Experiment 1

Introduction to Altera and Schematic Programming

Prepared by: Eng. Shatha Awawdeh, Eng.Eman Abu_Zaitoun

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 XOR B = (A& (~B)) | ((~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…).

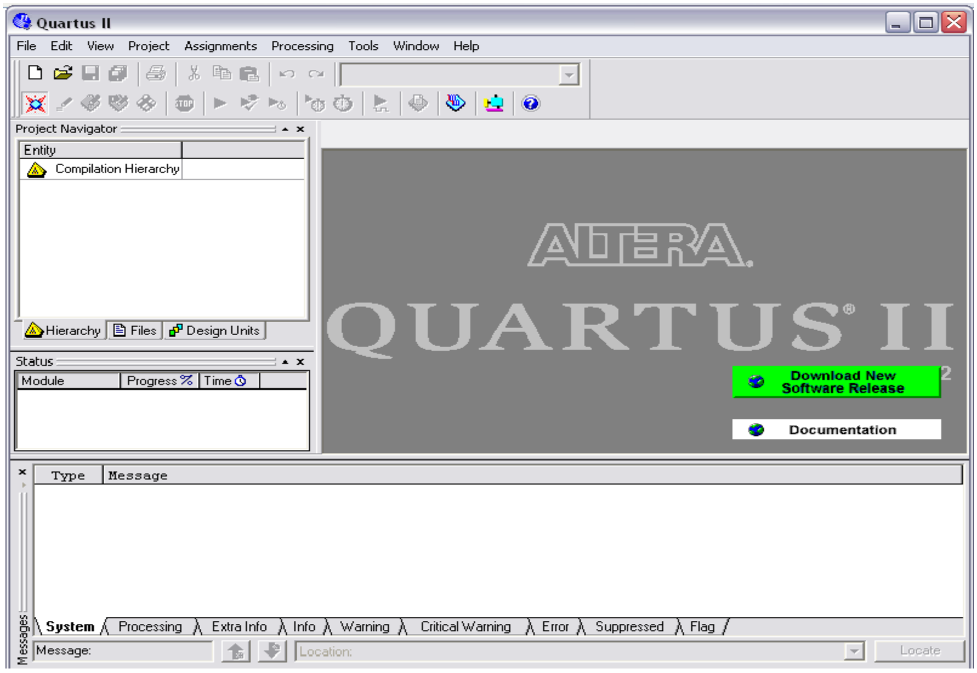


Figure 1. The main Quartus II display.

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.

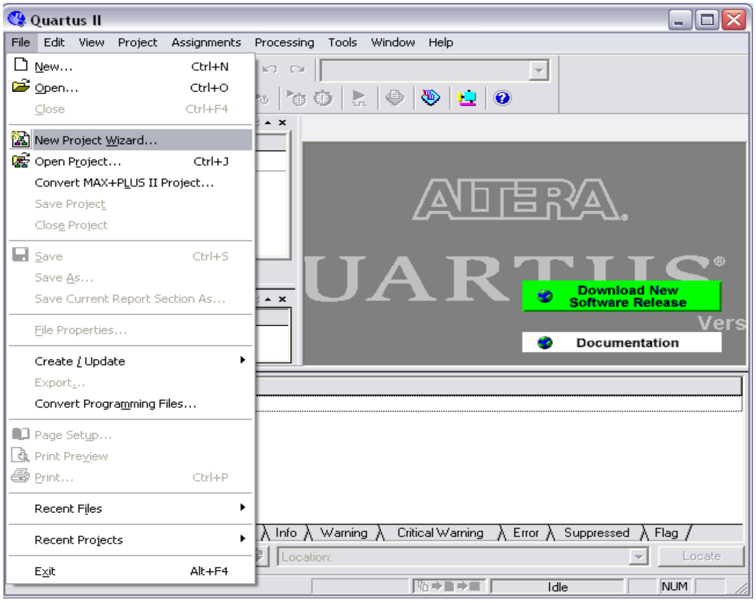
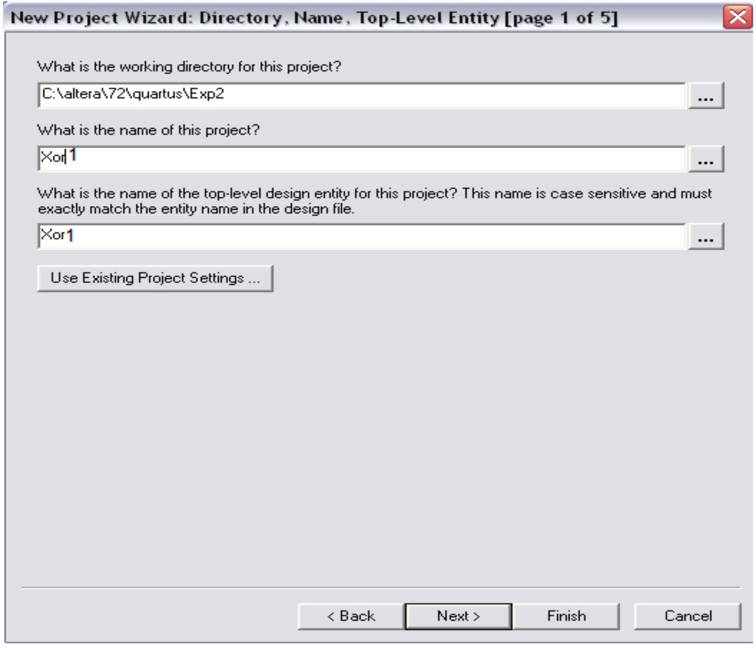


Figure 2a. Creation of a new project.

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.


New Project Wizard: Directory, Name, Top-Level Entity [page 1 of 5]

What is the working directory for this project?

C:\altera\72\quartus\Exp2

What is the name of this project?

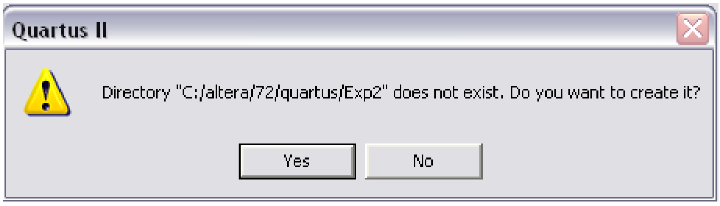
Xor1

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

Xor1

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.

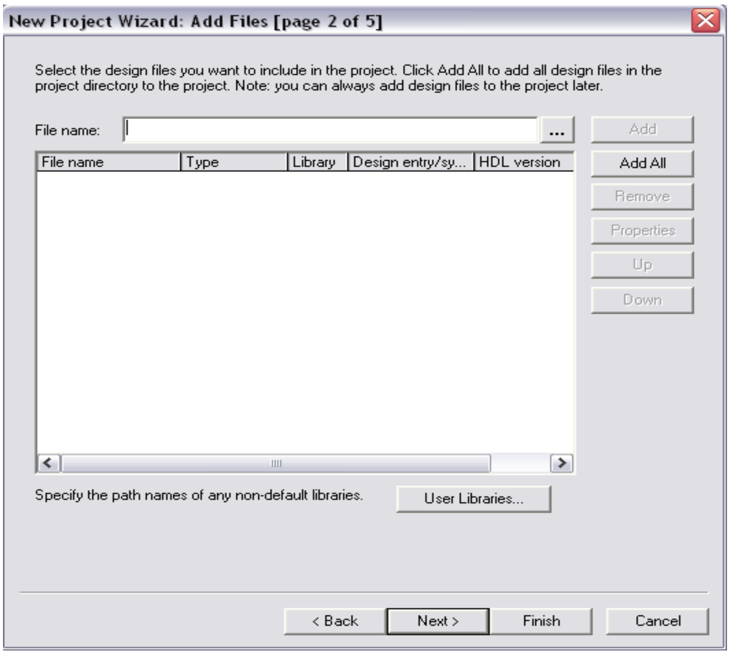


Quartus II

Directory "C:/altera/72/quartus/Exp2" does not exist. Do you want to create it?

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.



New Project Wizard: Add Files [page 2 of 5]

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project. Note: you can always add design files to the project later.

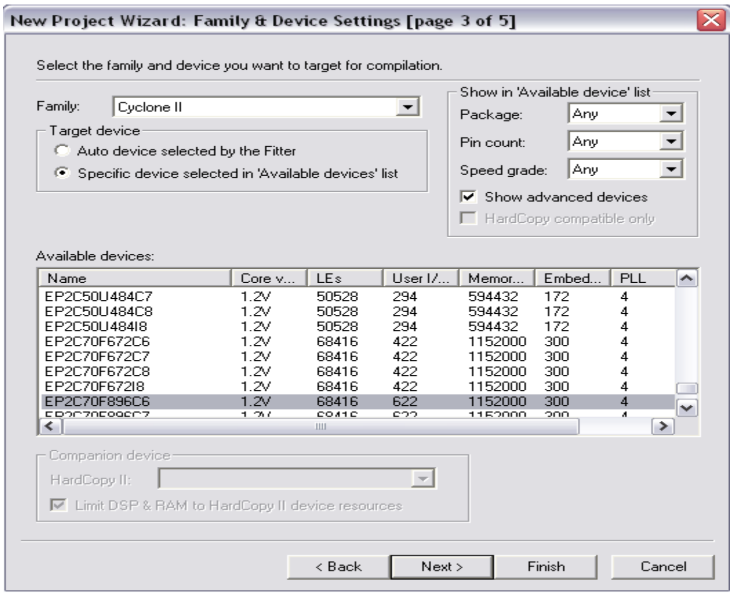
File name:

File name	Type	Library	Design entry/sy...	HDL version

Specify the path names of any non-default libraries.

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.



New Project Wizard: Family & Device Settings [page 3 of 5]

Select the family and device you want to target for compilation.

Family: Cyclone II

Target device

○ Auto device selected by the Fitter

● Specific device selected in 'Available devices' list

Show in 'Available device' list

Package: Any

Pin count: Any

Speed grade: Any

☑ Show advanced devices

☐ HardCopy compatible only

Available devices:

Name	Core v...	LEs	User I/...	Memor...	Embed...	PLL
EP2C50U484C7	1.2V	50528	294	594432	172	4
EP2C50U484C8	1.2V	50528	294	594432	172	4
EP2C50U484I8	1.2V	50528	294	594432	172	4
EP2C70F672C6	1.2V	68416	422	1152000	300	4
EP2C70F672C7	1.2V	68416	422	1152000	300	4
EP2C70F672C8	1.2V	68416	422	1152000	300	4
EP2C70F672I8	1.2V	68416	422	1152000	300	4
EP2C70F896C6	1.2V	68416	622	1152000	300	4
EP2C70F896C7	1.2V	68416	622	1152000	300	4

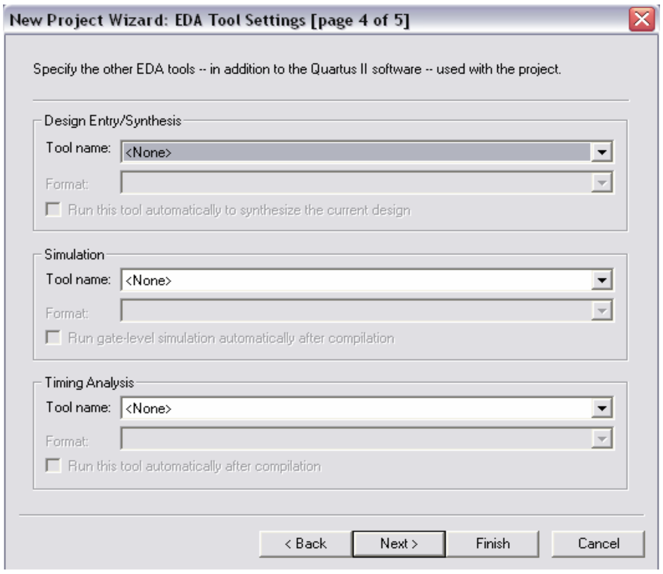
Companion device

HardCopy II:

☑ Limit DSP & RAM to HardCopy II device resources

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**.



New Project Wizard: EDA Tool Settings [page 4 of 5]

Specify the other EDA tools -- in addition to the Quartus II software -- used with the project.

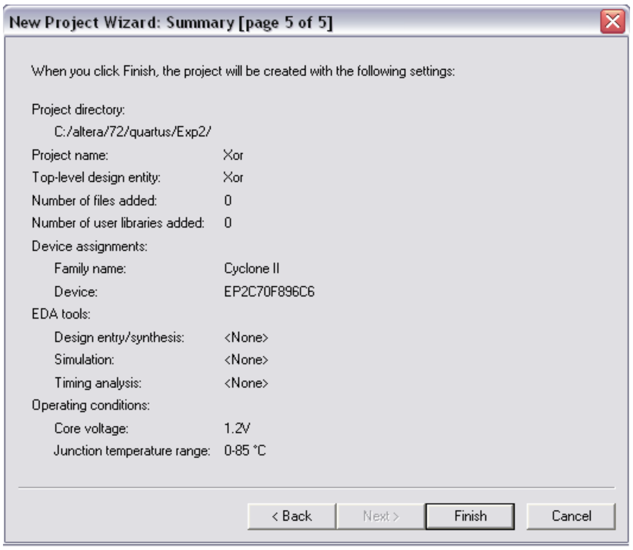
Design Entry/Synthesis
Tool name: <None>
Format:
□ Run this tool automatically to synthesize the current design

Simulation
Tool name: <None>
Format:
□ Run gate-level simulation automatically after compilation

Timing Analysis
Tool name: <None>
Format:
□ Run this tool automatically after compilation

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.

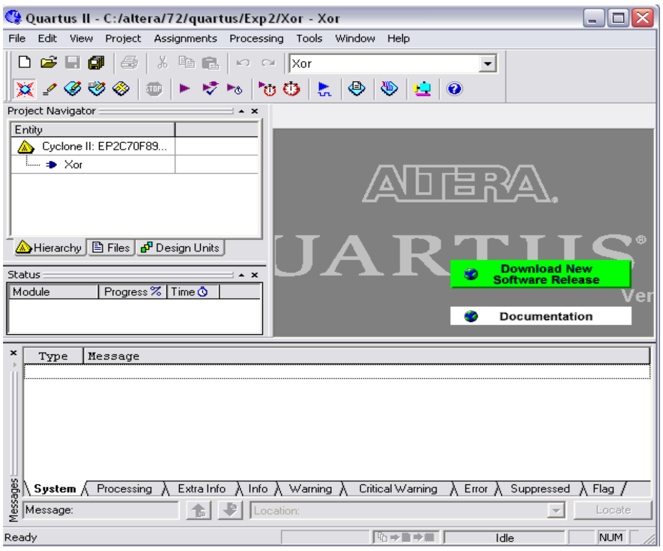


New Project Wizard: Summary [page 5 of 5]

When you click Finish, the project will be created with the following settings:

Project directory:	
C:/altera/72/quartus/Exp2/	
Project name:	Xor
Top-level design entity:	Xor
Number of files added:	0
Number of user libraries added:	0
Device assignments:	
Family name:	Cyclone II
Device:	EP2C70F896C6
EDA tools:	
Design entry/synthesis:	<None>
Simulation:	<None>
Timing analysis:	<None>
Operating conditions:	
Core voltage:	1.2V
Junction temperature range:	0-85 °C

Figure 7. Summary of the project settings.

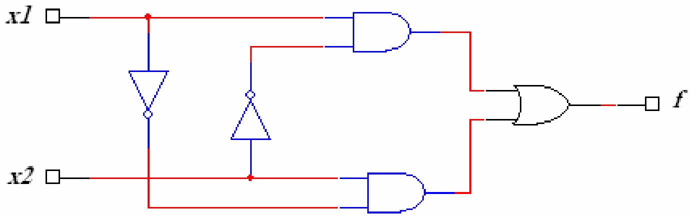


Quartus II - C:/altera/72/quartus/Exp2/Xor - Xor

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.



X1	*X2*	*f*
0	0	0
0	1	1
1	0	1
1	1	0

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.

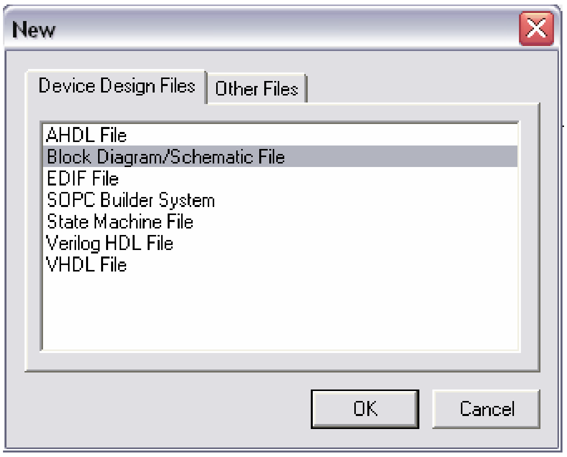
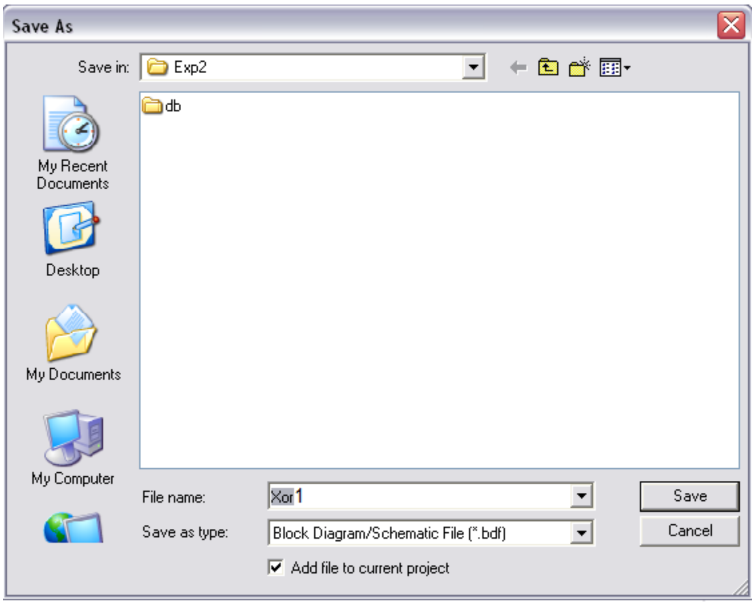


Figure 10. Choose to prepare a block diagram

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 **Xor1**, 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.



Save in: Exp2

File name: Xor1

Save as type: Block Diagram/Schematic File (*.bdf)

☑ Add file to current project

Figure 11. Name the file.

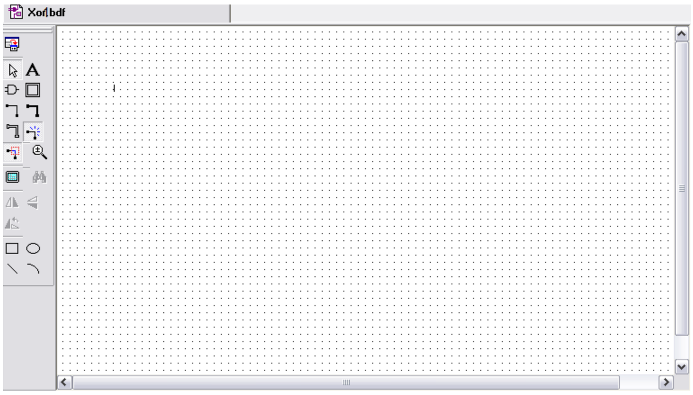


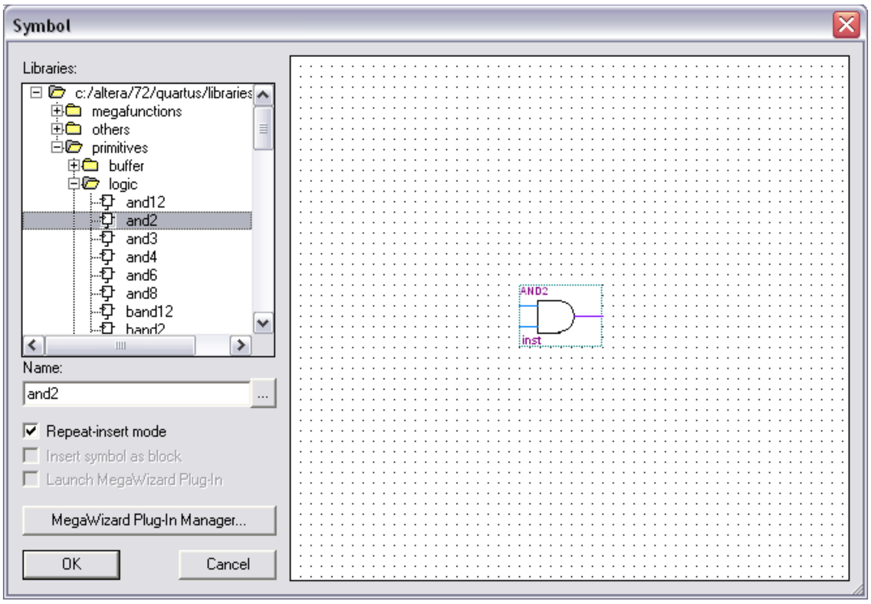
Figure 12. Graphic Editor window.

