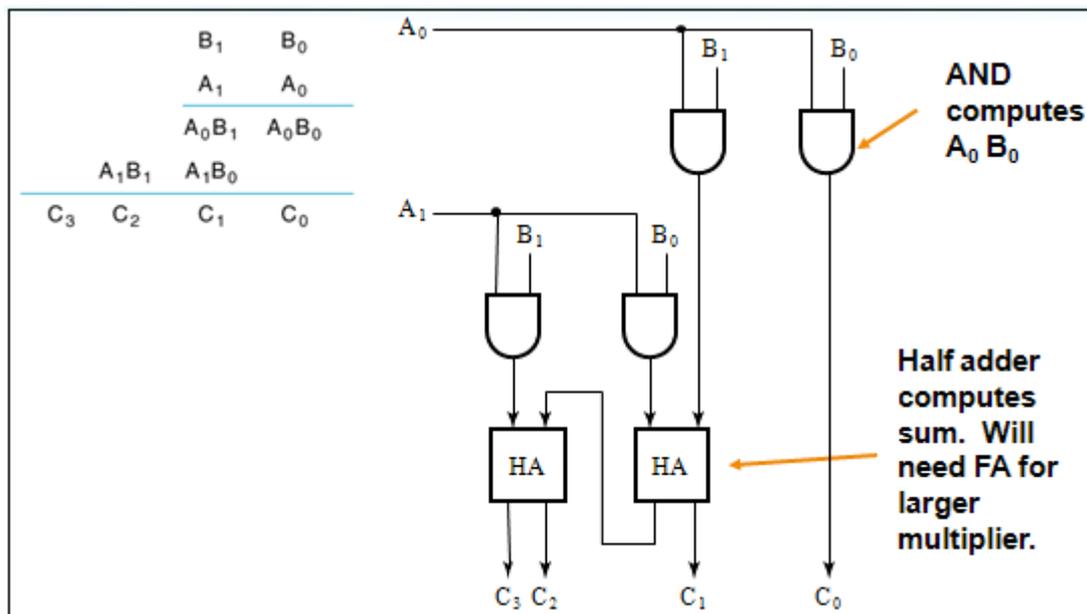
If we want to multiply tow numbers(A,B) ,each of them is consist of two bit as follows:

$$B = \{B_1 B_0\},$$

$$A = \{A_1 A_0\}$$

Then we multiply by doing single-bit multiplications and shifts.

$$\begin{array}{r}
 \phantom{A_1} B_1 \phantom{A_0} \phantom{A_1} B_0 \\
 \phantom{A_1} A_1 \phantom{A_0} \\
 \hline
 \phantom{A_1} A_0 B_1 \phantom{A_1} A_0 B_0 \\
 \phantom{A_1} A_1 B_1 \phantom{A_1} A_1 B_0 \\
 \hline
 C_3 \phantom{C_2} C_2 \phantom{C_1} C_1 \phantom{C_0} C_0
 \end{array}$$



Figure(4)

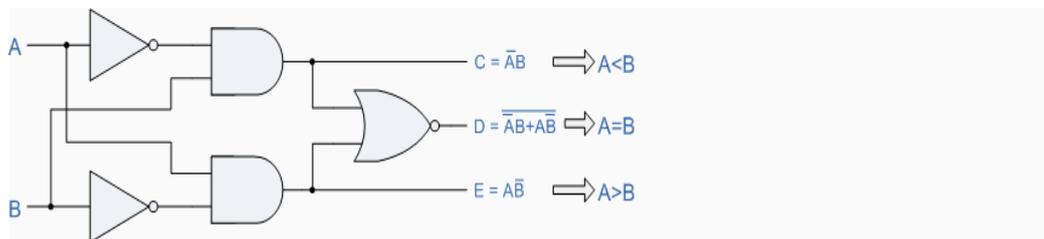
## Comparator:

Another common and very useful combinational logic circuit is that of the **Digital Comparator** circuit. Digital or Binary Comparators are made up from standard AND, NOR and NOT gates that compare the digital signals at their input terminals and produces an output depending upon the condition of the inputs. For example, whether input A is greater than, smaller than or equal to input B etc.