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.



Name: and2

☑ Repeat-insert mode

□ Insert symbol as block

□ Launch MegaWizard Plug-In

Figure 13. Choose a symbol from the library.

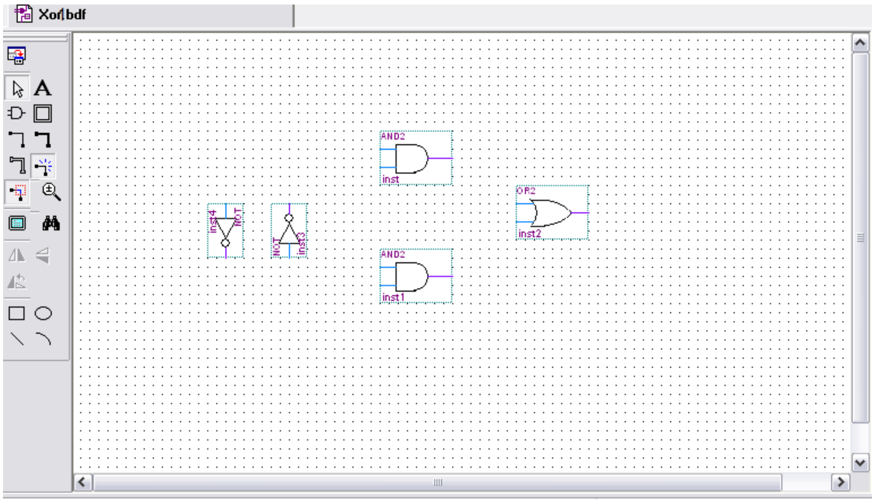


Figure 14. Import the gate symbols into the Graphic Editor window.

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.

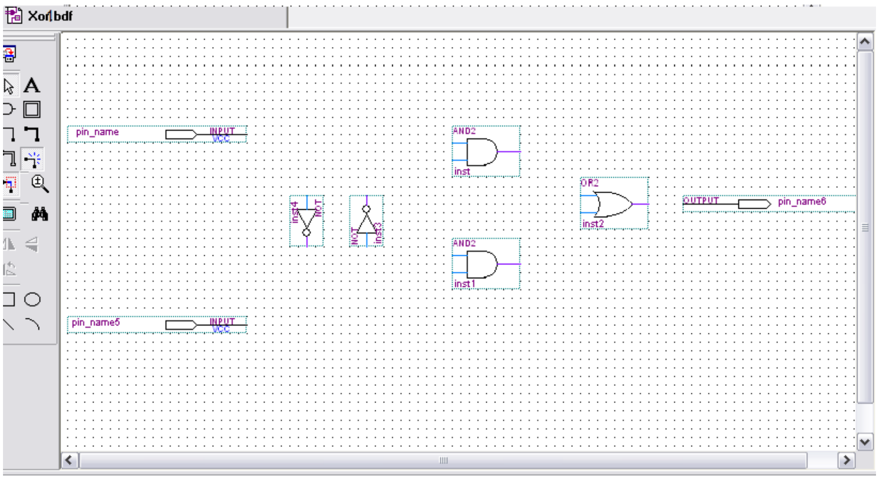
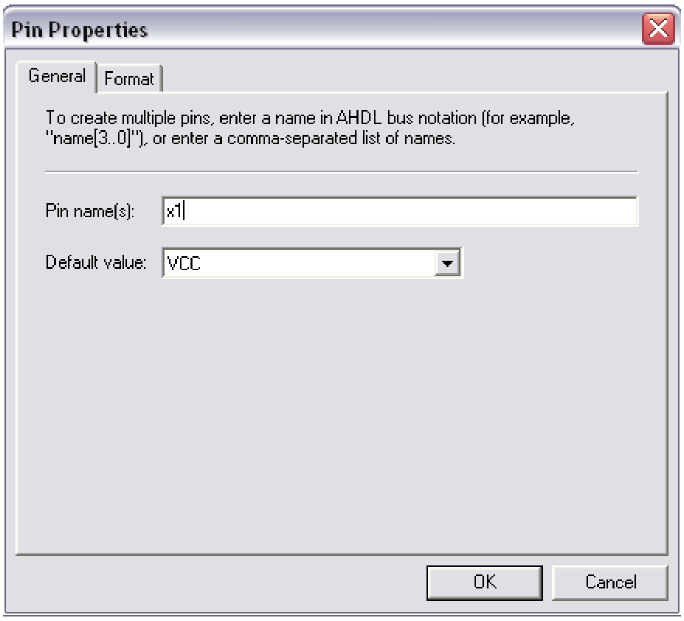


Figure 15. Import the input and output pins.

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.



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

Pin name(s): x1

Default value: VCC

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.**

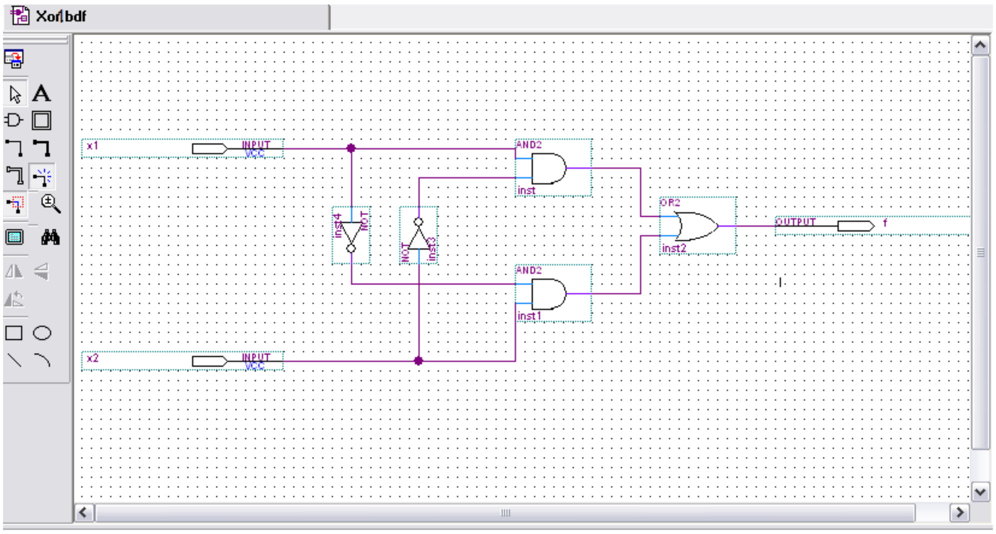


Figure 17. The completed schematic diagram.

4- Compiling the Designed Circuit

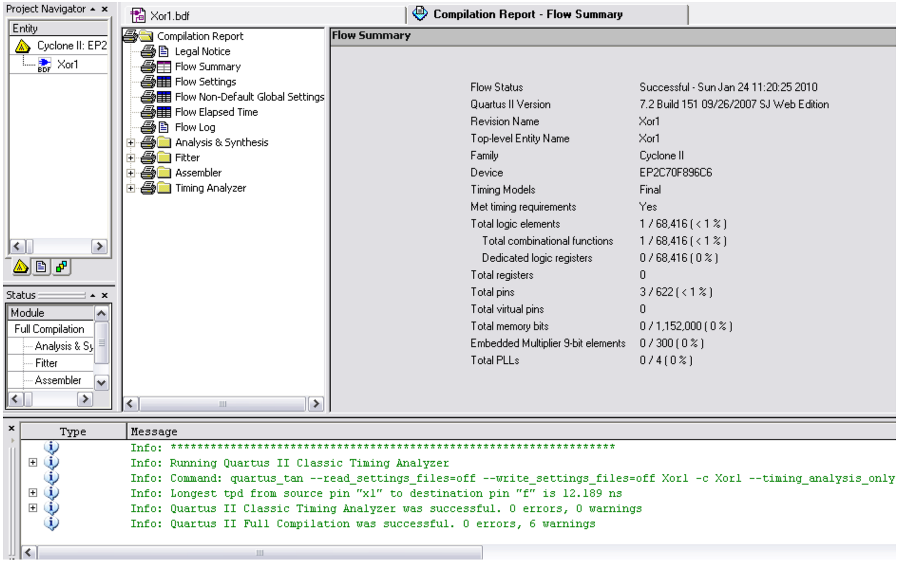
The entered schematic diagram file, Xor.bdf, is processed by several Quartus II tools that analyze the file, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the Compiler.

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.

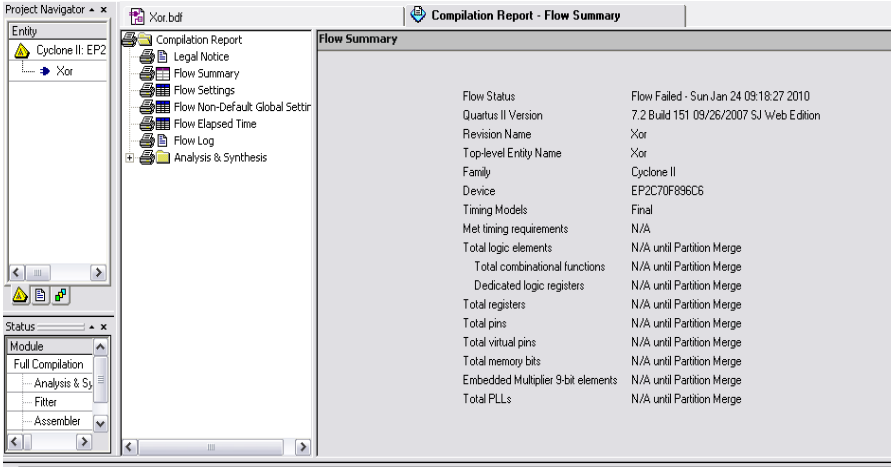


```
Flow Status                              Successful - Sun Jan 24 11:20:25 2010
Quartus II Version                       7.2 Build 151 09/26/2007 SJ Web Edition
Revision Name                            Xor1
Top-level Entity Name                    Xor1
Family                                   Cyclone II
Device                                   EP2C70F896C6
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: ******************************************************************
Info: Running Quartus II Classic Timing Analyzer
Info: Command: quartus_tan --read_settings_files=off --write_settings_files=off Xorl -c Xorl --timing_analysis_only
Info: Longest tpd from source pin "x1" to destination pin "f" 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).



```
Flow Status                          Flow Failed - Sun Jan 24 09:18:27 2010
Quartus II Version                   7.2 Build 151 09/26/2007 SJ Web Edition
Revision Name                        Xor
Top-level Entity Name                Xor
Family                               Cyclone II
Device                               EP2C70F896C6
Timing Models                        Final
Met timing requirements              N/A
Total logic elements                 N/A until Partition Merge
   Total combinational functions     N/A until Partition Merge
   Dedicated logic registers         N/A until Partition Merge
Total registers                      N/A until Partition Merge
Total pins                           N/A until Partition Merge
Total virtual pins                   N/A until Partition Merge
Total memory bits                    N/A until Partition Merge
Embedded Multiplier 9-bit elements   N/A until Partition Merge
Total PLLs                           N/A until Partition Merge
```

Figure 19. Compilation report for the failed design.

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

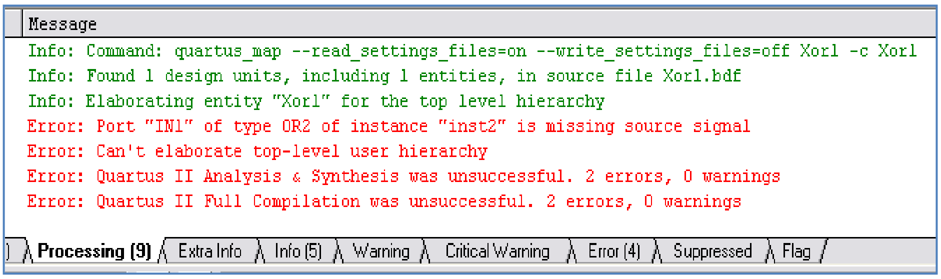