**Digital Comparators** can compare a variable or unknown number for example A (A1, A2, A3, .... An, etc) against that of a constant or known value such as B (B1, B2, B3, .... Bn, etc) and produce an output depending upon the result. For example, a comparator of 1-bit, (A and B) would produce the following three output conditions.

$$A > B, A = B, A < B$$

### 1-bit Comparator



Then the operation of a 1-bit digital comparator is given in the following Truth Table.

### Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



## Experiment 8

### Introduction to Latches and Flip-Flops and registers

#### Introduction:

The logic circuits that have been used until now were combinational logic circuits since the output of the device depends on the input data. Sequential logic circuits are defined as circuits whose outputs depend both on the present values of the inputs and the previous state of the circuits. Latches and flip-flops are basic sequential circuit whose operation we will investigate during this experiment. The difference between these two sequential devices is that flip-flop's output changes only at specific times determined by a clocking signal, while latch's output changes independent of a clocking signal.

Sequential circuits form the basis of registers, memories, and state machines, which in turn are vital functional units in digital design.

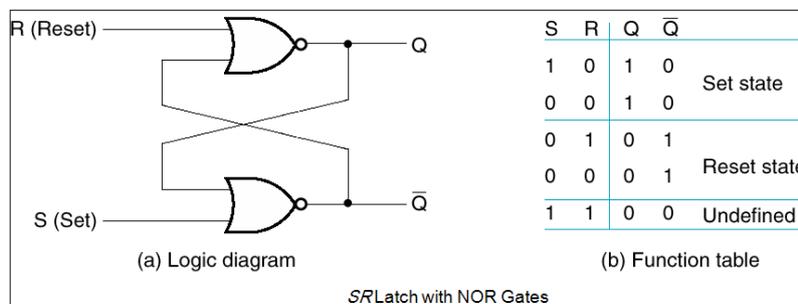
#### Objectives:

- Design, build, and test various sequential logic circuits.
- An in-depth study of the operation of S-R, J-K, master-slave, and edge-triggered latches and flip-flops.
- An introduction to commercially available flip-flops.

#### Procedure:

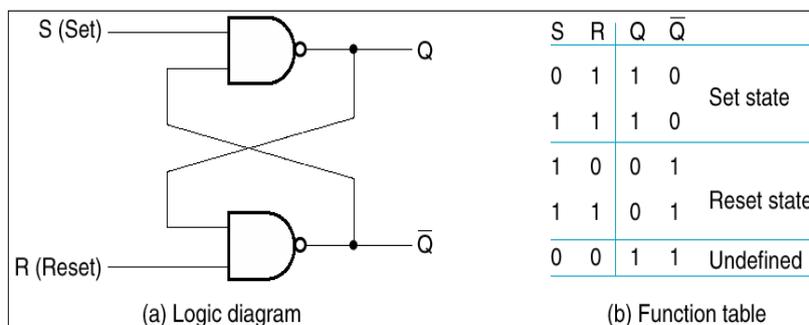
##### 1- The S-R Latch

The most basic sequential unit is the S-R latch. From this basic circuit flip-flops are constructed, and from flip-flops, the registers, memories, and state machines can be made. The basic S-R latch has two inputs, S and R, and two outputs, Q and Q`.



Figure(1)

Similar SR latch can be made from NANDs as follow:

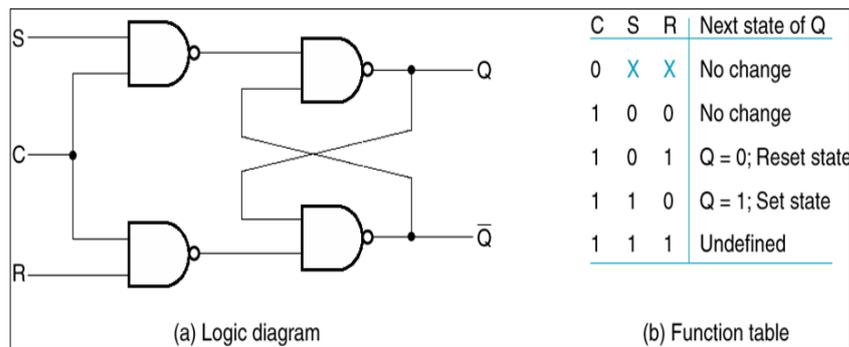


Figure(2)

## 2- The S-R Latch with Clock (S-R Flip-Flop)

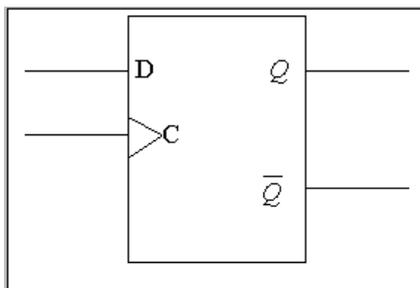
To achieve synchronous operation, the latch should change state only on the proper clock signal. For example, assume that the latch should change state only when the clock signal goes high, else the latch holds its value independently from the value of S and R.

So we can adjust the circuit we have implemented above to have a third input (Clk).



Figure(3)

## 3- D Flip-Flop

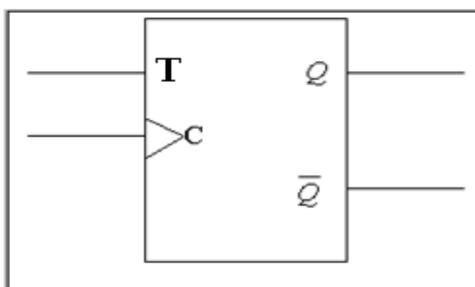


D	C	Q	$\bar{Q}$	Operation
0	Rising edge	0	1	Reset (stores 0)
1	Rising edge	1	0	Set (stores 1)

Figure(4)

- Notice that a D flip flop can be made from S-R flip flop by ensuring that the S and R outputs are the complement of each other at all times.

## 4- T Flip-Flop

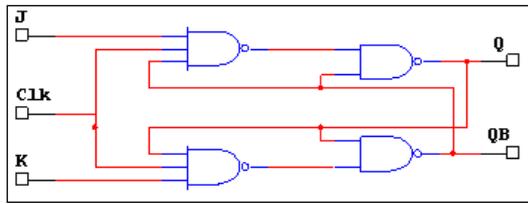


T	C	Q	Operation
0	Rising edge	0	Q (No change)
1	Rising edge	1	$\bar{Q}$ (complement)

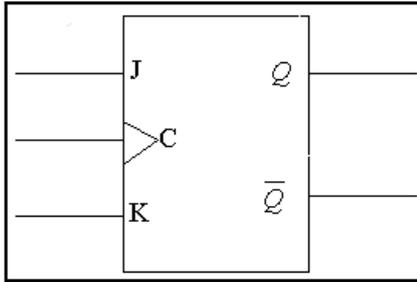
Figure(5)

### 5- J-K Flip-Flop

The J-K flip-flop is simply an S-R flip-flops that has been modified so that both inputs can be active at the same time. Where in the S-R flip-flop this condition was considered invalid, in the J-K flip-flop this condition toggles the output on successive clock cycles.