```
Message
Info: Command: quartus_map --read_settings_files=on --write_settings_files=off Xor1 -c Xor1
Info: Found 1 design units, including 1 entities, in source file Xor1.bdf
Info: Elaborating entity "Xor1" for the top level hierarchy
Error: Port "IN1" of type OR2 of instance "inst2" is missing source signal
Error: Can't elaborate top-level user hierarchy
Error: Quartus II Analysis & Synthesis was unsuccessful. 2 errors, 0 warnings
Error: Quartus II Full Compilation was unsuccessful. 2 errors, 0 warnings
```

Figure 20. Error messages.

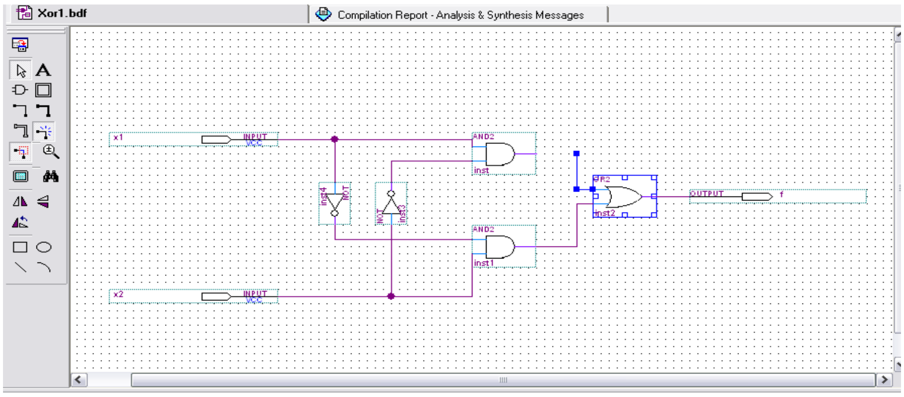


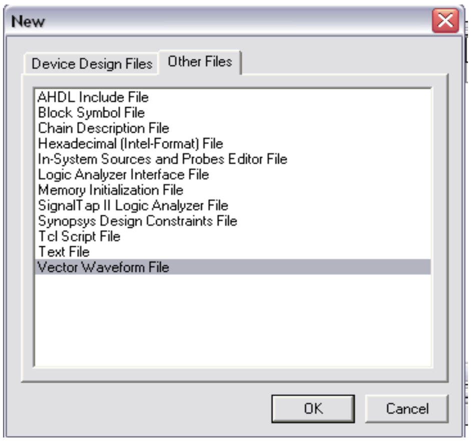
Figure 21. Identifying the location of the error.

After correcting all errors, recompile the circuit.

5- 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:

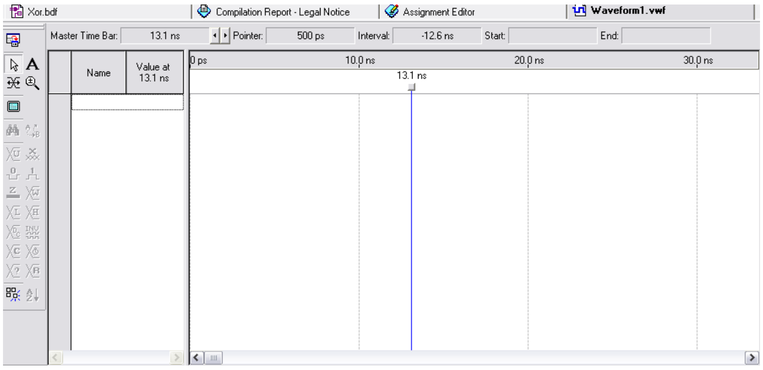
1. Open the Waveform Editor window by selecting **File > New>Other Files** tab, which gives the window shown in Figure 26.Choose **Vector Waveform File** and click **OK**.



AHDL Include File
Block Symbol File
Chain Description File
Hexadecimal (Intel-Format) File
In-System Sources and Probes Editor File
Logic Analyzer Interface File
Memory Initialization File
SignalTap II Logic Analyzer File
Synopsys Design Constraints File
Tcl Script File
Text File
Vector Waveform File

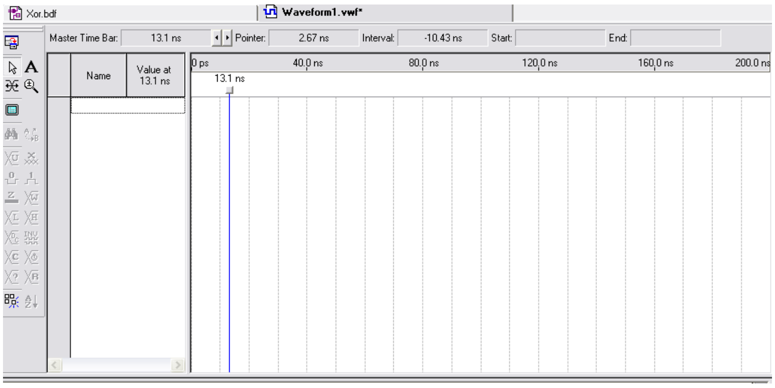
Figure 26. Need to prepare a new file.

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.



Master Time Bar: 13.1 ns Pointer: 500 ps Interval: -12.6 ns Start: End:

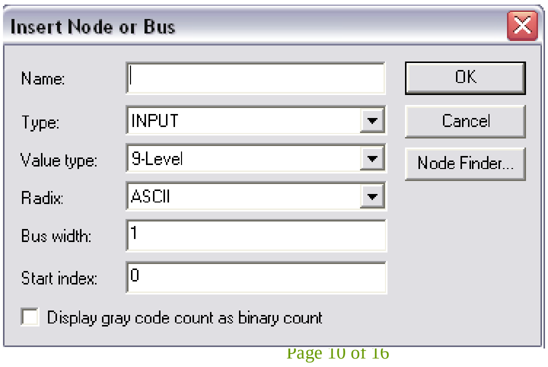
Figure 27. The Waveform Editor window.



Master Time Bar: 13.1 ns Pointer: 2.67 ns Interval: -10.43 ns Start: End:

Figure 28. The augmented Waveform Editor window.

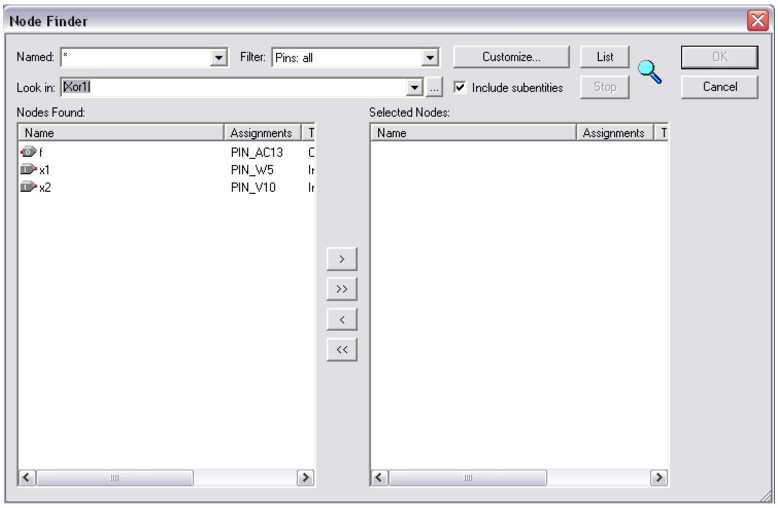
3. Next, we want to include the input and output nodes of the circuit to be simulated. Click **Edit > Insert> Insert Node or Bus** to open the window in Figure 29.



Name:
Type: INPUT
Value type: 9-Level
Radix: ASCII
Bus width: 1
Start index: 0
□ Display gray code count as binary count

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.



Node Finder

Named: * Filter: Pins: all Customize... List OK

Look in: |Xor1| ☑ Include subentities Stop Cancel

Nodes Found:

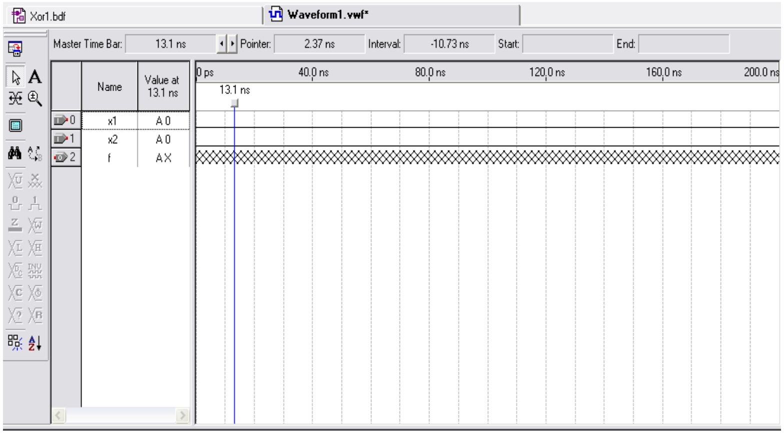
Name	Assignments	T
f	PIN_AC13	O
x1	PIN_W5	I
x2	PIN_V10	I

Selected Nodes:

Name	Assignments	T

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.



Xor1.bdf Waveform1.vwf*

Master Time Bar: 13.1 ns Pointer: 2.37 ns Interval: -10.73 ns Start: End:

	Name	Value at 13.1 ns
0	x1	A 0
1	x2	A 0
2	f	A X

Figure 31. The nodes needed for simulation.

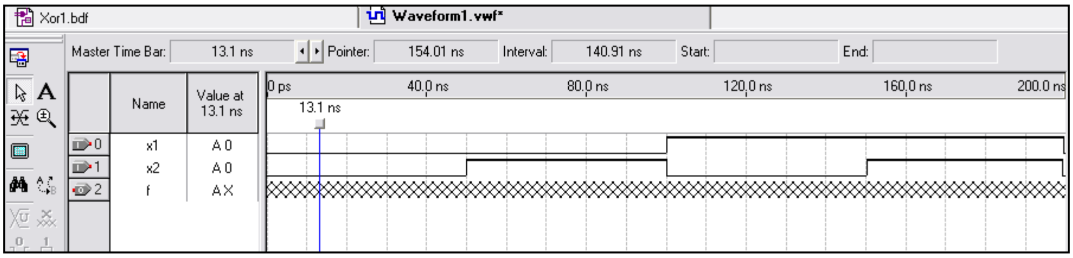
4. We will now specify the logic values to be used for the input signals x1 and x2 during simulation. The logic values at the output f will be generated automatically by the simulator. To make it easy to draw the desired waveforms, the Waveform Editor displays (by default) vertical guidelines and provides 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**.



Master Time Bar: 13.1 ns
Pointer: 154.01 ns
Interval: 140.91 ns
Start:
End:

	Name	Value at 13.1 ns
0	x1	A 0
1	x2	A 0
2	f	A X

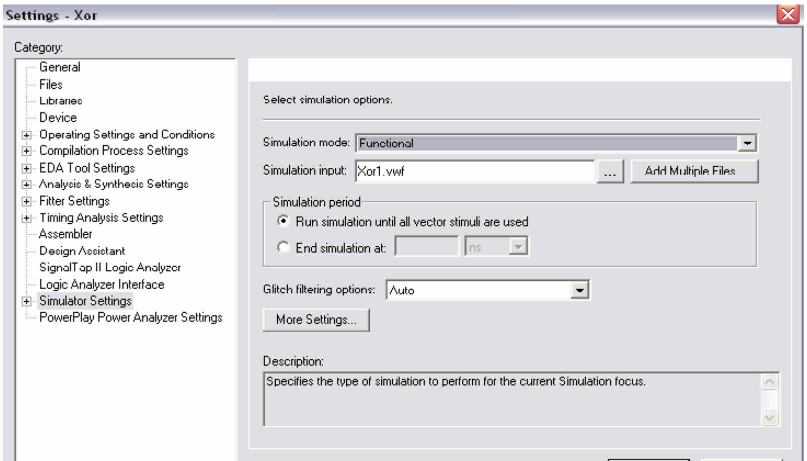
Figure 32. Setting of test values.

5.1 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 simulatio**n. 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.

5.1.1 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.



Settings - Xor

Category:
- General
- Files
- Libraries
- Device
- Operating Settings and Conditions
- Compilation Process Settings
- EDA Tool Settings
- Analysis & Synthesis Settings
- Fitter Settings
- Timing Analysis Settings
- Assembler
- Design Assistant
- SignalTap II Logic Analyzer
- Logic Analyzer Interface
- Simulator Settings
- PowerPlay Power Analyzer Settings

Select simulation options.

Simulation mode: Functional
Simulation input: Xor1.vwf

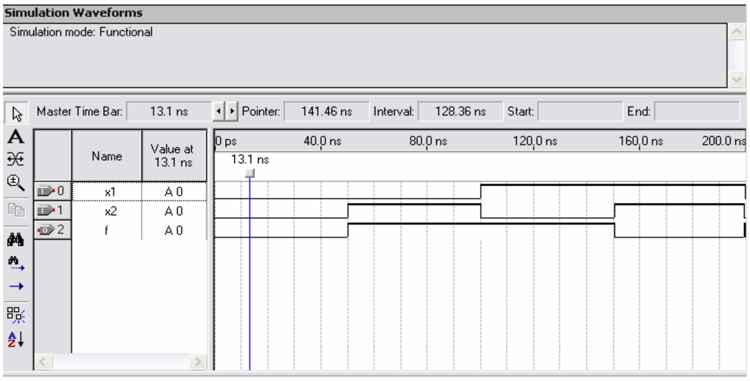
Simulation period
● Run simulation until all vector stimuli are used
○ End simulation at: ns

Glitch filtering options: Auto

Description:
Specifies the type of simulation to perform for the current Simulation focus.

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.



Simulation Waveforms
Simulation mode: Functional

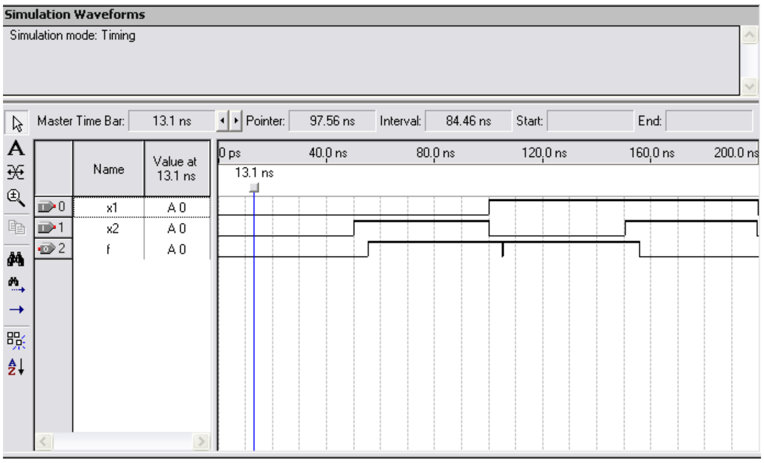
Master Time Bar: 13.1 ns | Pointer: 141.46 ns | Interval: 128.36 ns | Start: | End:

	Name	Value at 13.1 ns
0	x1	A 0
1	x2	A 0
2	f	A 0

Figure 34. The result of functional simulation.

5.1.2 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.



Simulation Waveforms
Simulation mode: Timing

Master Time Bar: 13.1 ns | Pointer: 97.56 ns | Interval: 84.46 ns | Start: | End:

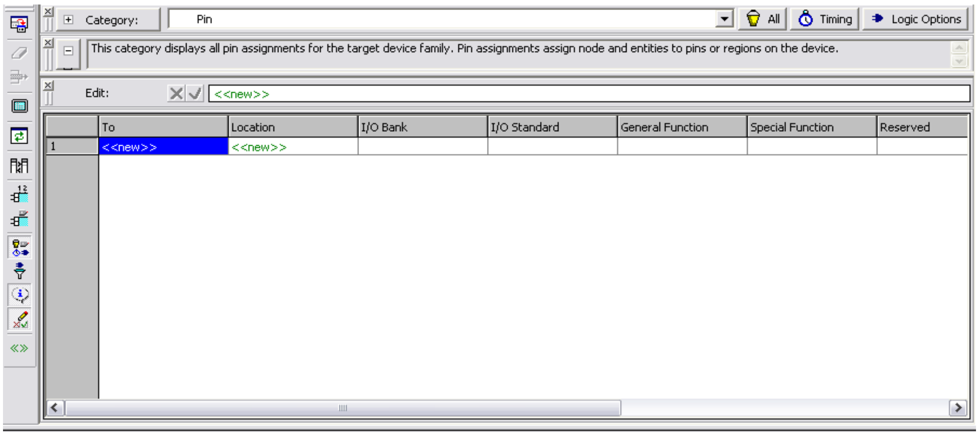
	Name	Value at 13.1 ns
0	x1	A 0
1	x2	A 0
2	f	A 0

Figure 35. The result of timing simulation.

6- 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 **LEDR10**, 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.



Category: Pin

This category displays all pin assignments for the target device family. Pin assignments assign node and entities to pins or regions on the device.

Edit: <<new>>

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
1	<<new>>	<<new>>					

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.

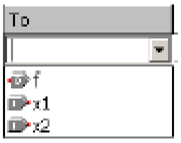
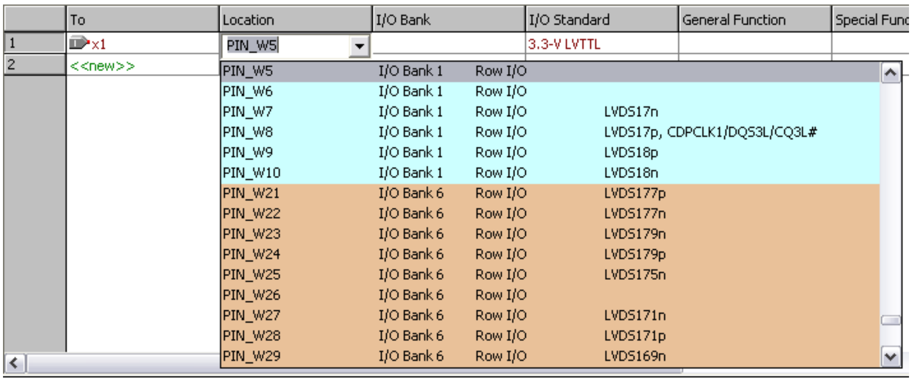


Figure 23. The drop-down menu displays the input and output names.

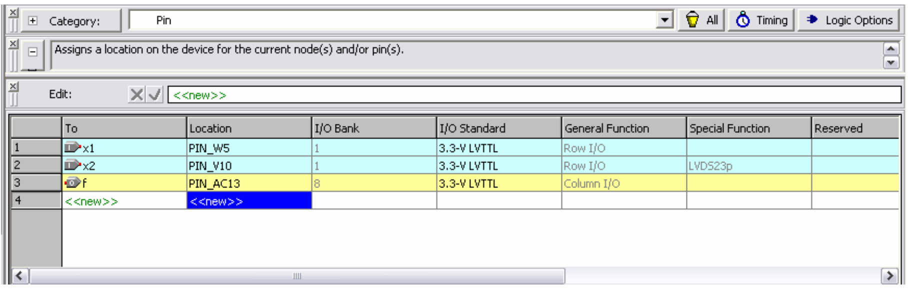
Scroll down and select **PIN_W5**. Instead of scrolling down the menu to find the desired pin, you can just **type the name of the pin (W5)** in the Location box. Use the same procedure to assign input x2 to pin V10 and output f to pin AC13, which results in the image in Figure 25.



	To	Location	I/O Bank	I/O Standard	General Function	Special Func
1	x1	PIN_W5		3.3-V LVTTL		
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/DQS3L/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.



Category: Pin

Assigns a location on the device for the current node(s) and/or pin(s).

Edit: <<new>>

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
1	x1	PIN_W5	1	3.3-V LVTTL	Row I/O		
2	x2	PIN_V10	1	3.3-V LVTTL	Row I/O	LVDS23p	
3	f	PIN_AC13	8	3.3-V LVTTL	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.

7- Programming and Configuring the FPGA Device

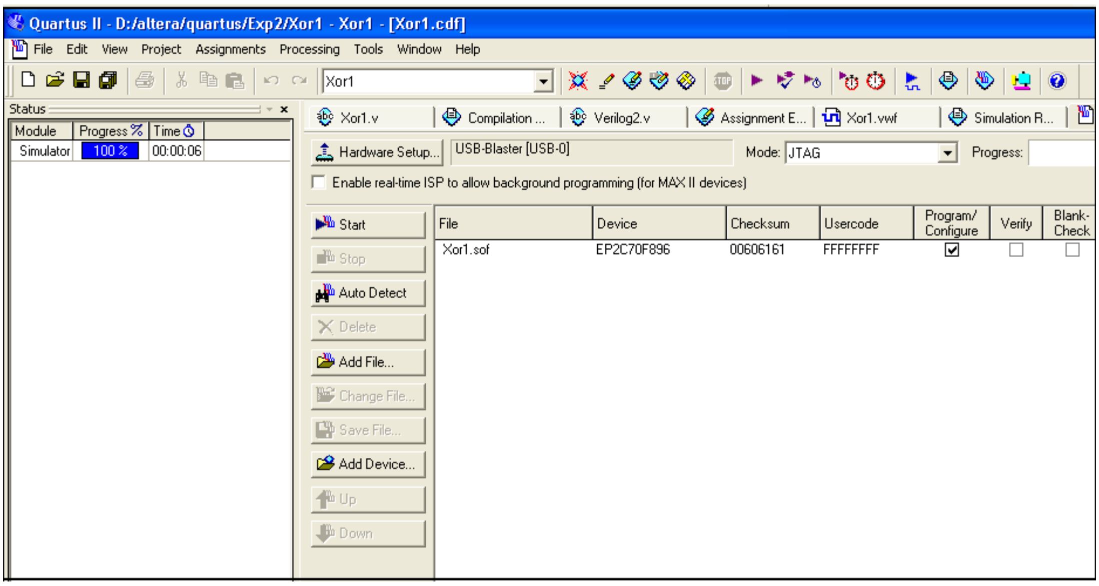
The FPGA device must be programmed and configured to implement the designed circuit. The required configuration file is generated by the Quartus II Compiler's Assembler module. Altera's DE2 board allows the configuration to be done in two different ways, known as JTAG and AS modes. The configuration 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.**



Quartus II - D:/altera/quartus/Exp2/Xor1 - Xor1 - [Xor1.cdf]

Module	Progress %	Time
Simulator	100 %	00:00:06

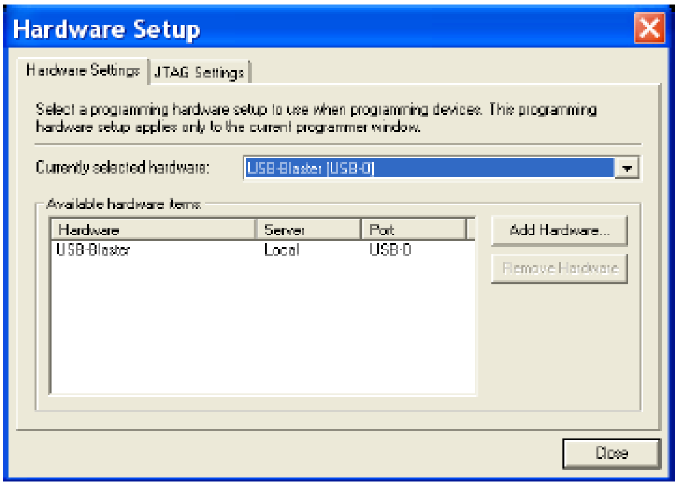
Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress:

□ Enable real-time ISP to allow background programming (for MAX II devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check
Xor1.sof	EP2C70F896	00606161	FFFFFFFF	☑	□	□

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.



Select a programming hardware setup to use when programming devices. This programming hardware setup applies only to the current programmer window.

Currently selected hardware: USB-Blaster [USB-0]

Available hardware items

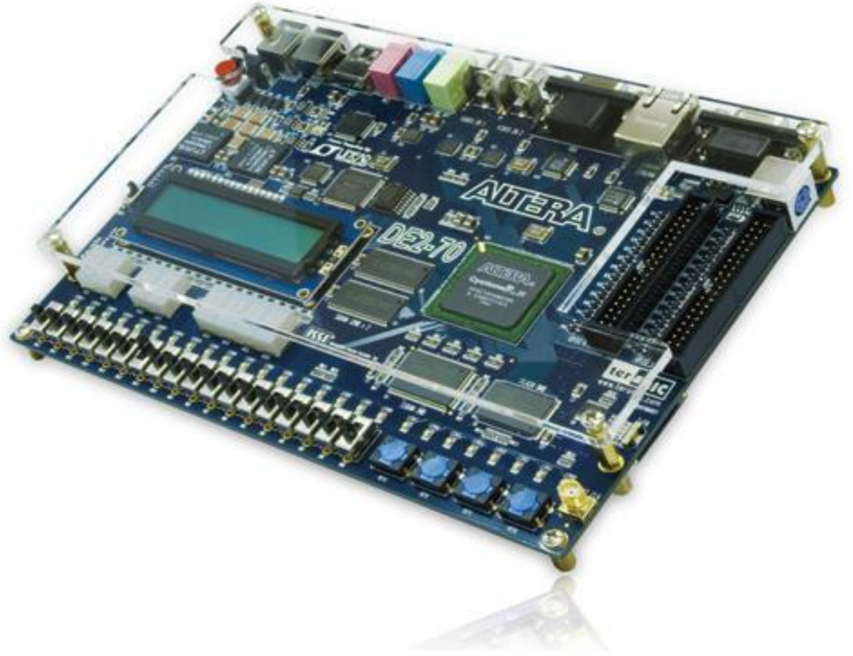
Hardware	Server	Port
USB-Blaster	Local	USB-0

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.

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

Labsheet 1: Introduction to Altera and Schematic Programming



Name:

Student ID:

Section:

Part 1: (In lab)

You are required to build the schematic diagram for the following function:

$$F(X,Y) = \overline{X.Y}$$

You have to follow the instructions below:

1. Create a new project.
2. Build the schematic of the function.

Paste a snap-shot of your schematic here:

3. Compile the project.
4. Perform **Functional** simulation for the function.

Paste a snap-shot of your entire simulation here:

Part 2: (In lab)

You are required to build the schematic diagram for the following function:

$$F(A, B, C) = (A.C) + B$$

You have to follow the instructions below:

1. Create new project.
2. Build the schematic of the function.

Paste a snap-shot of your schematic here:

3. Compile the project.
4. Perform **Timing** simulation for the function.

Paste a snap-shot of your entire simulation here:

5. Perform the following pin assignments then download your design on the FPGA.

    - A : SW [3] (Pin_AC27)
    - B : SW [2] (Pin_AB25)
    - C : SW [1] (Pin_AB26)
    - F : LEDR [0] (Pin_AJ6)

6. For the following switch combinations, determine the status of the LEDR[0] as you see it on the FPGA to verify the correctness of your design:

SW[3]	**SW[2]**	**SW[1]**	**LEDR[0] (On or Off)**
0	0	1	
0	1	1	
1	0	1	

Experiment 2
Introduction to Verilog Programming using Quartus II software

Prepared by: Eng. Shatha Awawdeh, Eng.Eman Abu_Zaitoun

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., **if**s, **while**s, **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.

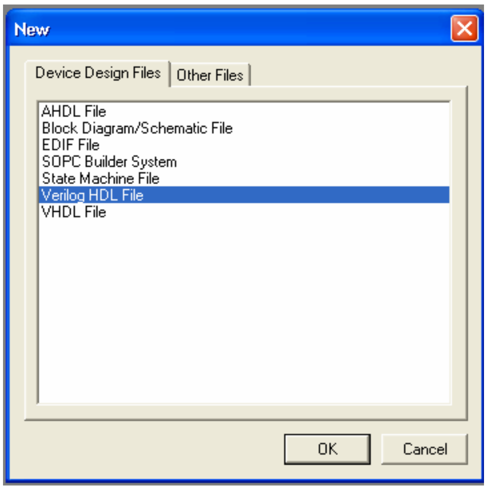


Figure (1)

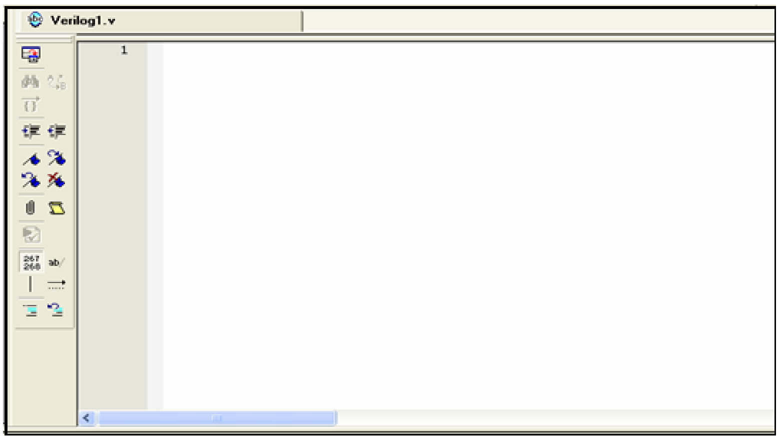
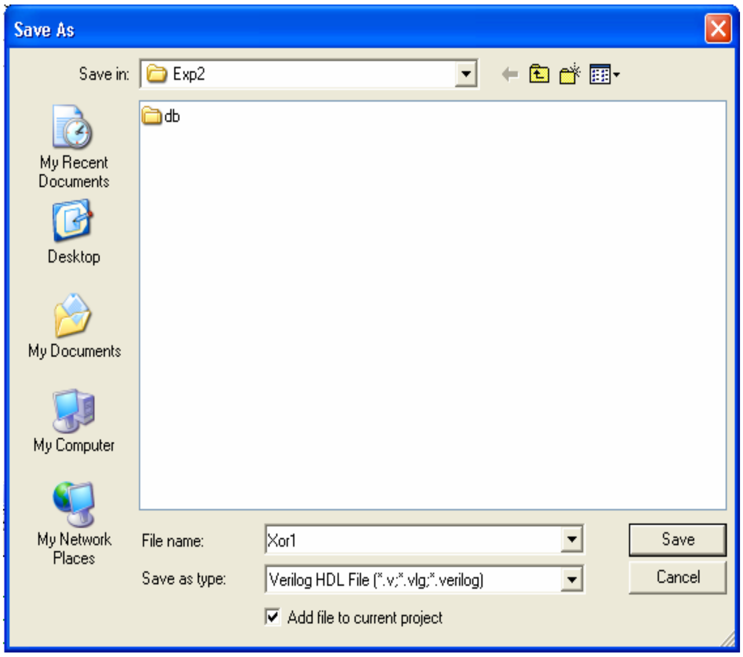


Figure (2)

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).



File name: Xor1

Save as type: Verilog HDL File (*.v;*.vlg;*.verilog)

☑ Add file to current project

Figure (3)

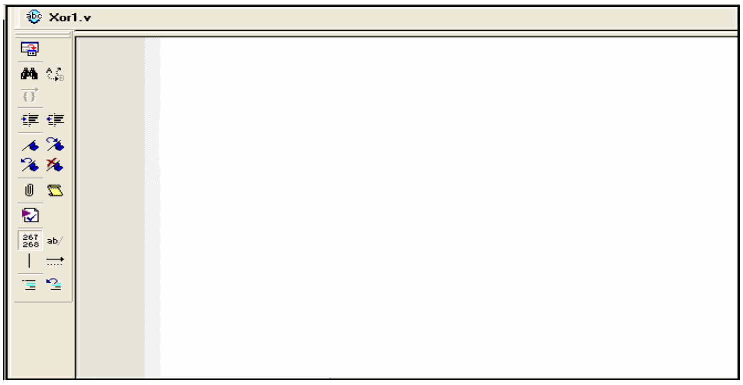
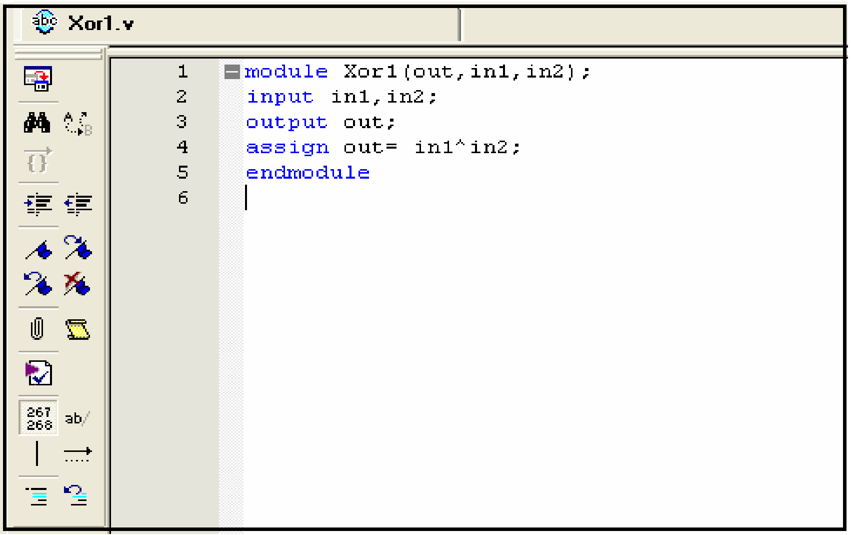


Figure (4)

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.