Figure(6)

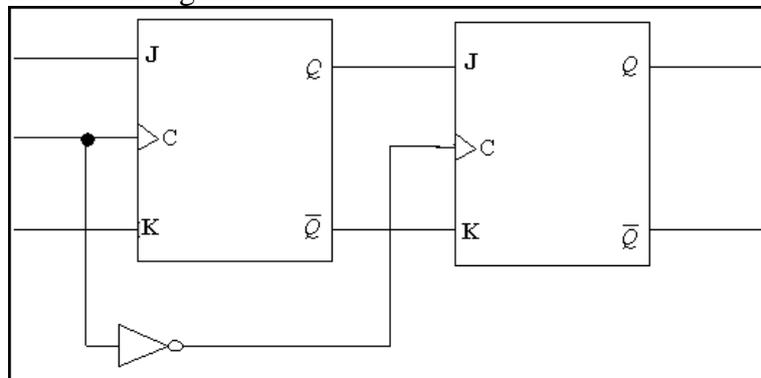


Figure(7)

J	K	C	Q	$\bar{Q}$	Operation
0	0	Rising edge	$Q_0$	$\bar{Q}_0$	Hold (no change)
0	1	Rising edge	0	1	Reset
1	0	Rising edge	1	0	Set
1	1	Rising edge	$\bar{Q}_0$	$Q_0$	Toggle

### 6- Master-Slave Flip-Flop

There is a slight problem with using a clock pulse. During the time the clock is high, the flip-flop performs identically to the regular asynchronous latch. Thus, if the inputs changed multiple times while the clock was high, the state of the latch could also change multiple times. One technique for eliminating multiple- state transition during a single clock cycle is the use of a master-slave arrangement.



Figure(8)

The left or master Latch in Figure above forms the inputs to the flip-flop, and the right or slave latch forms the outputs of the flip-flop. The master latch looks at the inputs while the clock is high. When the clock returns low, the slave latch is enabled, using the outputs of the master latch as its inputs. Thus the inputs are "read" while the clock is high and transferred to the outputs when the clock returns low.

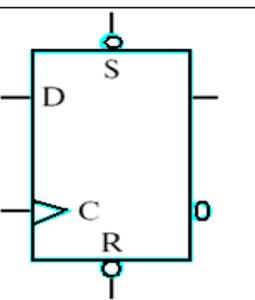
## 7- Direct inputs:

- **Set/Reset independent of clock**

- Direct set or preset
- Direct reset or clear

S	R	C	D	Q	$\bar{Q}$
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	Undefined	
1	1	↑	0	0	1
1	1	↑	1	1	0

(b) Function table

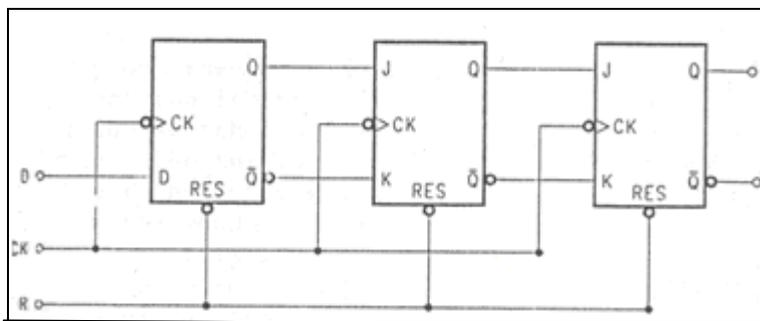


(c) Simplified Symbol

Figure(9)

## 8- 3-Stage Shift Register

A group of cascaded FFs used to store related bits of information is known as a register. A register that is used to store information arriving from a source is called a shift register. Each FF output of a shift register is connected to the input of the next FF, and a common clock pulse is applied to all FFs. Hence, the shift register is a synchronous sequential circuit. The storage capacity of a register is the number of bits of digital data it can store. Each FF in a register represents one-bit storage capacity, therefore, the number of FFs in a register determine its total storage capacity.



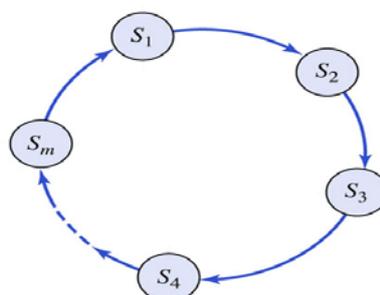
Figure(10)

## Counter:

Counter: is essentially a register that goes through a predetermined sequence of states. The gates in the counter are connected in such a way as to produce the prescribed sequence of binary states.

The counting sequence is often depicted by a graph called a **state diagram**.

A counter with m-states has the following state diagram:



Each node  $S_i$  denotes the states of the counter and the arrows in the graph denote the order in which the states occur.

Counters are available in two categories: **ripple (Asynchronous) counters** and **synchronous counters**.

### 1) Ripple (Asynchronous) Counter:

In a ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops; In other words, clock inputs of the flip-flops are triggered by output transitions of other Flip-flops, rather than a common clock signal.

The output of each FF is connected to the clock input of the next flip-flop in sequence.

### 3-Stage Asynchronous Binary Counter

In the previous experiment, the edge-triggered JK FF was wired to operate as a toggle. Every time a clock pulse was detected at the input, the output changed state. After two clock pulses were detected, the output of the FF returned to its original state. As a result, there were two state changes of the output and the frequency of the input clock was divided by two. Therefore two events occurred, the number of clock pulses was counted and the frequency of the output was divided by 2. The circuit of Figure 3 contains the logic diagram for a three bit asynchronous binary counter with Q<sub>2</sub> being the MSB. The frequency of the input clock is divided by two for the first FF and divided by two for the second FF and then divided by two again for the third FF. The frequency at Q<sub>2</sub> has been divided by eight or  $2^n$  where n is the number of FFs in the circuit. There are also eight states in the truth table. This factor  $2^n$  is also called the **Modulus or MOD** of the counter. Since this counter has 3 FFs, it is referred to as a MOD 8 counter. The MOD of any counter may be modified by connecting the proper combinational logic between the outputs of the appropriate FF and the Clear input. To convert the counter in Figure 3 to a MOD 7 counter, NAND the Q<sub>0</sub>, Q<sub>1</sub>, Q<sub>2</sub> inputs and connect the output of the NAND gate to the CLEAR input (active low input) of all the FFs. Figure 3 is an asynchronous device since the preceding FF must complete one cycle to provide the clock pulse for the next FF in the counter. The FFs do not change state at the same time and this creates a ripple effect in the way that the output of each FF changes state. This ripple effect is more noticeable in a MOD 16 or higher counter when the count resets from 15 or the maximum count back to 0. Another name for the asynchronous counter is the Ripple Counter.

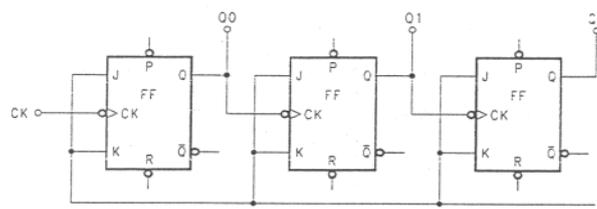


Figure (11)

### Advantages of Ripple Counters:

- Simple hardware and design.

### Disadvantages of Ripple Counters:

- They are asynchronous circuits, and can be unreliable and delay dependent, if more logic is added.
- Large ripple counters are slow circuits due to the length of time required for the ripple to occur.