```
module Xor1(out,in1,in2);
input in1,in2;
output out;
assign out= in1^in2;
endmodule
```

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.

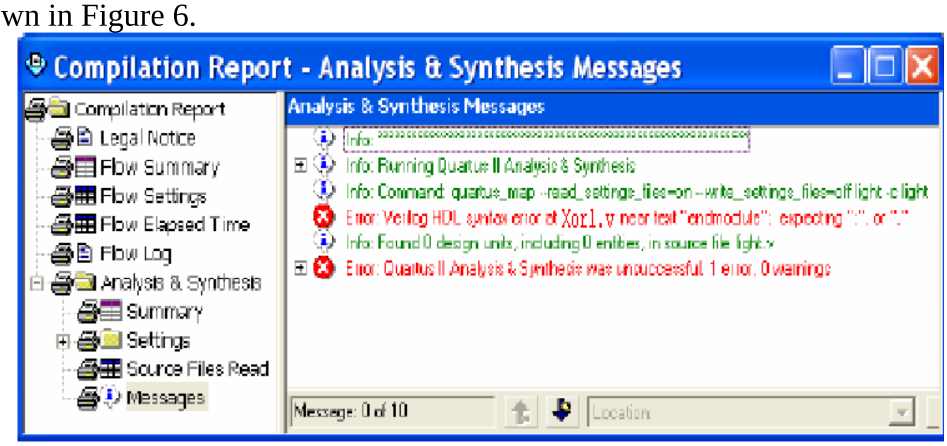
Errors

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.

.



```
Info: ***********************************************************
Info: Running Quartus II Analysis & Synthesis
Info: Command: quartus_map --read_settings_files=on --write_settings_files=off light -c light
Error: Verilog HDL syntax error at Xor1.v near text "endmodule"; expecting ";", or ","
Info: Found 0 design units, including 0 entities, in source file light.v
Error: Quartus II Analysis & Synthesis was unsuccessful. 1 error, 0 warnings
```

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)

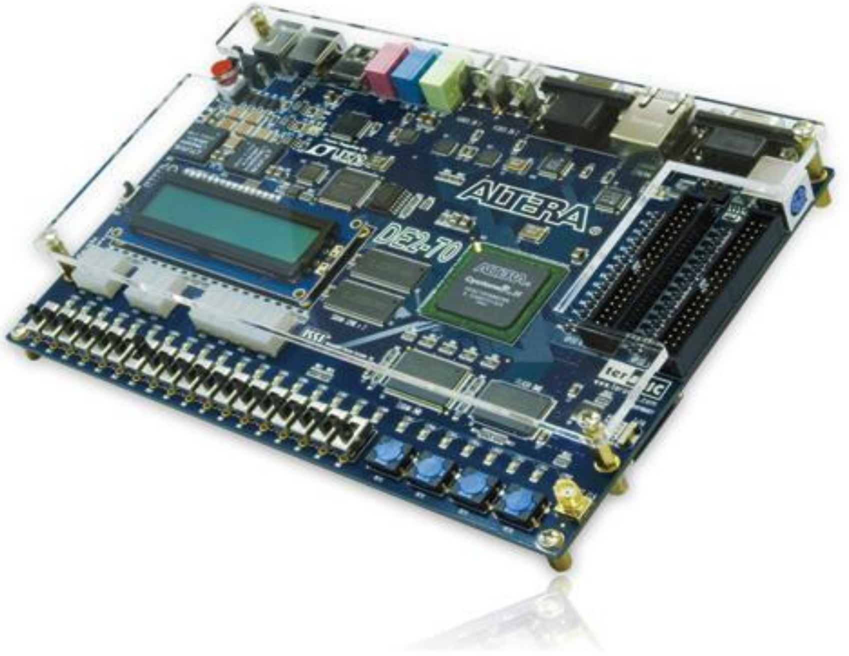
```
module Xor1(in1, in2, out);
input in1, in2;
output out;
wire w1,w2,w3,w4;

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.

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

Labsheet 2: Introduction to Verilog Programming using Quartus II software



Name:

Student ID:

Section:

Part1: (In Lab)

You are required to use Verilog **structural** modeling to design function F given by the following equation:

$$F(x,y,z) = (x.y.z) + (x \oplus z)$$

Follow the instructions below:

1. Open Quartus II software and create a new project.

2. Create and add a Verilog file named "lib.v" to your project.

3. In the file "lib.v", write modules for 2-input AND gate, 2-input OR gate, and 2-input XOR gate using behavioral modeling.

4. Create and add a Verilog file named "circuit1.v" to your project.

5. In the file "circuit1.v", write a module named "circuit1" that implements function F using structural modeling.

6. Set the top-level entity to be "circuit1.v" (Assignment → settings → general) or (from Files tab in project navigator right click on "circuit1.v" file → set as top level entity).

7. Compile your project and run functional simulation that shows **all possible input combinations**.

8. Perform the following pin assignments then download and test your design on the FPGA:

   - x: SW [3] → (Pin_AC27)
   - y: SW [2] → (Pin_AB25)
   - z: SW [1] → (Pin_AB26)
   - F: LEDR [0] → (Pin_AJ6)

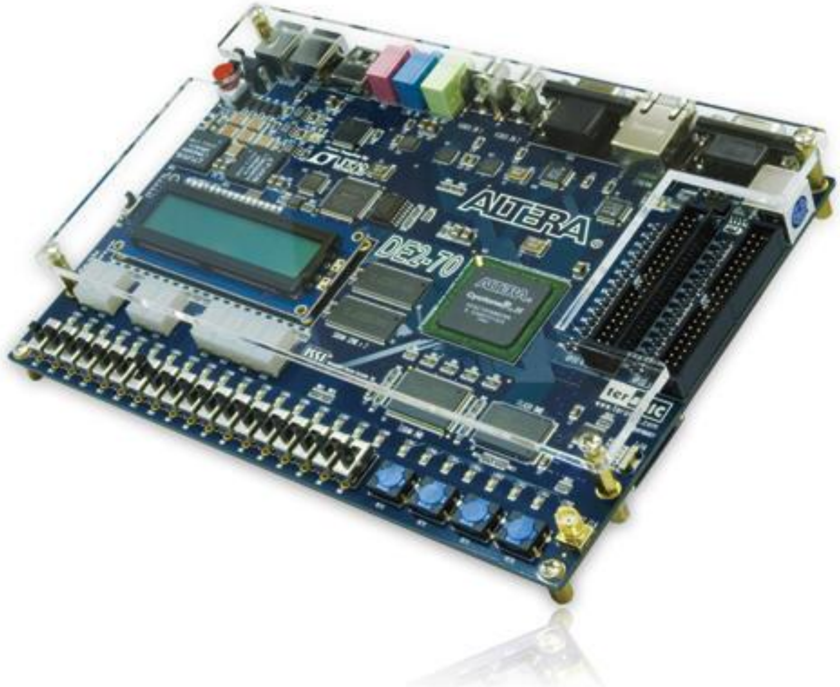
Copy your "lib.v" code here:

Copy your “circuit1.v” code here:

Paste a snap-shot of your entire simulation here:

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

Labsheet 3: Basic Logic Gates Implementation Using Breadboards and Discrete Gates



Name:

Student ID:

Section:

Part 1: Check Elementary Functions:

1- Insert the Quad 2-input AND gate IC into the bread-boarding socket, connect pin 14 to +5V and pin 7 to GND. Experimentally verify that this AND gate is working properly by determining its truth table. **(In Lab)**

Input		Output
i1	i2	F
0	0	
0	1	
1	0	
1	1	

2- Can a NOR gate be used as an inverter? How?

3- Can a NAND gate be used as a buffer? How?

Part 2: Simple Design:

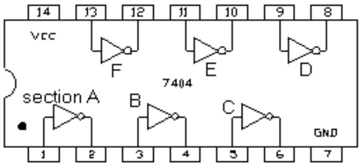
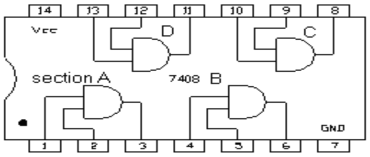
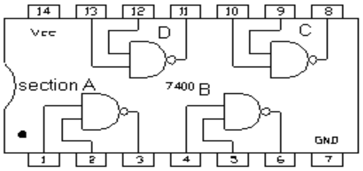
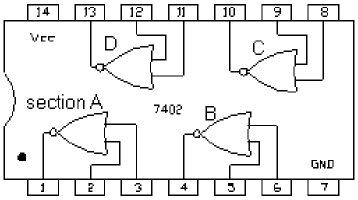
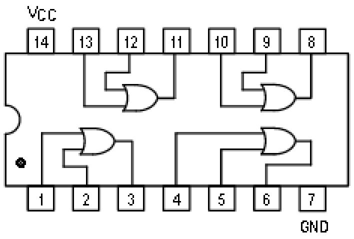
1- Build function **F** circuit on your breadboard and construct the truth table based on the logic outputs for every possible input: **(In lab)**

$F = \overline{(XY)}.(X + Y)$

Input		Output
X	Y	F
0	0	
0	1	
1	0	
1	1	

2- Which gate function does F represent?

Use the following table when needed.

Gate	IC Code	Pin Configuration
Hex-Inverter	74LS04	 14 13 12 11 10 9 8 Vcc F E D 7404 section A B C GND 1 2 3 4 5 6 7
Quad 2-input AND gate	74LS08	 14 13 12 11 10 9 8 Vcc D C section A 7408 B GND 1 2 3 4 5 6 7
Quad 2-input NAND gate	74LS00	 14 13 12 11 10 9 8 Vcc D C section A 7400 B GND 1 2 3 4 5 6 7
Quad 2-input NOR gate	74LS02	 14 13 12 11 10 9 8 Vcc D C section A 7402 B GND 1 2 3 4 5 6 7
Quad 2-input OR gate	74LS32	

Experiment 3
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 semiconductors 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.

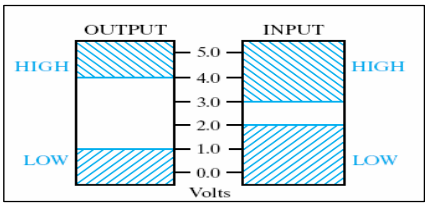


Figure 1

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.

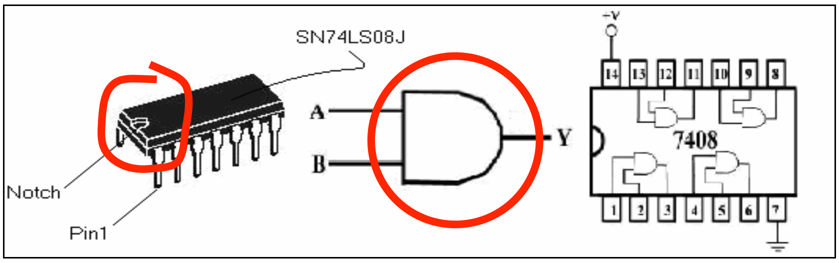


Figure 2

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.

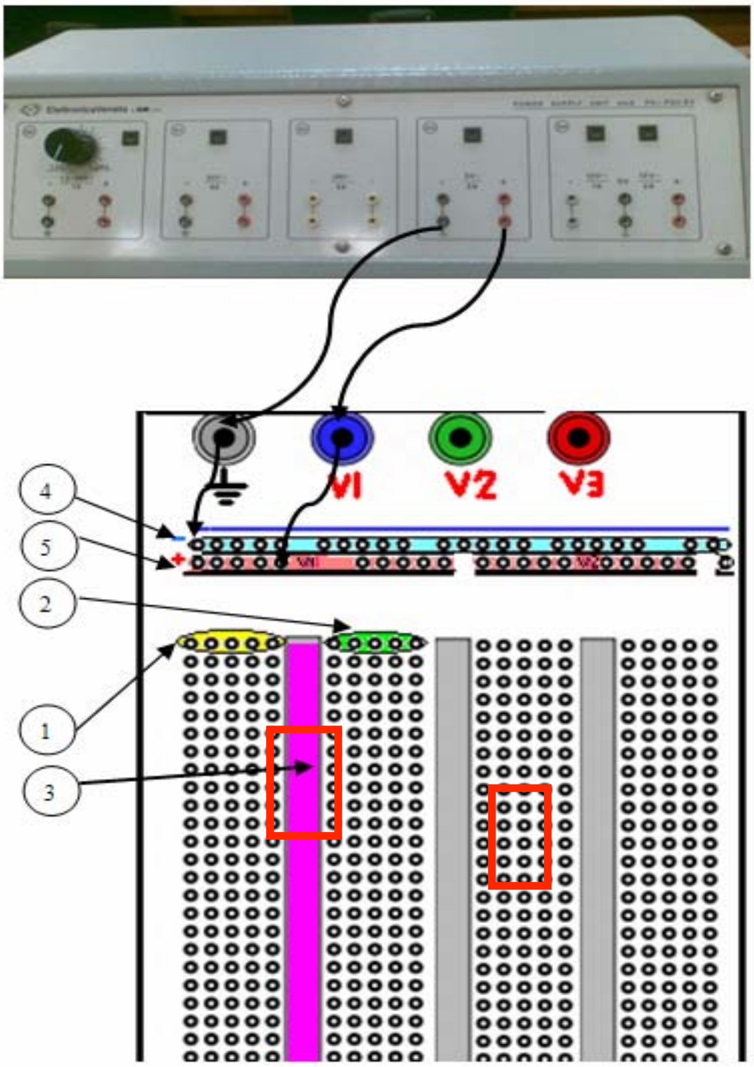


Figure 3

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_{cc} 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.

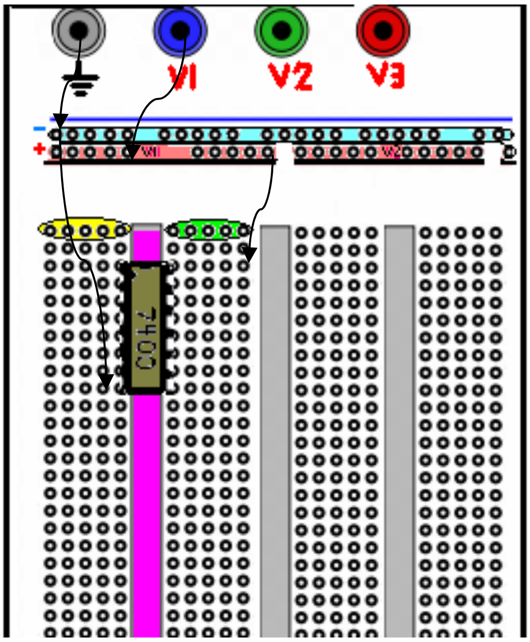


Figure 4

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.**

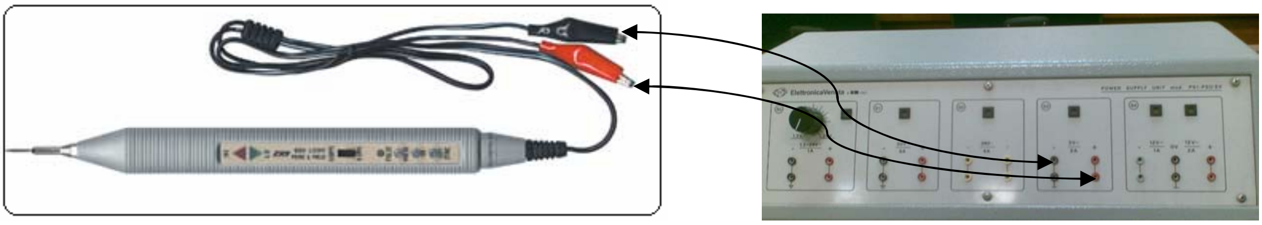


Figure 5

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.

Experiment 4
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:

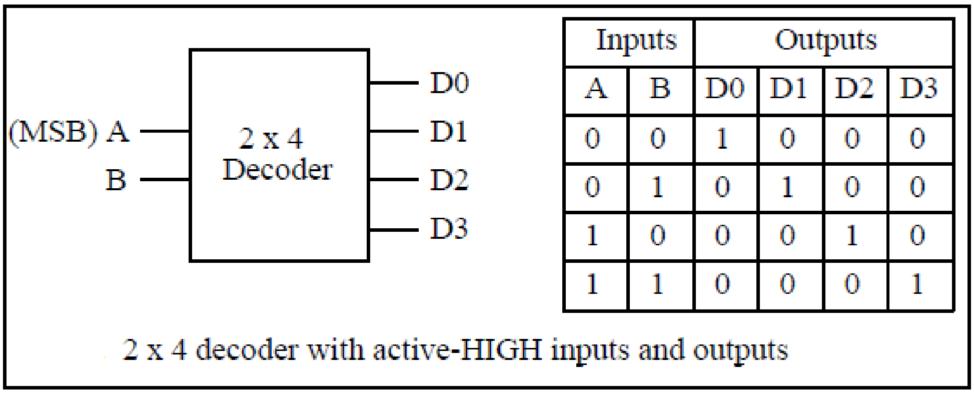
- Design, build, and test a variety of Decoders, Encoders.
- Demonstrate the operations and applications of Decoders, Encoders.
- Implement logic functions using Decoders.

Decoder:

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

- A decoder has n inputs and m outputs, where m ≤ 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.