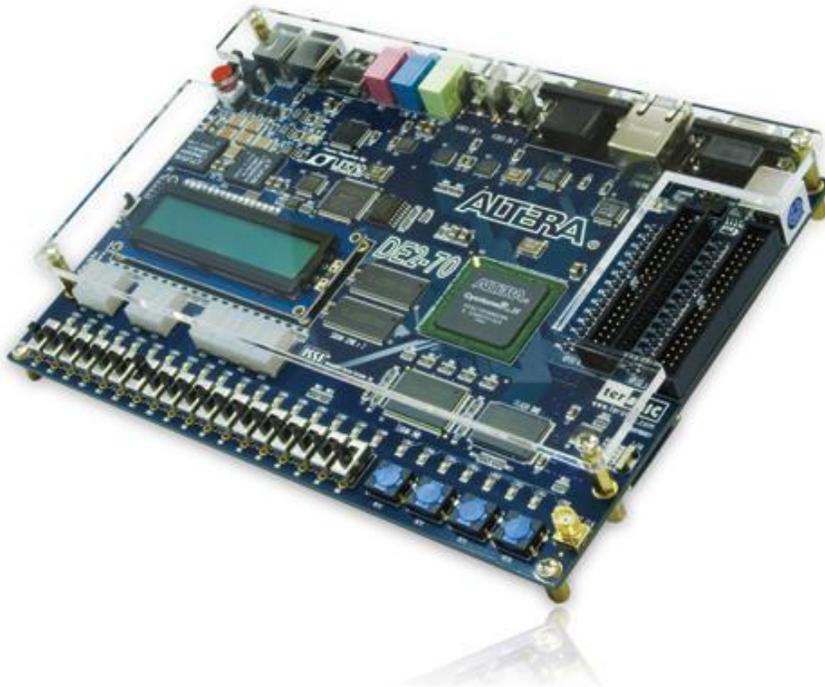
## 2) Synchronous Binary Counter

In the previous Asynchronous binary counter example, we saw that the output of one counter stage is connected directly to the input of the next counter stage and so on along the chain, and as a result the asynchronous counter suffers from what is known as "Propagation Delay". However, with **Synchronous Counters**, the external clock signal is connected to the clock input of EVERY individual flip-flop within the counter so that all of the flip-flops are clocked together simultaneously (in parallel) at the same time giving a fixed time relationship. This results in all the individual output bits changing state at exactly the same time with no ripple effect and therefore, no propagation delay.



University of Jordan  
Faculty of Engineering and Technology  
Department of Computer Engineering  
Digital Logic Laboratory 0907234

## Labsheet10: Memory Design



Name:

Student ID:

Section:

❖ In this lab you are required to implement the 2x2 Memory shown in Figure 1.

Create a new project named **MainMemory** in QuartusII. Then add the files **MemCell.bdf**, **MemWord.bdf**, **dec1to2.bsf**, and **Memory.bdf** to your project.

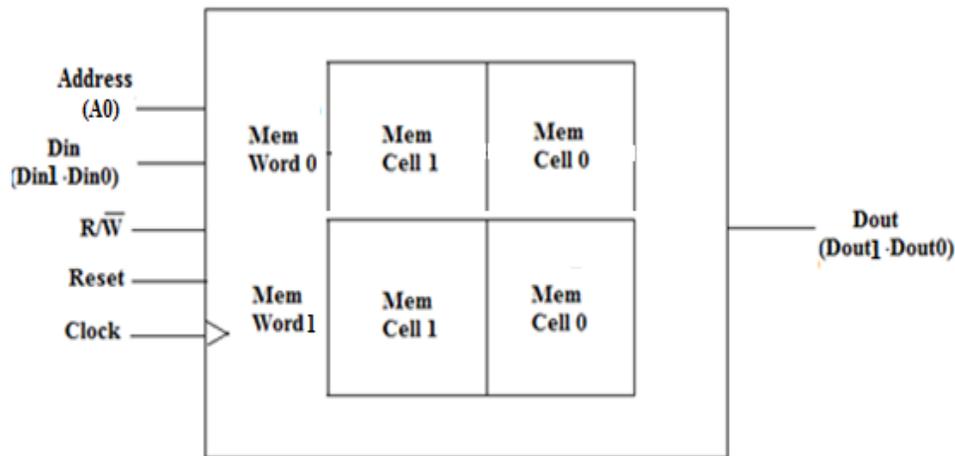


Figure 1. 2x2 Memory

**Part 1: Memory Cell Implementation:**

1. Open the file **MemCell.bdf** and build the schematic diagram of the memory cell circuit shown in Figure 2.

Note: the name of the D-Flip Flop symbol in QuartusII library is 'dff' and the name of the Tri-State buffer symbol is 'tri'.

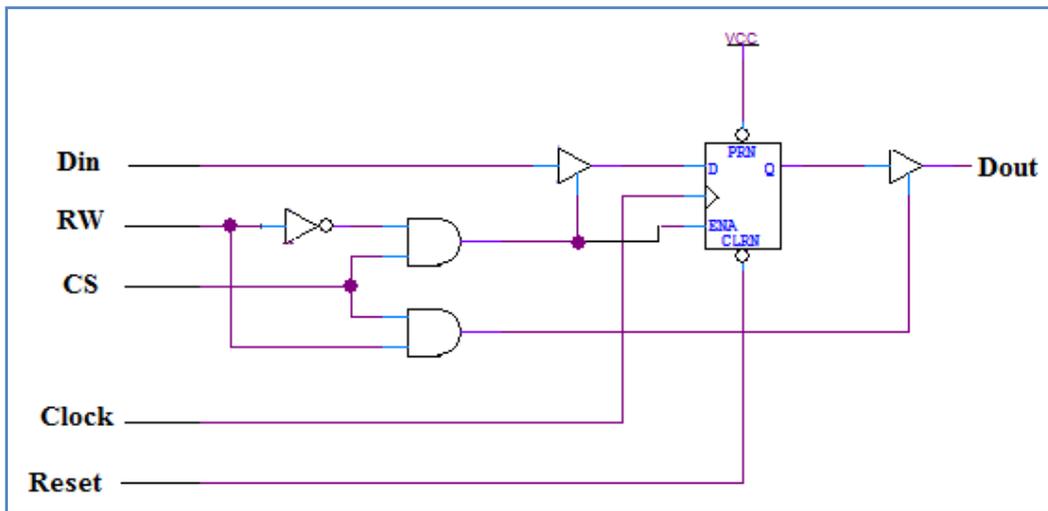


Figure 2. Memory Cell Schematic Diagram

2. Convert the memory cell schematic diagram you built to a block as in Figure 3.

Note: (File → Create/update → Create symbol files for current file)

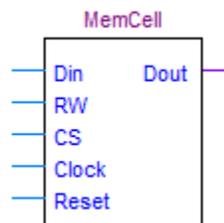
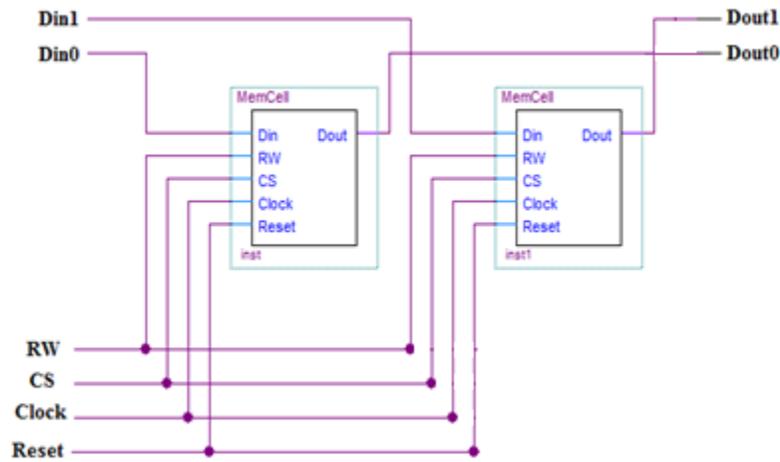


Figure 3. Memory Cell block

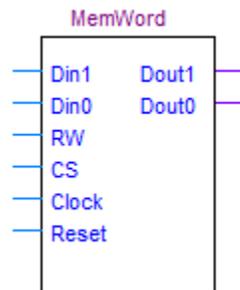
**Part 2: Memory Word Implementation:**

1. Open the file **MemWord.bdf** and build the schematic diagram of the memory word circuit shown in Figure 4. Note that you need to use the memory cell block you created in the first part.