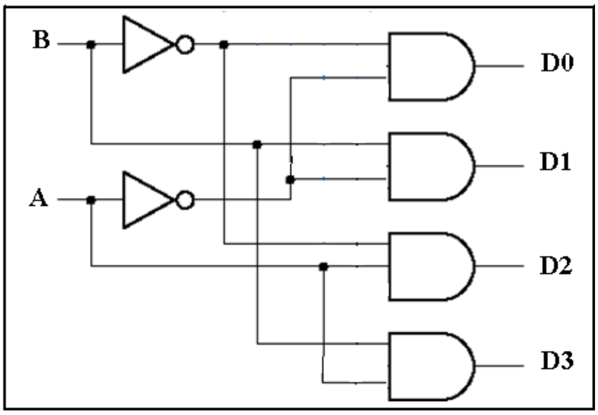


Inputs		Outputs			
A	B	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

2 x 4 decoder with active-HIGH inputs and outputs

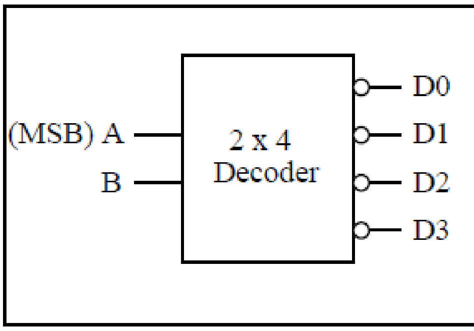
Figure(1)

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



Figure(2)

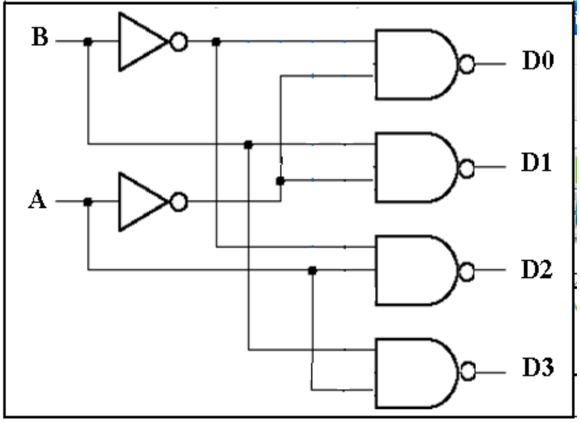
Some decoders,have active-LOW outputs. Figure below shows a block diagram and a truth table for a 2 x 4 decoder with active-LOW outputs.



Inputs		Outputs			
A	B	D0	D1	D2	D3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Figure(3)

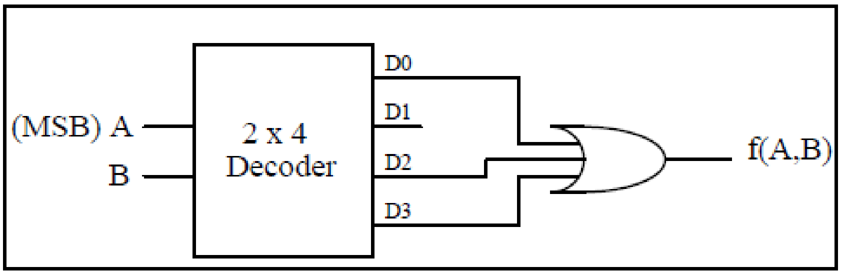
Logic Diagram of 2 x 4decoder with active-LOW outputs:



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) =Σ(0, 2, 3) then f (A,B)= D0 + D2 + D3 so f can be implemented by the circuit shown in Figure below:



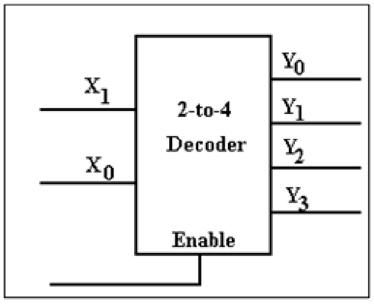
Figure(5)

➢ 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.



Enable	X_1	X_0	Y_0	Y_1	Y_2	Y_3
0	d	d	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Figure(6)

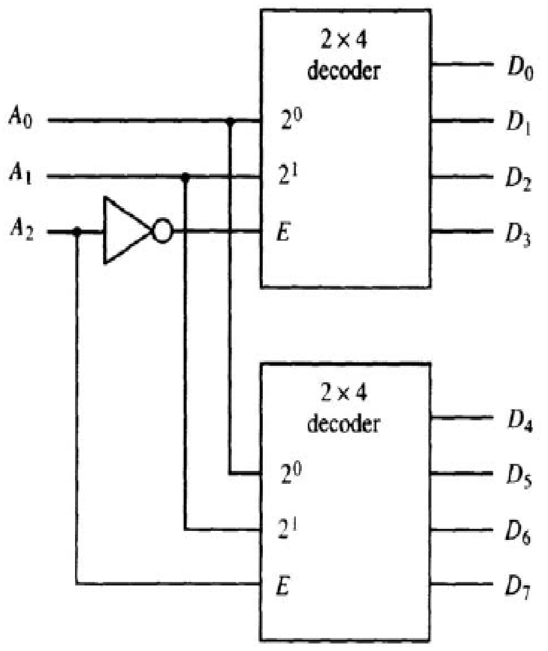
If the decoder enable signal is active low, then the decoder is active (enabled) when enable is 0 and not active (disabled) when enable = 1.

➢ 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 .

A 3 × 8 decoder constructed with two 2 × 4 decoders.

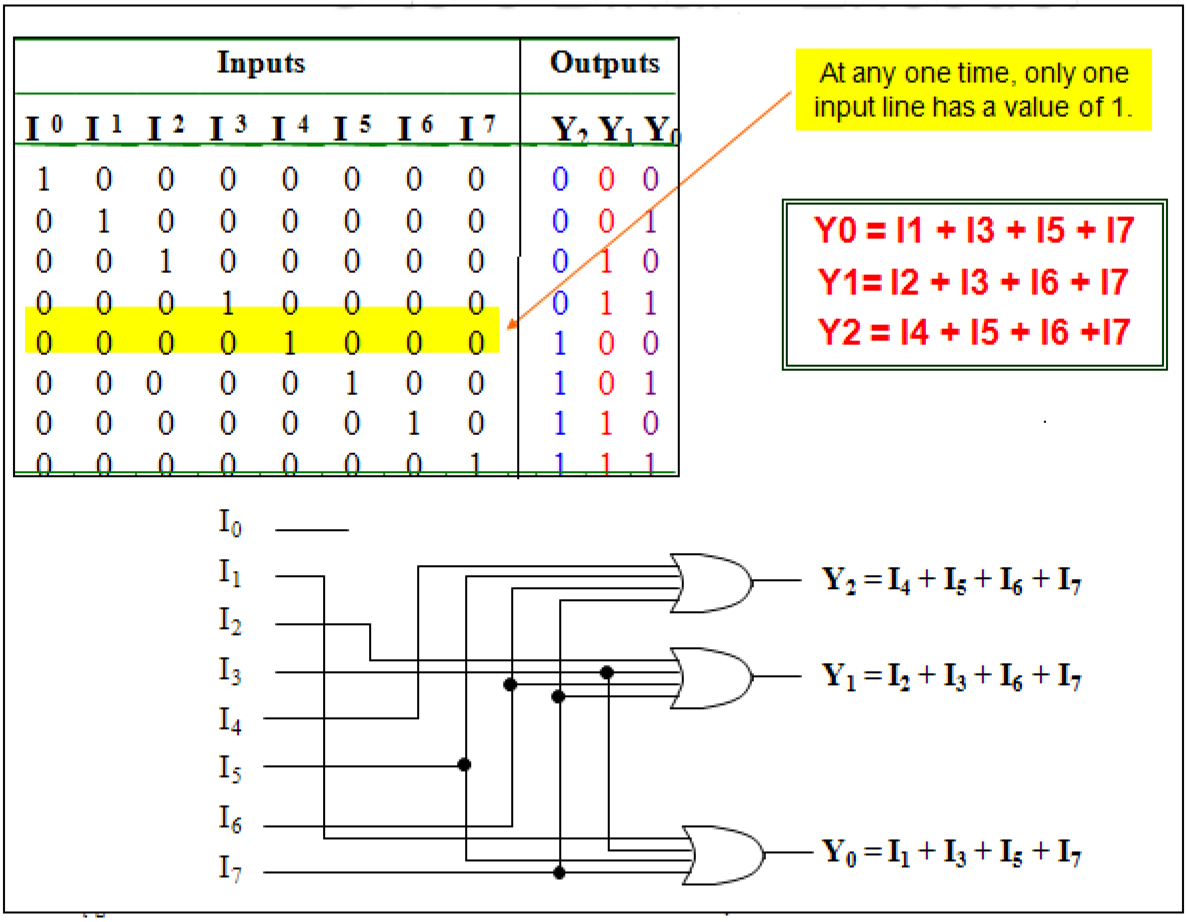


Figure(7)

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.



Inputs								Outputs		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

At any one time, only one input line has a value of 1.

Y0 = I1 + I3 + I5 + I7

Y1= I2 + I3 + I6 + I7

Y2 = I4 + I5 + I6 +I7

I_0
I_1
I_2
I_3
I_4
I_5
I_6
I_7
$Y_2 = I_4 + I_5 + I_6 + I_7$
$Y_1 = I_2 + I_3 + I_6 + I_7$
$Y_0 = I_1 + I_3 + I_5 + I_7$

Figure(8)

If we look carefully at the Encoder circuits that we got, we see that if more then 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.

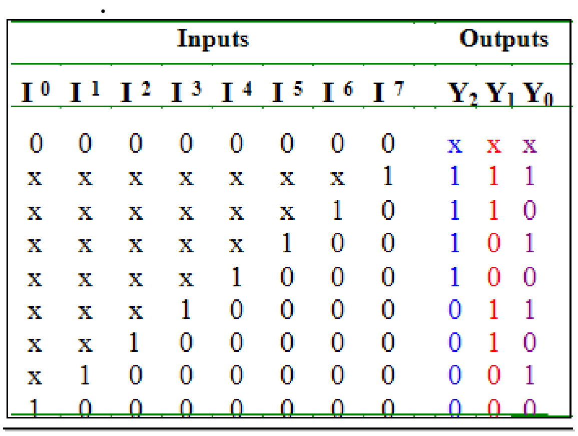
- **Priority Encoder** :
  With a priority encoder, we may have more than one input with a value of 1. How do we can decide which input subscript to encode by assign a priority to each of the subscripts.

  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_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Y_2	Y_1	Y_0
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:

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

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

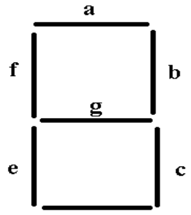
$Y_2 = I_7 + I_6 + I_5 + I_4$

- **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 I0 =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:

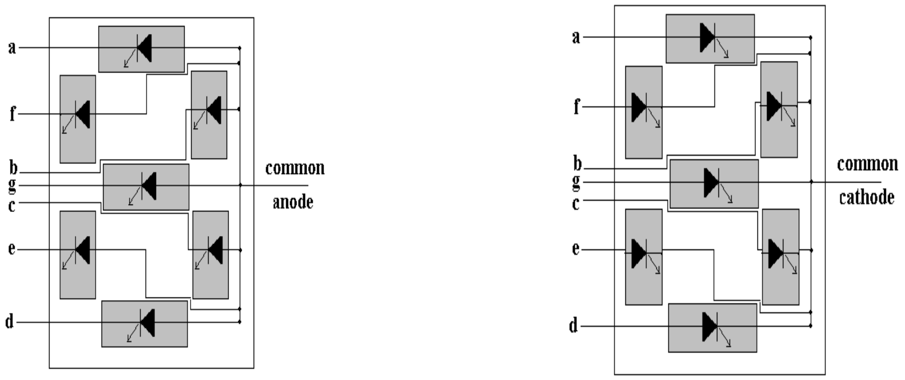
The 7-segment display consist in small bars (the segments) that, set in the way indicated in Figure below, with standard letters **a** through **g;** enable the representation of the ten decimal numbers (0-9) and some other characters.



Figure(9)

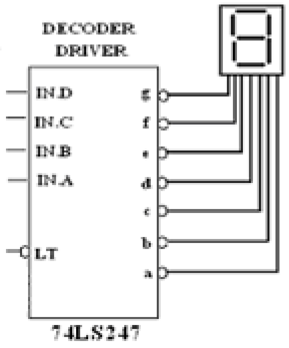
Electrically the LEDs behave like standard diodes at solid state, with the only difference that there is a higher voltage between anode and cathode, in case of direct polarization.
There are two main types of 7-segment displays (As shown in Figure below):

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



Figure(10)

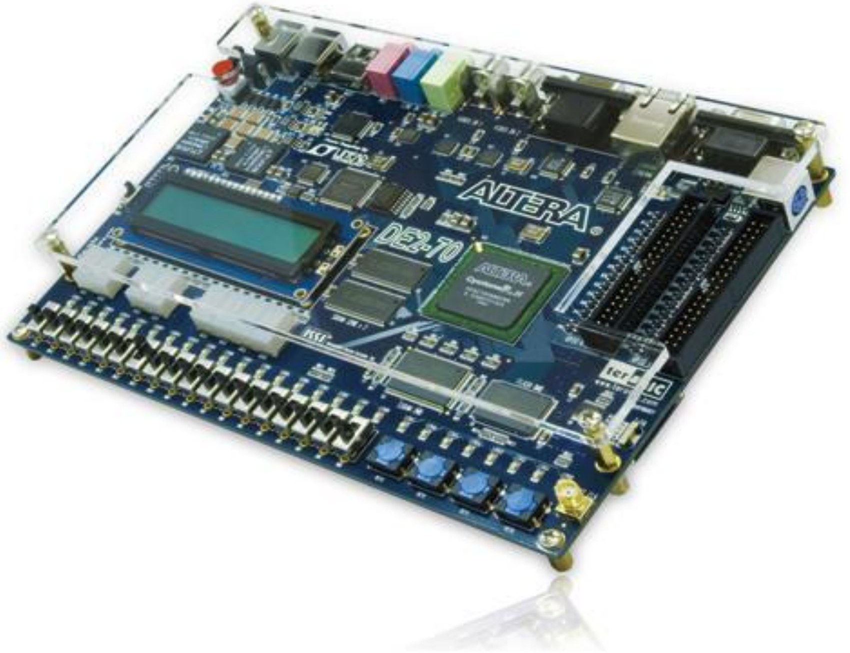
The 7-segment display device requires seven separate inputs. To use this display device, the binary code called Binary Coded Decimal (BCD) is converted to 7-segment code and supplied to the input of the display device. The circuit that performs the conversion is called a **BCD to 7-segment decoder/driver**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



Figure(11)

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

Labsheet4: Decoder and Encoder Simulation



Name:

Student ID:

Section:

Problem Description:

You are requested to design a system that monitors 7 digital devices in your home. Each of these devices continuously sends a signal to the system to report its status. The device sends logic '0' if it is working properly; otherwise, the device sends logic '1' to indicate that it is not working and needs to be fixed. The monitoring system uses two seven segment displays to show the status of these devices and the number of the faulty device, if any, as shown in Figure 1.

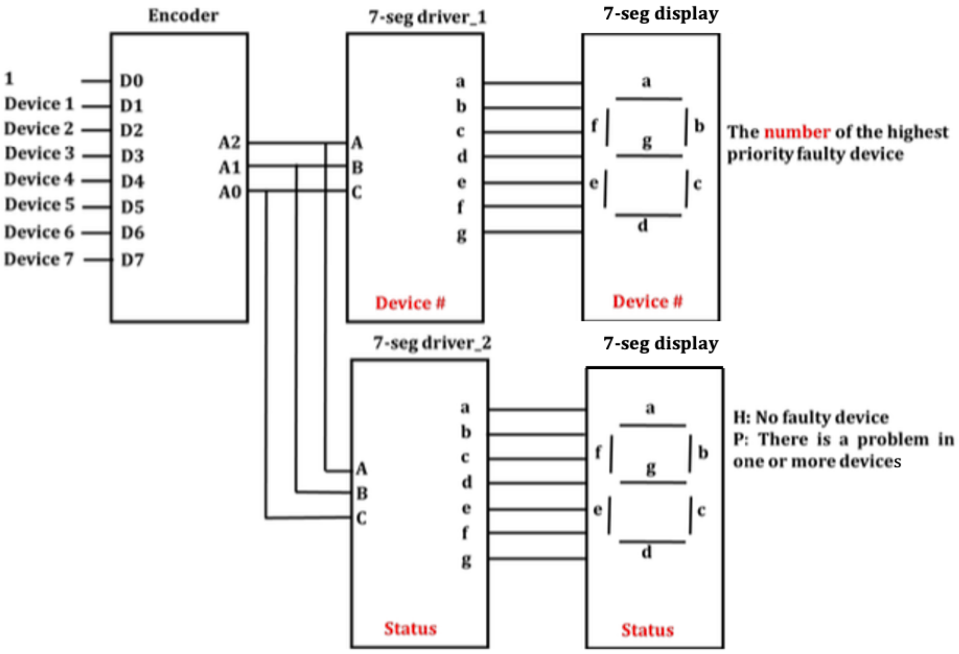


Figure 1: Monitoring System Block Diagram

The system operates as follows:

1. The signals received from the devices are connected to an 8-to-3 high priority encoder that outputs the number of the faulty device with the highest priority in binary format. For example, the encoder outputs '110' when Device 6 is the only faulty one. In case there is more than one faulty device, the encoder outputs the highest faulty device number. For instance, if devices 5 and 2 are faulty, the encoder outputs '101'. **In case none of the devices is faulty, the encoder outputs '000'**.

2. The output of the encoder is connected to a 7-segment **driver/decoder** labeled "Device #" that converts the device number into the corresponding 7-segment code. The output of this driver is connected to the 7-segment **display** labeled "Device#" to show the faulty device number, if any. **The "Device#" 7-segment display is turned off when none of the devices is faulty**.

3. The output of the encoder is also connected to another 7-segment **driver/decoder** labeled "Status" which in turn is connected to a 7-segment **display** labeled "Status" which displays the overall status of the system. When there are no faulty devices, the "Status" 7-segment displays the letter "H" (i.e. the system is healthy). On the other hand, the letter "P" is displayed to indicate there is a problem in the system when there is at least one faulty device.

In order to implement the system, it is required first to implement the encoder and the 7-segment **drivers/decoders** in Verilog **behaviorally**. Second, the three components are combined structurally in the top-level module.

Part1: 8-to-3 High Priority Encoder:

a) Fill the 8-to-3 high priority encoder truth table: (Pre-Lab)

Inputs								Outputs			Output value in decimal (A2 is MSB)
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0	

b) Write the Boolean equations of outputs A2, A1 and A0: (Pre-Lab)

A2 =
A1 =
A0 =

c) Write the Verilog **behavioral** implementation of the 8-to-3 high priority encoder in the "**encoder.v**" file. This module should have eight inputs (i.e. D7-D0) and three outputs (A2-A0):

Paste your “encoder.v” code here:

d) Set "**encoder.v**" as top-level entity and perform functional simulation of the 8-to-3 high priority encoder. The simulation report should show the **eight input combinations** given in the above truth table and the outputs should be **combined together with A2 as the MSB and format is decimal**.

Paste a snapshot of your “encoder.v” simulation report here:

Part2: "Device #" 7-segment Driver/Decoder:

a) Read the 7-segment display appendix at the end of the labsheet.
b) Fill the truth table for the "**Device #**" 7-segment **driver/decoder** given in Figure 1 assuming **common-anode** 7-segment display:

Inputs			Outputs							Output value in hexadecimal (a is MSB)
A	B	C	a	b	c	d	e	f	g	
0	0	0								
0	0	1								
0	1	0								
0	1	1								
1	0	0								
1	0	1								
1	1	0								
1	1	1								

c) Write the Boolean equations of outputs a, b, c, d, e, f, and g:

a =

b =

c =

d =

e =

f =

g =

d) Write the Verilog **behavioral** implementation of the "**Device#**" 7-segment **driver/decoder** in the "**segdriver_device.v**" file. This module should have three inputs (i.e. A, B and C; where A is the MSB) and seven outputs (i.e. a, b, c, d, e, f, and g).

Paste your “segdriver_device.v” code here:

e) Set "**segdriver_device.v**" as top-level entity and perform functional simulation of the "**Device#**" 7-segment **driver/decoder**. The simulation report should include all input combinations and the outputs should be **combined together with** "**a**" **as the MSB and format is hexadecimal**.

Paste a snapshot of your “segdriver_device.v” simulation report here:

Part3: "Status #" 7-segment Driver/Decoder:

a) Fill the truth table for the "**Status**" 7-segment **driver/decoder** given in Figure 1 assuming **common-anode** 7-segment display:

Inputs			Outputs							Output value in hexadecimal (a is MSB)
A	B	C	a	b	c	d	e	f	g	
0	0	0								
0	0	1								
0	1	0								
0	1	1								
1	0	0								
1	0	1								
1	1	0								
1	1	1								

b) Write the Boolean equations of outputs a, b, c, d, e, f, and g:

a =
b =
c =
d =
e =
f =
g =

c) Write the Verilog **behavioral** implementation of the "**Status**" 7-segment **driver/decoder** in the "**segdriver_status.v**" file. This module should have three inputs (i.e. A, B and C; where A is the MSB) and seven outputs (i.e. a, b, c, d, e, f, and g).

Paste your “segdriver_status.v” code here:

d) Set "**segdriver status.v**" as top-level entity and perform functional simulation of the "**Status**" 7-segment **driver/decoder**. The simulation report should include all input combinations and the outputs should be **combined together with "a" as the MSB and format is hexadecimal**.

Paste a snapshot of your “segdriver_status.v” simulation report here:

Part4: Monitoring System:

a) Write the Verilog **structural** implementation of the overall system shown in Figure 1 in the "**circuit1.v**" file. This module should have eight inputs (i.e. D7-D0) and 14 outputs (i.e. "a, b, c, d, e, f, and g" for the "**Device#**" 7- segment driver and "a1, b1, c1, d1, e1, f1, and g1" for the "**Status**" 7-segment driver).