**Figure4. MemoryWord Schematic Diagram**

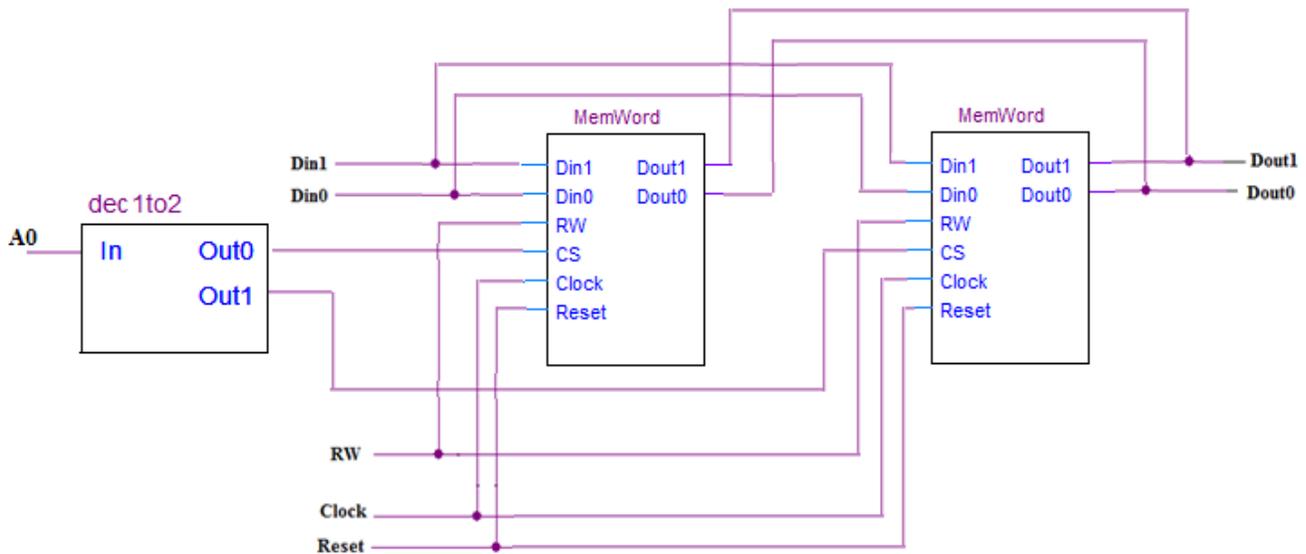
2. Convert the memory word schematic diagram you built to a block as in Figure 5.  
Note: (File → Create/update → Create symbol files for current file)



**Figure5. MemoryWord block**

### Part 3: 2x2 Memory Implementation:

1. Open the file **Memory.bdf** and build the schematic diagram of the 2x2 memory shown in Figure 6 using the memory cell block you created in the second part and the 1-to-2 decoder block (provided in the **dec1to2.bsffile**).



**Figure6. 2x2 MemorySchematic Diagram**

2. Perform the following pins assignment:

- Inputs:
  - Din0: SW[1] → PIN\_AB26
  - Din 1: SW[2] → PIN\_AB25
  
  - A0: SW[10] → PIN\_W5
  
  - RW: SW [16] → PIN\_L7
  - Reset: SW [17] → PIN\_L8
  
  - Clock: PB0 → PIN\_T29
- Outputs:
  - Dout0: LEDR[1] → PIN\_AK5
  - Dout1: LEDR[2] → PIN\_AJ5

3. Download your design on the FPGA and test it.