Paste your “circuit1.v” code here:

b) Set "**circuit1.v**" as top-level entity and perform functional simulation of the full system. The simulation report should contain the same eight input combinations in the truth table of the 8-to-3 high priority encoder. The outputs "a, b, c, d, e, f, and g" should be **combined together with "a" as the MSB and format is hexadecimal**. Similarly, the outputs "a1, b1, c1, d1, e1, f1, and g1" should be **combined together with "a1" as the MSB and format is hexadecimal**.

Paste a snapshot of your “circuit1.v” simulation report here:

c) Set the following pin assignment to the inputs and outputs in the "**circuit1.v**" file, download your design on the FPGA, and fill the following table:

Input Switches		
D1	iSW[0]	PIN_AA23
D2	iSW[1]	PIN_AB26
D3	iSW[2]	PIN_AB25
D4	iSW[3]	PIN_AC27
D5	iSW[4]	PIN_AC26
D6	iSW[5]	PIN_AC24
D7	iSW[6]	PIN_AC23

Device# 7-seg Display		
a	oHEX0_D[0]	PIN_AE8
b	oHEX0_D[1]	PIN_AF9
c	oHEX0_D[2]	PIN_AH9
d	oHEX0_D[3]	PIN_AD10
e	oHEX0_D[4]	PIN_AF10
f	oHEX0_D[5]	PIN_AD11
g	oHEX0_D[6]	PIN_AD12

Status 7-seg Display		
a1	oHEX1_D[0]	PIN_AG13
b1	oHEX1_D[1]	PIN_AE16
c1	oHEX1_D[2]	PIN_AF16
d1	oHEX1_D[3]	PIN_AG16
e1	oHEX1_D[4]	PIN_AE17
f1	oHEX1_D[5]	PIN_AF17
g1	oHEX1_D[6]	PIN_AD17

SW[6]	SW[5]	SW[4]	SW[3]	SW[2]	SW[1]	SW[0]	oHEX0	oHEX1
0	0	0	0	0	0	0		
0	0	0	1	0	0	0		
0	0	0	0	1	1	1		

Paste a snapshot of your Pin Assignment Editor window:

Appendix: 7-Segment Display

The 7-segment display consists of seven LEDs arranged as shown in Figure 2. The LEDs are indicated by the letters a, b, c, d, e, f, and g. Each LED behaves like a diode with the following two connection types:

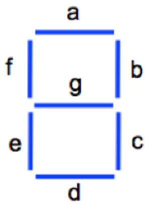


Figure 2: 7-segment Display

1. Common cathode: LEDs illuminate when positive logic is applied
2. Common anode: LEDs illuminate when negative logic is applied

By controlling the illumination of the seven LEDs, the 7-segment can display all decimal digits as follows:

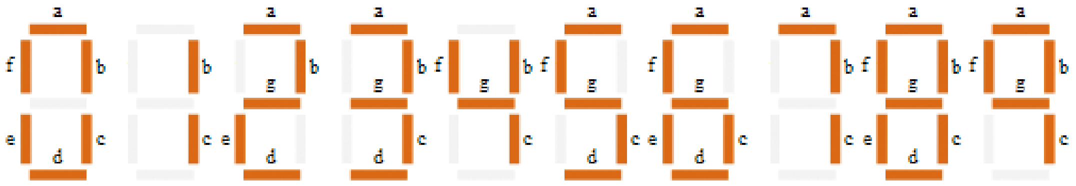


Figure 3: Displaying Decimal Digits on 7-segment Display

The 7-segment display requires a special **decoder** device called the 7-segment driver. As shown in the figure below, the input of the 7-segment driver is a 4-bit binary number that specifies which decimal/hexadecimal digit to display. The 7-segment driver produces seven 1-bit outputs that control the illumination of the seven LEDs in the 7- segment display.

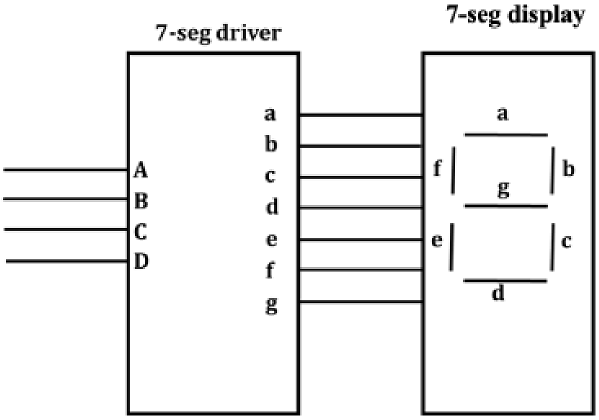


Figure 4: Connecting 7-segment Driver to 7-segment Display

Experiment 5
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.

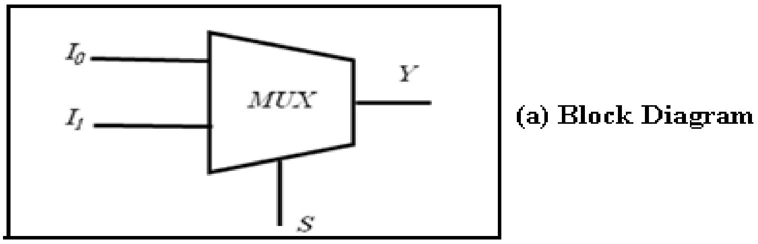
A multiplexer (MUX) performs the function of selecting the input on any one of 'n' input lines and feeding this input to one output line. Multiplexers are used as one method of reducing the number of integrated circuit packages required by a particular circuit design. This in turn reduces the cost of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

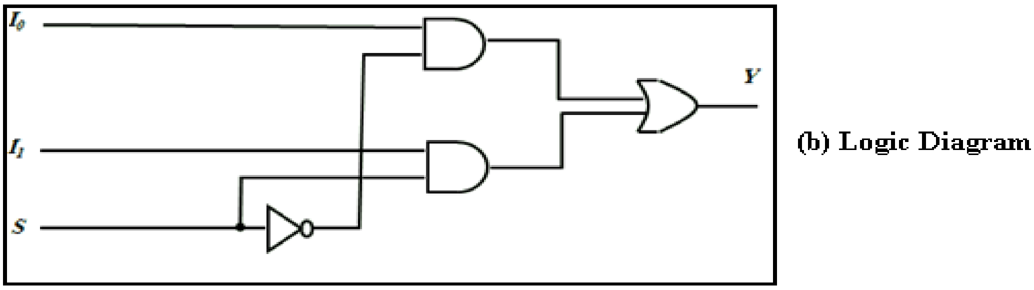
Multiplexer

- 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.



(a) Block Diagram

.



(b) Logic Diagram

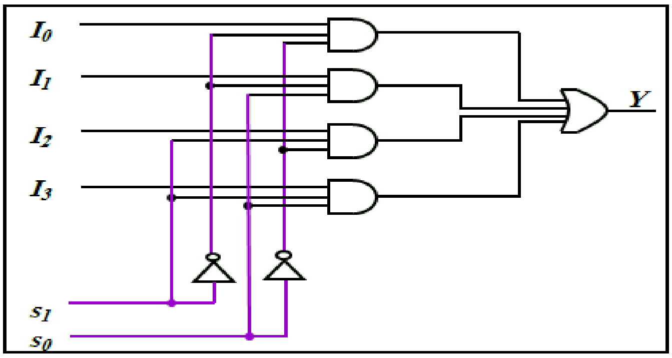
Figure(1)

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)

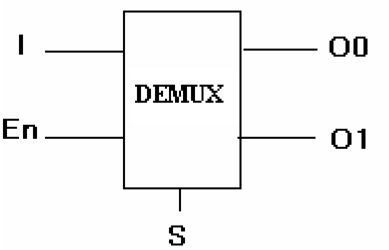
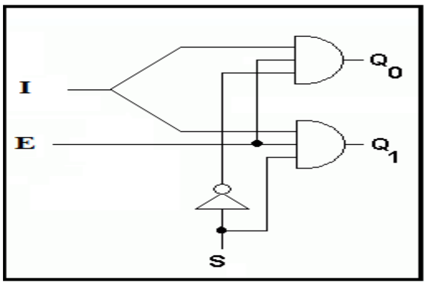


Fig 3

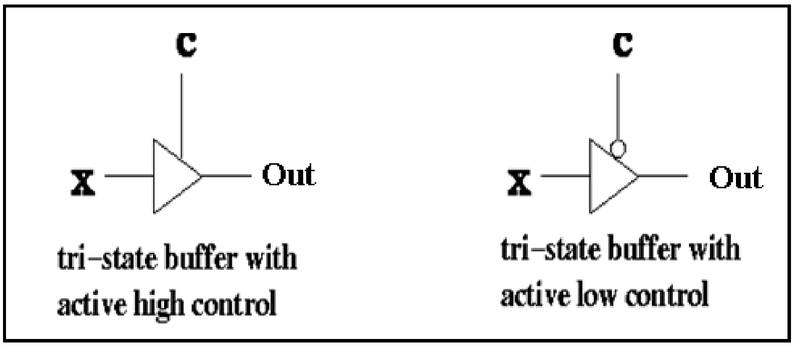
Input			Output	
I	En	S	O0	O1
X	0	X	X	X
0	1	0	0	X
1	1	0	1	X
0	1	1	X	0
1	1	1	X	1



Figure(4):Demultiplixer

Tri-state buffer

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

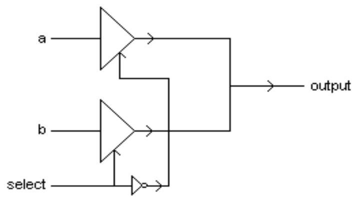
Active-low tri-state buffers

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:

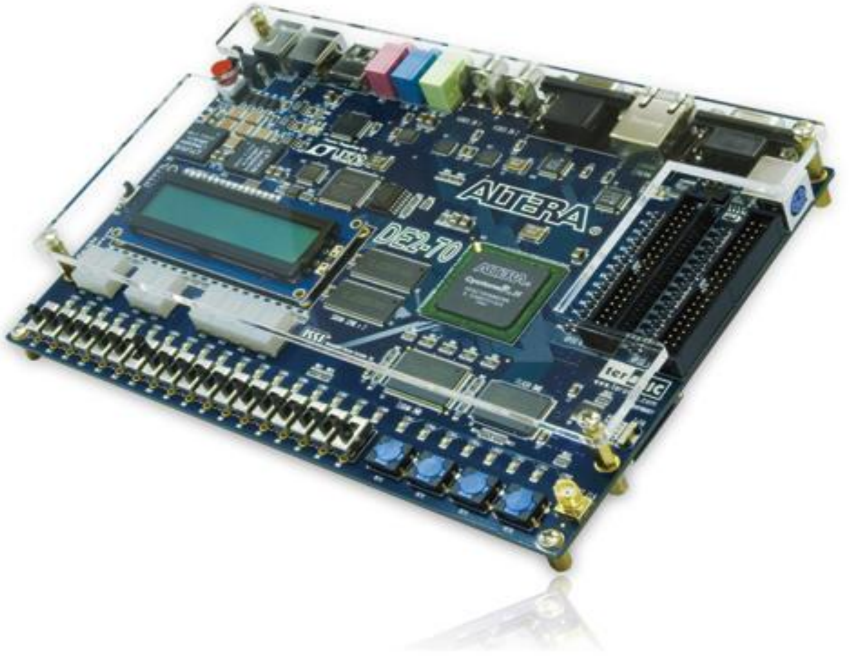
Given below is the circuit diagram of a 2 to 1 MUX implemented by a tristate buffer.



2 to 1 MUX using a Tri State Buffers

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

Labsheet5: Multiplexer and Demultiplexer Simulation



Name:

Student ID:

Section:

Problem Description:

In this experiment, you will design the two-bit time-division multiplexing (TDM) communication circuit shown in Figure 1. On the sender side, a 2-to-1 dual multiplexer is used to determine which transmitter (Tx0 or Tx1) is allowed to send data. On the receiver side, a 1-to-2 dual demultiplexer is used to determine which port (X or Y) is used to receive data. Two 7- segment displays are connected to the receivers to display the value received (in decimal) if the receiver is selected. If the receiver is not selected, the (-) sign should be displayed on its 7-segment display. Notice that port X is connected to receiver0 7-segment display and port Y is connected to receiver1 7-segment display.

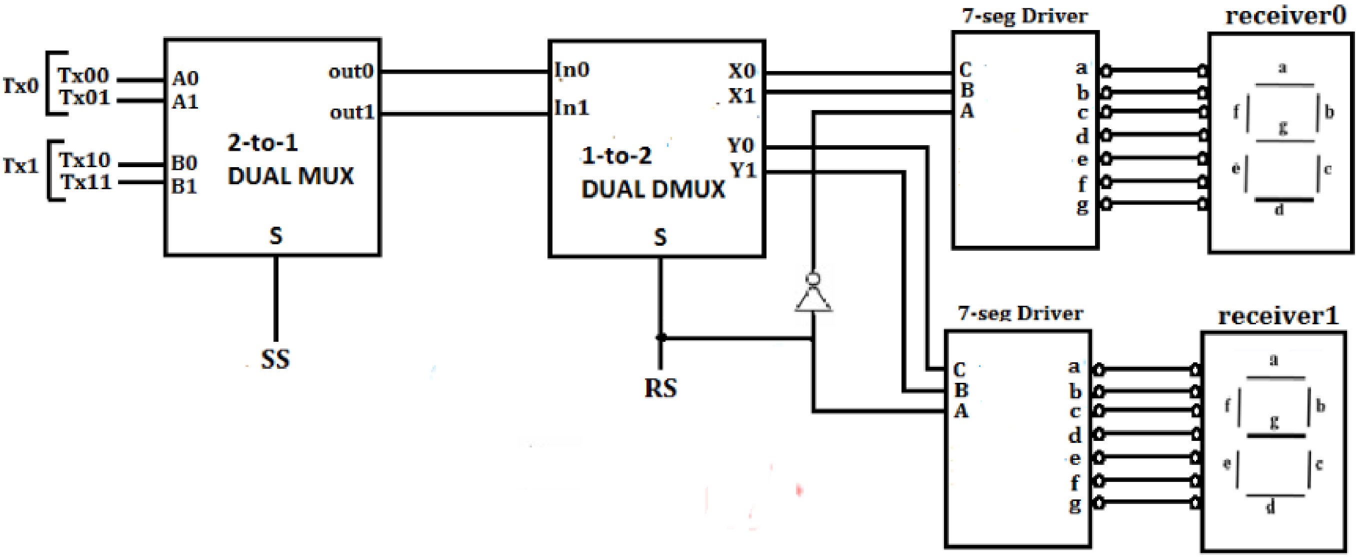


Figure 1: 2-bit TDM Communication Circuit

The experiment is divided into four parts: 2-to-1 dual multiplexer implementation, 1-to-2 dual demultiplexer implementation, 7-segment driver implementation, and top-level entity.

Part1: 2-to-1 Dual MUX

a) Fill-in the truth table for the 2-to-1 MUX shown in Figure 2. (Pre-Lab)

Inputs			Outputs
S	**I1**	**I0**	**out**

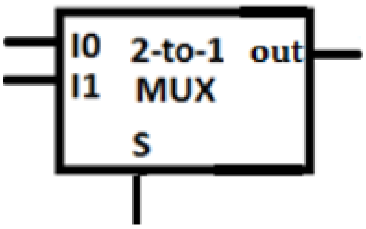


Figure 2: 2-to-1 MUX

b) Write the Boolean equation of output "out": (Pre-Lab)

out =

c) Implement the 2-to-1 MUX using **structural** Verilog code in a module named "mux1" inside the "mux1.v" file. This module has three inputs (i.e. S, I1, and I0) and one output (i.e. out). Use the gates given in the given "lib.v" file.

Paste your "mux1.v" code here:

d) Implement the 2-to-1 Dual MUX using **structural** Verilog code in a module named "dualmux" inside the "dualmux.v" file by instantiating two instances of module "mux1" connected as shown in Figure 3. This module has five inputs (A1, A0, B1, B0, and S) and two outputs (out1, out0).

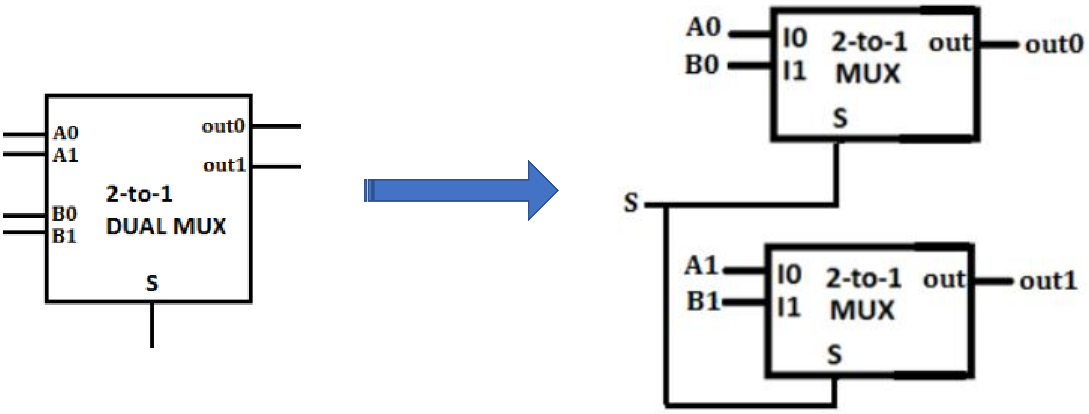


Figure 3: 2-to-1 Dual MUX

Paste your "dualmux.v" code here:

Part2: 1-to-2 Dual DMUX

a) Fill-in the truth table for the 1-to-2 DMUX shown in Figure 4. (Pre-Lab)

Inputs		Outputs	
S	D	X	Y

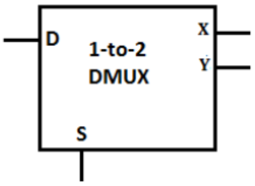


Figure 4: 1-to-2 DMUX

b) Write the Boolean equations of outputs X and Y: (Pre-Lab)

X =

Y =

c) Implement the 1-to-2 DMUX using **structural** Verilog code in a module named "dmux1" inside the "dmux1.v" file. This module has two inputs (i.e. S and D) and two outputs (i.e. X and Y). Use the gates given in the "lib.v" file.

Paste your "dmux1.v" code here:

d) Implement the 1-to-2 Dual DMUX using **structural** Verilog code in a module named "dualdmux" inside the "dualdmux.v" file by instantiating two instances of module "dmux1" connected as shown in Figure 5. This module has three inputs (In1, In0, and S) and four outputs (Y1, Y0, X1, and X0).

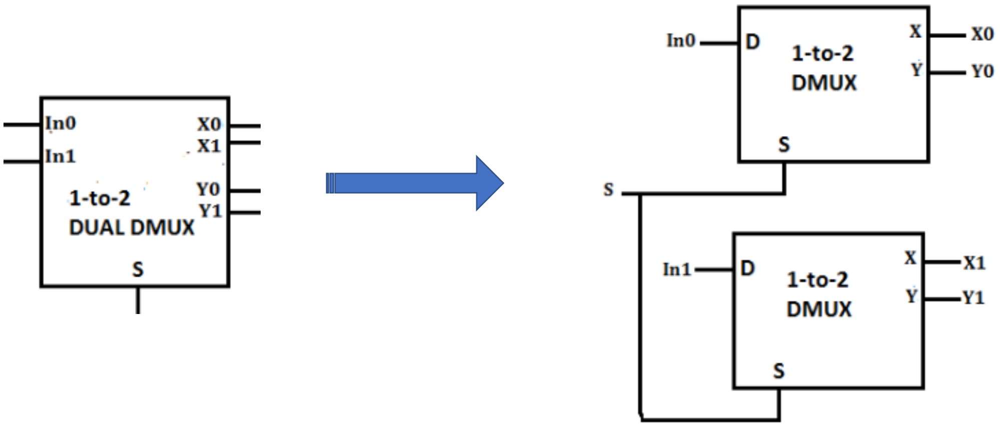


Figure 5: 1-to-2 Dual DMUX

Paste your “dualdmux.v” code here:

Part3: 7-segment Driver/Decoder

The 7-segment driver in the TDM circuit has three inputs: A, B, and C. Input “A” represents a control signal which identifies whether the receiver represented by the corresponding display is selected to receive data or not.

- If A=0, then regardless of the values of B and C, a dash symbol (-) should be displayed to indicate that this receiver is not receiving data. (Dash symbol can be generated by making segment "g" ON and turning off all the remaining segments)
- If A=1, then the decimal value corresponding to the received bits (B and C) should be displayed.

a) According to the above specification, fill-in the truth table for the 7-segment driver assuming **common-anode** 7-segment display:

Inputs			Outputs							Output value in hexadecimal (a is MSB)
A	B	C	a	b	c	d	e	f	g	
0	**0**	**0**								
0	**0**	**1**								
0	**1**	**0**								
0	**1**	**1**								
1	**0**	**0**								
1	**0**	**1**								
1	**1**	**0**								
1	**1**	**1**								

b) Write the Boolean equations of outputs a, b, c, d, e, f, and g:

a =

b =

c =

d =

e =

f =

g =

c) Implement the 7-segment driver using **behavioral** Verilog code in a module named "segdriver" inside the "segdriver.v" file. This module has three inputs (i.e. A, B and C; where A is the MSB) and seven outputs (i.e. a, b, c, d, e, f, and g).

Paste your "segdriver.v" code here:

Part4: TDM Communication Circuit

a) Implement the TDM circuit shown in Figure 1 using **structural** Verilog code in a module named "circuit1" inside the "circuit1.v" file. This module has six inputs (i.e. Tx01, Tx00, Tx11, Tx10, SS, and RS) and 14 outputs (i.e. "a, b, c, d, e, f, and g" for "receiver0" 7-segment display and "a1, b1, c1, d1, e1, f1, and g1" for "receiver1" 7-segment display).

Paste your "circuit1.v" code here:

b) Set “circuit1.v” as top-level entity and assign the following pins to the inputs and outputs in the "**circuit1.v**" file, download your design on the FPGA, and test it:

Input Switches		
Tx00	iSW[1]	PIN_AB26
Tx01	iSW[2]	PIN_AB25
Tx10	iSW[5]	PIN_AC24
Tx11	iSW[6]	PIN_AC23
SS	iSW[8]	PIN_AD24
RS	iSW[9]	PIN_AE27

Receiver0 7-seg Display		
a	oHEX3_D[0]	PIN_P6
b	oHEX3_D[1]	PIN_P4
c	oHEX3_D[2]	PIN_N10
d	oHEX3_D[3]	PIN_N7
e	oHEX3_D[4]	PIN_M8
f	oHEX3_D[5]	PIN_M7
g	oHEX3_D[6]	PIN_M6

Receiver1 7-seg Display		
a1	oHEX4_D[0]	PIN_P1
b1	oHEX4_D[1]	PIN_P2
c1	oHEX4_D[2]	PIN_P3
d1	oHEX4_D[3]	PIN_N2
e1	oHEX4_D[4]	PIN_N3
f1	oHEX4_D[5]	PIN_M1
g1	oHEX4_D[6]	PIN_M2

c) Perform functional simulation using the combinations given the table below. The inputs “TX11 and TX10” should be **combined together with “TX11” as the MSB and format is unsigned decimal**. The inputs “TX01 and TX00” should be **combined together with “TX01” as the MSB and format is unsigned decimal**. The outputs “a, b, c, d, e, f, and g” should be **combined together with** “**a**” **as the MSB and format is hexadecimal**. Similarly, the outputs “a1, b1, c1, d1, e1, f1, and g1” should be **combined together with** “**a1**” **as the MSB and format is hexadecimal**.

Inputs						Outputs	
SS	**RS**	**TX11**	**TX10**	**TX01**	**TX00**	**a-to-g in hexadecimal**	**a1-to-g1 in hexadecimal**
0	0	1	0	0	0		
0	0	1	1	0	1		
0	1	0	1	1	0		
0	1	0	0	1	1		
1	0	1	0	0	0		
1	0	1	1	0	1		
1	1	0	1	1	0		
1	1	0	0	1	1		

Paste a snapshot of your “circuit1.v” simulation report here:

Experiment 6
Arithmetic Circuits Design and Implementation

Introduction:

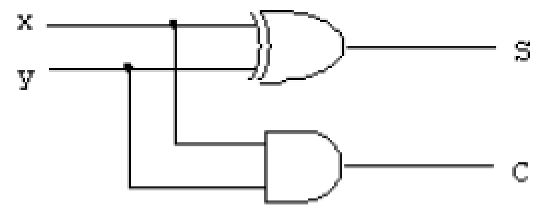
Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: the appropriate operand is simply negated before being added.

Objectives:

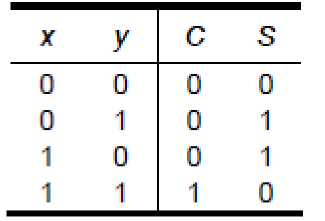
- To understand the concept of Half and Full Adders.
- Design and build Ripple Carry Adder .
- Introduce 4-bit magnitude comparator.
- Design and implement binary multiplier

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.



Logic Diagram of the Half-Adder



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

Table of the Half-Adder

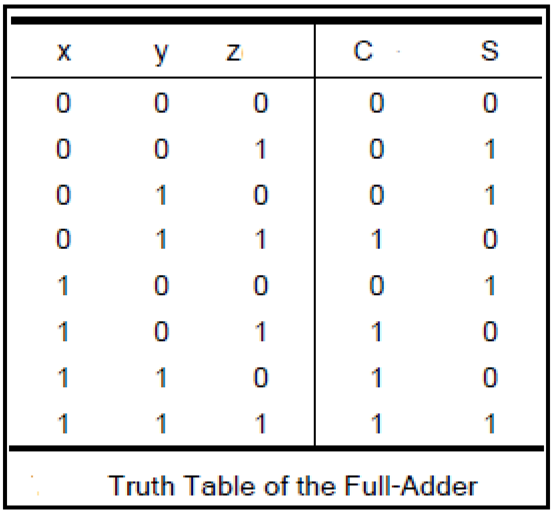
Figure(1)

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

$$C = xy$$

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

.

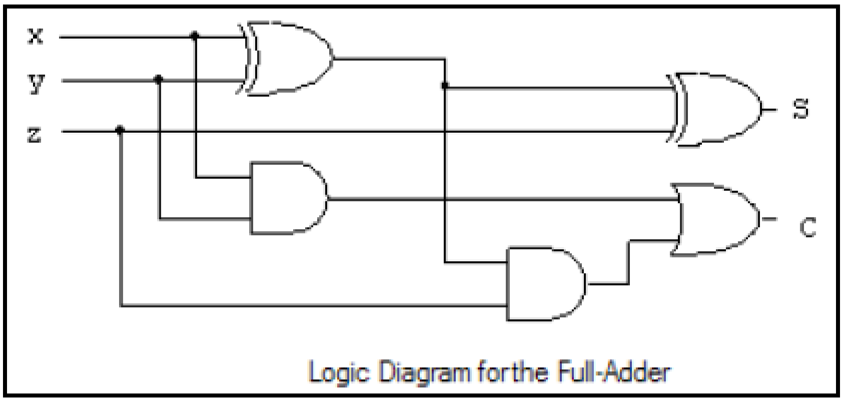
From the truth table, we can obtain the Boolean expression of C & S outputs as follows :

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

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



Logic Diagram forthe Full-Adder

Figure(2)

Ripple Adder

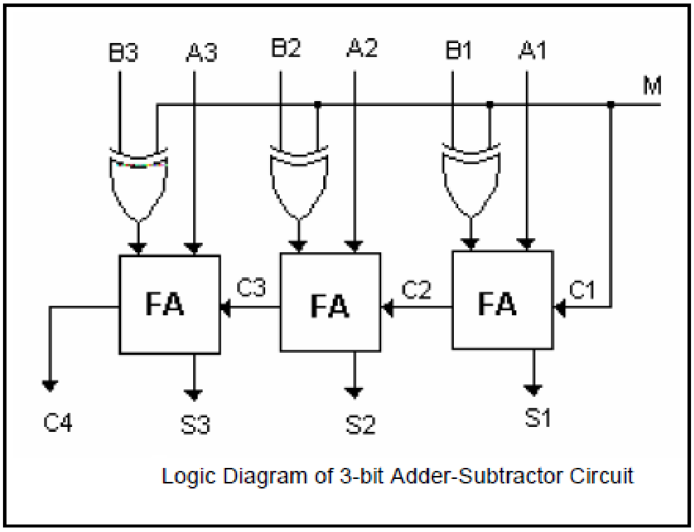
Two binary numbers, each of n bits, can be added using a ripple adder, a cascade of n full adders; each full adder handles one bit. Each Cout of a full adder is connected to the Cin of the higher full adder. The Cin of the least significant full adder is set to 0.

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 minued. 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 minued 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.

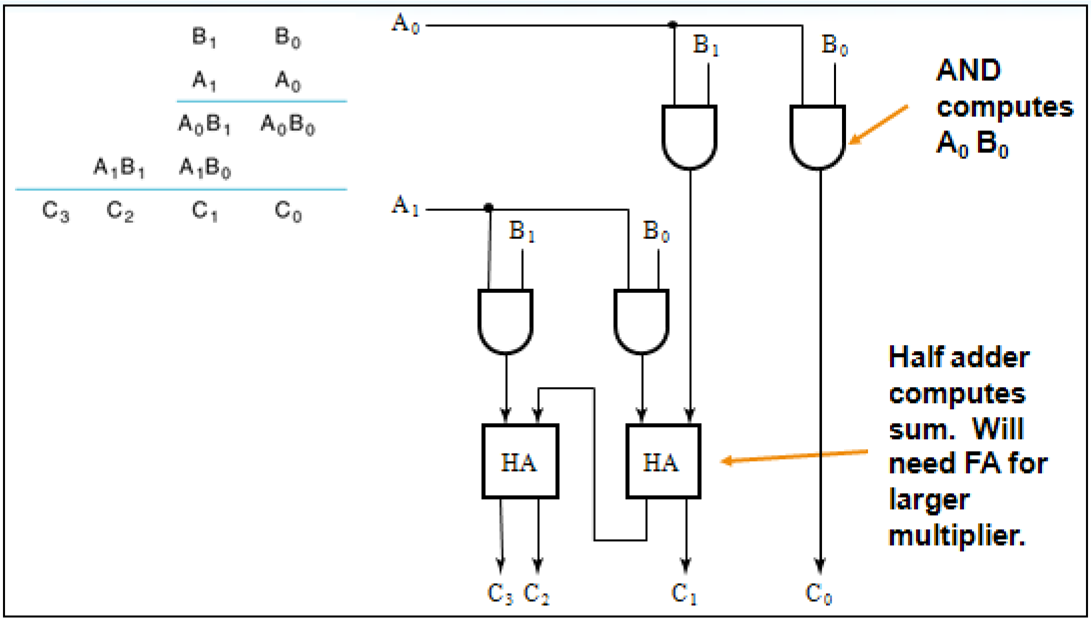


Figure(3)

Multiplier:

If we want to multiply tow numbers(A,B) ,each of them is consist of two bit as follows:
B = {B1 B0},
A = {A1 A0}
Then we multiply by doing single-bit multiplications and shifts.

		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0



AND computes A_0 B_0

Half adder computes sum. Will need FA for larger multiplier.

Figure(4)

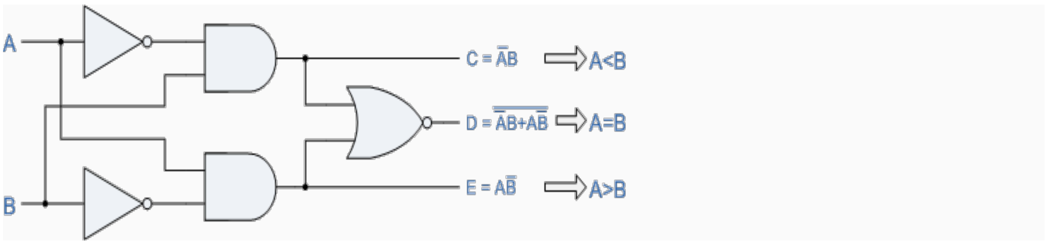
Comparator:

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.

$$A > B, \quad A = B, \quad A < B$$

1-bit Comparator


A
B
$C = \overline{A}B$ ⇒ A<B
$D = \overline{\overline{A}B + A\overline{B}}$ ⇒ A=B
$E = A\overline{B}$ ⇒ A>B

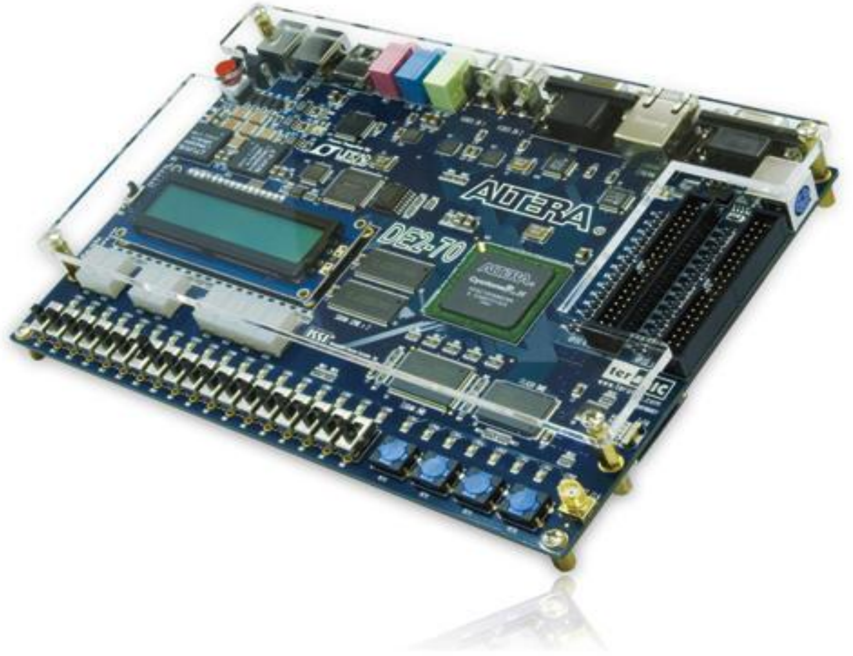
Then the operation of a 1-bit digital comparator is given in the following Truth Table.

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

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

Labsheet6: Arithmetic and Logic Unit (ALU) Design and Simulation



Name:

Student ID:

Section:

Problem Description:

The goal of this experiment is to design a simple 2-bit ALU combinational circuit that performs three **unsigned** operations: addition, subtraction and multiplication. As shown in Figure 1, the ALU circuit has two 2-bit unsigned numbers A {A1A0} and B {B1B0} as inputs and a 4-bit unsigned number R {R3R2R1R0} as an output. The ALU circuit also has 2-bit control signal m {m1m0} that is used to choose the desired ALU operation as shown in Table 1.

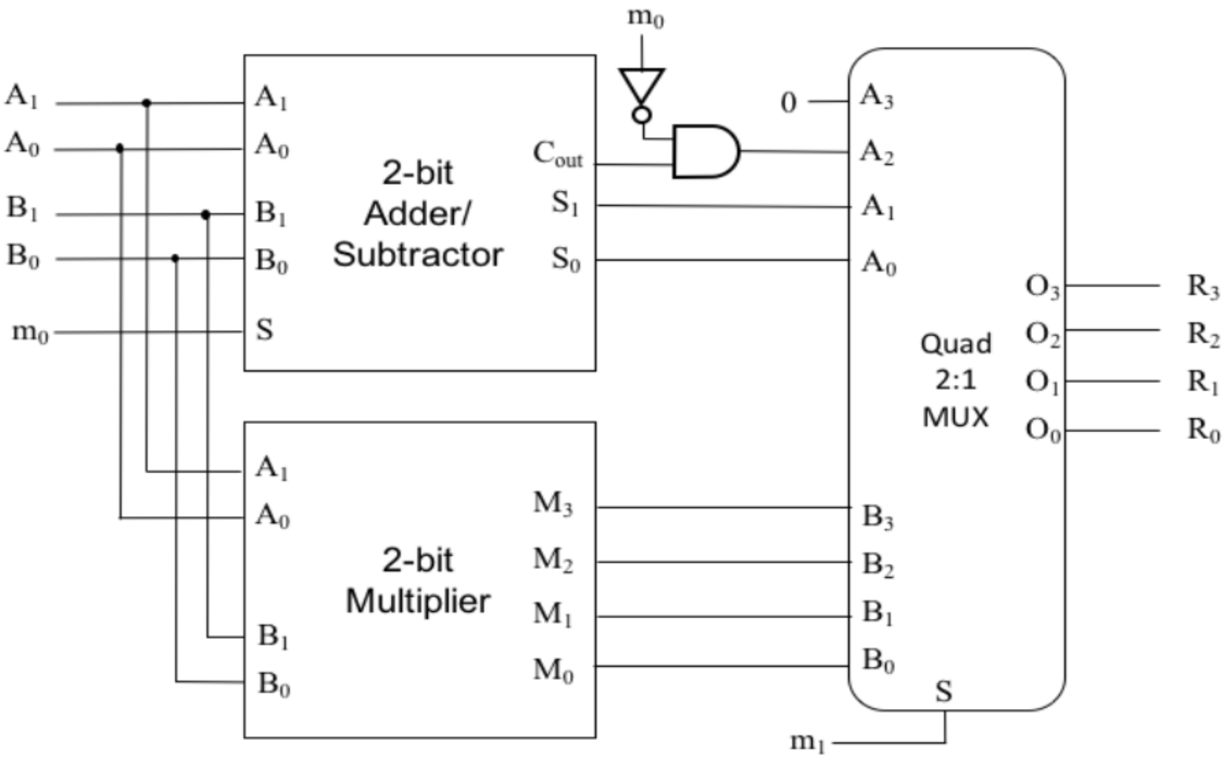


Figure 1: ALU Diagram

Table 1: ALU Truth Table

m_1	m_0	Arithmetic Operation
0	0	Addition
0	1	Subtraction
1	X	Multiplication

The experiment is divided into four parts: 2-to-1 **quad** multiplexer implementation, 2-bit ripple-carry adder/subtractor implementation, 2-bit multiplier implementation, and top-level entity.

Part1: 2-to-1 Quad MUX

a) Inside the **Mux2to1.v** file, write a **behavioral** Verilog code to implement a 2-to-1 multiplexer module. This module has three inputs (I0, I1 and S) and one output (OUT). (Pre-lab)

b) Inside the **QuadMux2to1.v** file, write a **structural** Verilog code to implement the **quad** 2-to-1 multiplexer module using four instances of the 2-to-1 multiplexer design in part a. This module has nine inputs (A3, A2, A1, A0, B3, B2, B1, B0 and S) and four outputs (O3, O2, O1 and O0). Notice that when S = 0, input A {A3A2A1A0} is selected. (Pre-lab)

Paste your “Mux2to1.v” code here:

Paste your “QuadMux2to1.v” code here:

Part2: 2-bit Ripple-Carry Adder/Subtractor

a) Fill-in the truth table for the 1-bit Full Adder (FA) shown in Figure 2. (Pre-Lab)

Inputs			Outputs	
Cin	**in1**	**in0**	**Cout**	**sum**

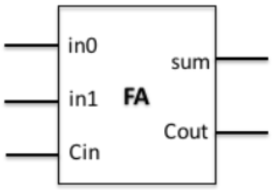


Figure 2: 1-to-2 DMUX

b) Write the Boolean equations of outputs sum and Cout: (Pre-Lab)

sum =
Cout =

c) Inside the **FA.v** file, write a **behavioral** Verilog code to implement the Full Adder module. This module has three inputs (in0, in1 and Cin) and two outputs (sum and Cout).

Paste your "FA.v" code here:

d) Inside the **TwoBitAdderSubtractor.v** file, write a **structural** Verilog code to implement the 2-bit adder/subtractor module shown in Figure 3 using two cascaded full adders and two XOR gates from **lib.v**.. This module has five inputs (A0, A1, B0, B1, and S) and three outputs (S0, S1, and Cout).

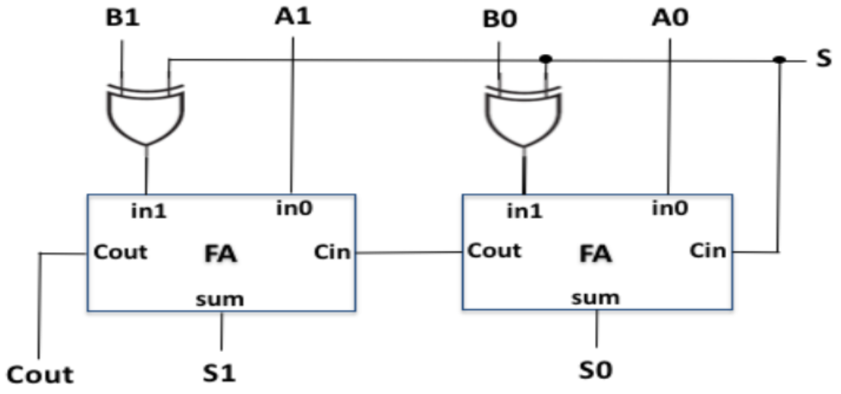


Figure 3: 2-bit Adder/Subtractor

Paste your "TwoBitAdderSubtractor.v" code here:

Part3: 2-bit Multiplier

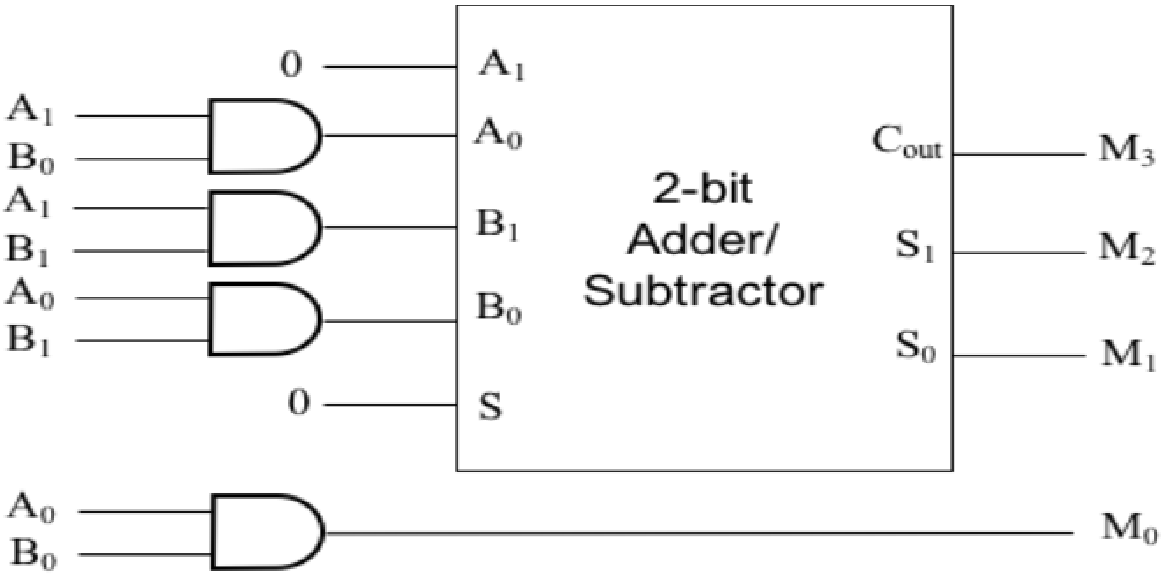


Figure 4: 2-bit Multiplier

a) Inside the **Mul.v** file, write a **structural** Verilog code to implement the 2-bit multiplier module shown in Figure 4 using one instance of the 2-bit adder/subtractor circuit and four instances of the 2-input AND gate module from **lib.v**. This module has four inputs (A0, A1, B0, and B1) and four outputs (M0, M1, M2 and M3).

Paste your “Mul.v” code here:

Part4: ALU Circuit

a) Inside the **ALU.v** file, write a **structural** Verilog code to implement the full ALU circuit module given in Figure 1.

Paste your "ALU.v" code here:

b) Set "ALU.v" as top-level entity and perform functional simulation using the waveform file given to you "ALU.vwf" and make sure that your design is functionally correct.

Paste a snapshot of your "ALU.v" simulation report here:

Part5: ALU Implementation on FPGA

In order to implement and test the ALU design on the FPGA, four 7-segment displays are needed to show the values of operand A, operand B, operation (i.e. addition, subtraction, or multiplication), and result. You are provided with two Verilog files that are already completed for you: “segdriver2.v” for operation and “segdriver4.v” for operands and result.

a) Inside the **ALU_FPGA.v** file, write a **structural** Verilog code that interfaces the ALU circuit with the four 7-segment drivers.

b) Set “ALU_FPGA.v” as top-level entity and assign the following pins to the inputs and outputs in the "**ALU_FPGA.v**" file, download your design on the FPGA, and test it.

Input Switches		
B0	iSW[1]	PIN_AB26
B1	iSW[2]	PIN_AB25
m0	iSW[3]	PIN_AC27
m1	iSW[4]	PIN_AC26
A0	iSW[5]	PIN_AC24
A1	iSW[6]	PIN_AC23
Result 7-seg Display		
a0	oHEX0_D[0]	PIN_AE8
b0	oHEX0_D[1]	PIN_AF9
c0	oHEX0_D[2]	PIN_AH9
d0	oHEX0_D[3]	PIN_AD10
e0	oHEX0_D[4]	PIN_AF10
f0	oHEX0_D[5]	PIN_AD11
g0	oHEX0_D[6]	PIN_AD12
Operand-B 7-seg Display		
a1	oHEX1_D[0]	PIN_AG13
b1	oHEX1_D[1]	PIN_AE16
c1	oHEX1_D[2]	PIN_AF16
d1	oHEX1_D[3]	PIN_AG16
e1	oHEX1_D[4]	PIN_AE17
f1	oHEX1_D[5]	PIN_AF17
g1	oHEX1_D[6]	PIN_AD17
Operation 7-seg Display		
a2	oHEX2_D[0]	PIN_AE7
b2	oHEX2_D[1]	PIN_AF7
c2	oHEX2_D[2]	PIN_AH5
d2	oHEX2_D[3]	PIN_AG4
e2	oHEX2_D[4]	PIN_AB18
f2	oHEX2_D[5]	PIN_AB19
g2	oHEX2_D[6]	PIN_AE19
Operand-A 7-seg Display		
a3	oHEX3_D[0]	PIN_P6
b3	oHEX3_D[1]	PIN_P4
c3	oHEX3_D[2]	PIN_N10
d3	oHEX3_D[3]	PIN_N7
e3	oHEX3_D[4]	PIN_M8
f3	oHEX3_D[5]	PIN_M7
g3	oHEX3_D[6]	PIN_M6

Experiment 7
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.

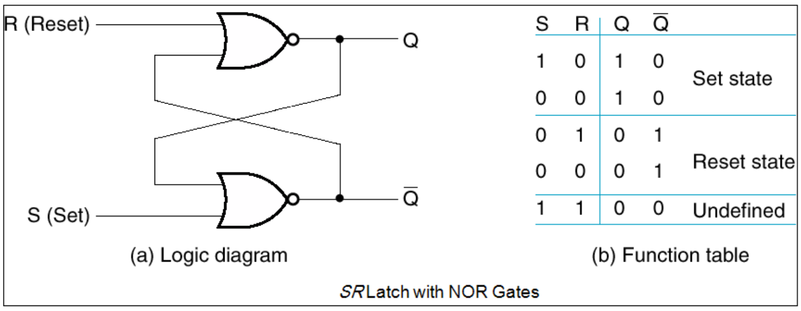
Objectives:

- 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

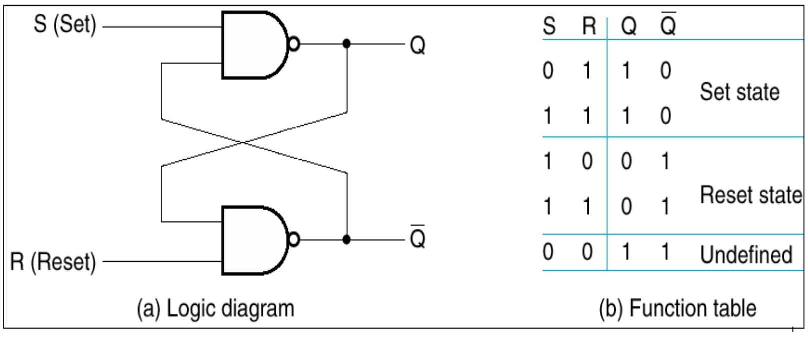
The most basic sequential unit is the S-R latch. From this basic circuit flip-flops are constructed, and from flip-flops, the registers, memories, and state machines can be made. The basic S-R latch has two inputs, S and R, and two outputs, Q and Q`.



S	R	Q	$\bar{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

Figure(1)

Similar SR latch can be made from NANDs as follow:



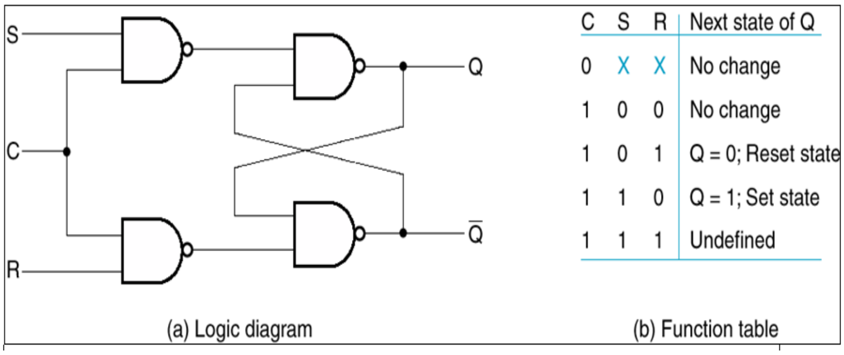
S	R	Q	$\bar{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

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).



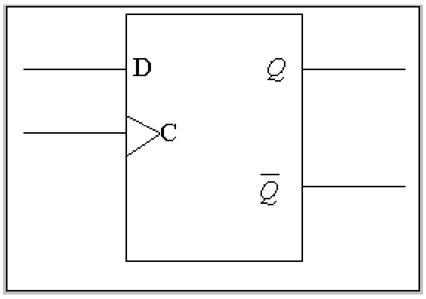
(a) Logic diagram

C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set state
1	1	1	Undefined

(b) Function table

Figure(3)

3- D Flip-Flop

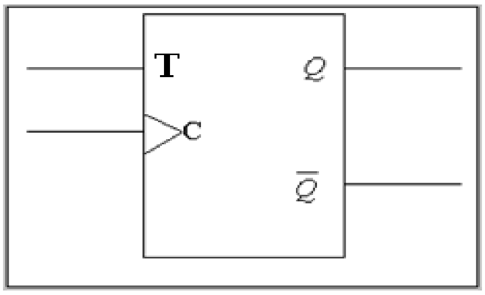


Figure(4)

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

- ➢ 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

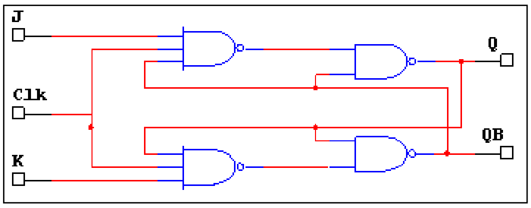


T	C	Q	Operation
0	Rising edge	0	Q **(No change)**
1	Rising edge	1	$\overline{Q}$ **(complement)**

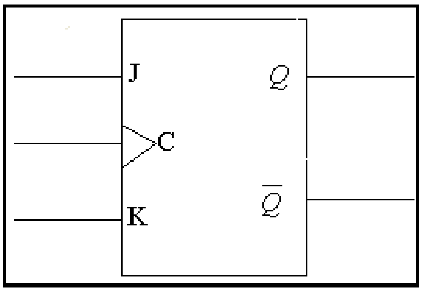
Figure(5)

5- J-K Flip-Flop

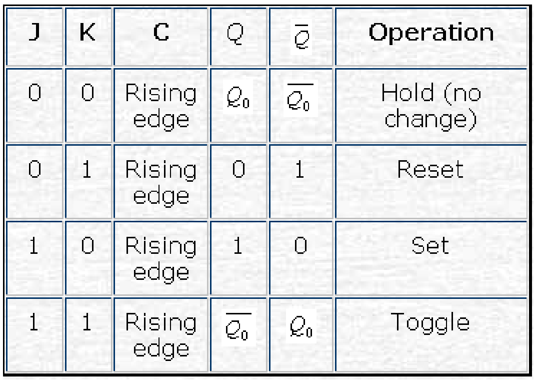
The J-K flip-flop is simply an S-R flip-flops that has been modified so that both inputs can be active at the same time. Where in the S-R flip-flop this condition was considered invalid, in the J-K flip-flop this condition toggles the output on successive clock cycles.



Figure(6)



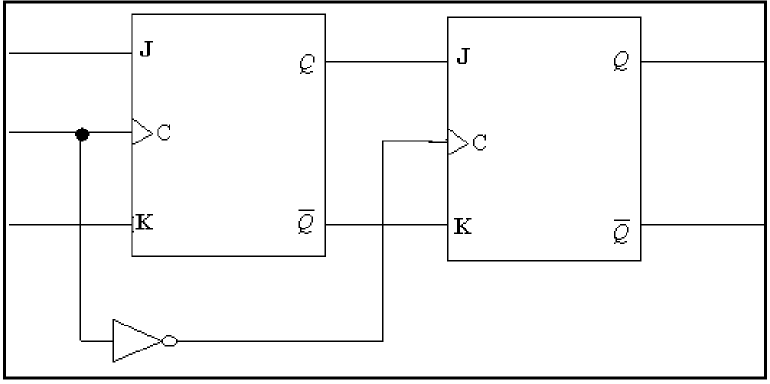
Figure(7)



J	K	C	Q	$\overline{Q}$	Operation
0	0	Rising edge	Q_0	$\overline{Q_0}$	Hold (no change)
0	1	Rising edge	0	1	Reset
1	0	Rising edge	1	0	Set
1	1	Rising edge	$\overline{Q_0}$	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-salve arrangement.

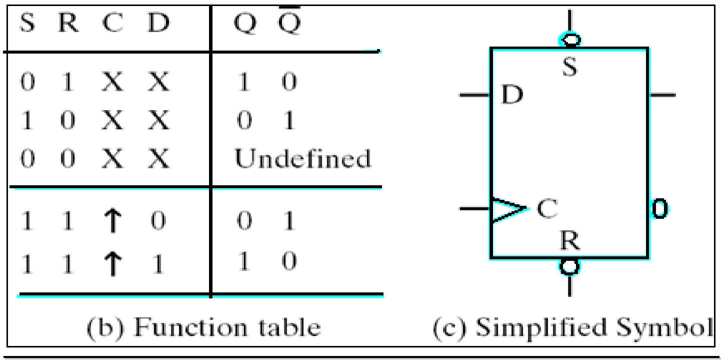


Figure(8)

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

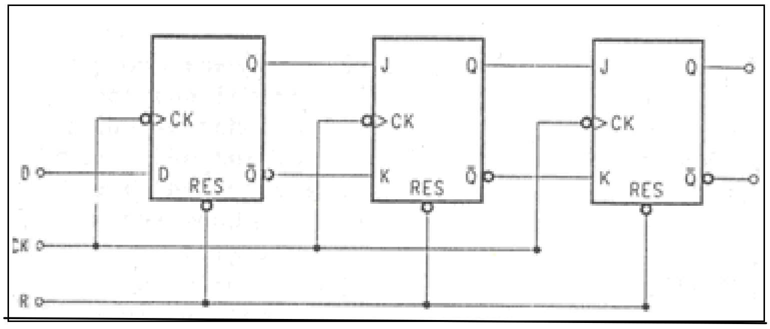
(b) Function table

(c) Simplified Symbol

Figure(9)

8- 3-Stage Shift Register

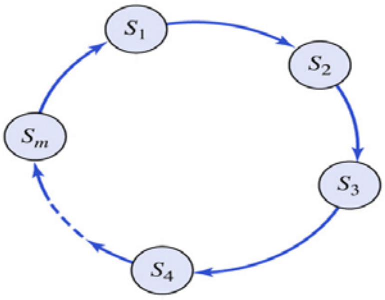
A group of cascaded FFs used to store related bits of information is known as a register. A register that is used to store information arriving from a source is called a shift register. Each FF output of a shift register is connected to the input of the next FF, and a common clock pulse is applied to 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 Si 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 Q2 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 Q2 has been divided by eight or 2n were 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_0, Q1, Q2 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.

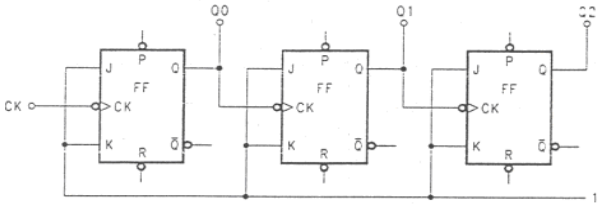


Figure (11)

Advantages of Ripple Counters:

- Simple hardware and design.

Disadvantages of Ripple Counters:

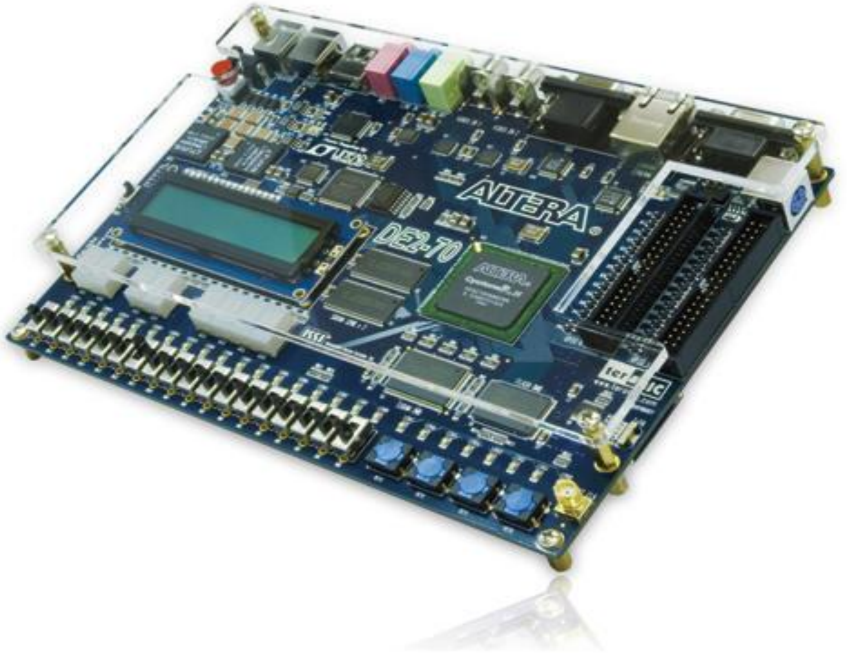
- They are asynchronous circuits, and can be unreliable and delay dependent, if more logic is added.
- Large ripple counters are slow circuits due to the length of time required for the ripple to occur.

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

Labsheet 7: Latches and Flip-Flops



Name:

Student ID:

Section:

Part 1: Clocked SR-latch Verilog Implementation and Simulation

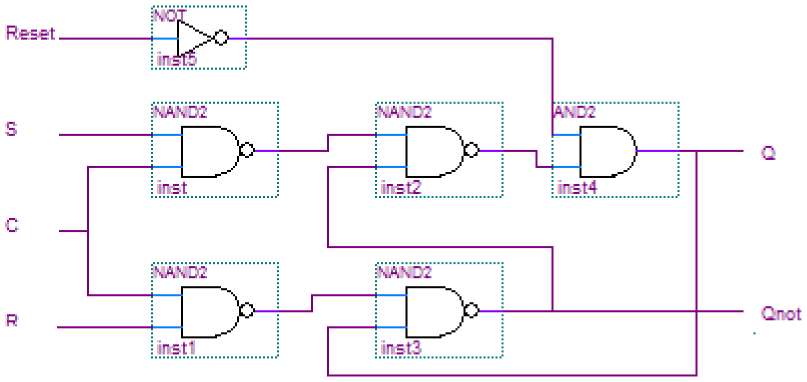


Figure1 : Clocked SR-latch Implementation

SR-Latch Function table

Reset	C	S	R	Operation
1	x	x	x	Reset
0	0	x	x	Hold
0	1	0	0	Hold
0	1	0	1	Reset
0	1	1	0	Set
0	1	1	1	Forbidden

1. Inside the **SR2.v** file, write a **structural** Verilog module to implement the clocked SR-latch given in Figure1 . The modules for the basic gates are given in the **lib.v** file.

2. Use the **SR2.vwf** vector waveform file to perform **functional** simulation for your module in SR2.v (i.e. SR2.v should be set as top-level entity). Validate that the outputs' values are correct.

3. Paste your SR2.v code and a snap shot of the simulation report in the spaces given below.

Paste your "SR2.v" code here:

Paste a snapshot of your "SR2.v" simulation report here:

Part 2: D-FF Implementation and Simulation

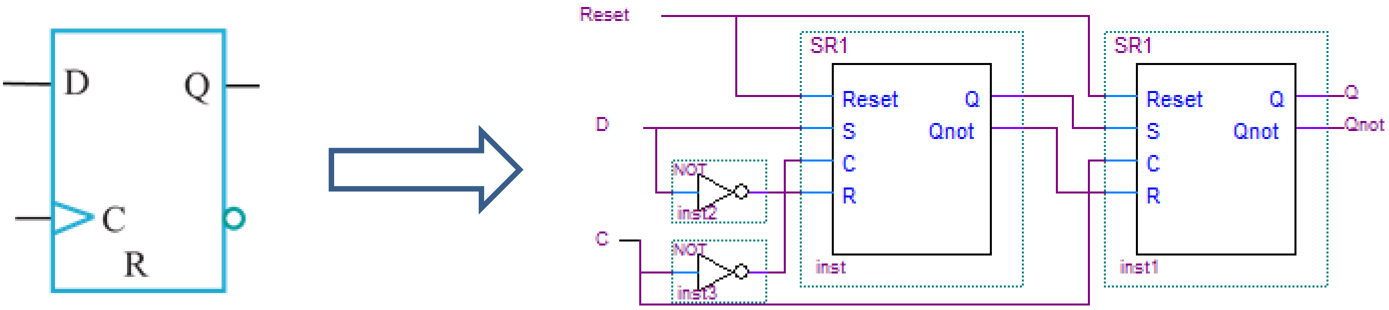


*Figure 2: **D-FF Implementation***

1. Fill the function table below with the correct operation for the positive-edge triggered D-FF**.**

R	C	D	Operation
1	x	x	
0	↑	0	
0	↑	1	

2. Inside the **dff1.v** file, write a **structural** Verilog module to implement the D-FF circuit given in Figure 2. You need to use the clocked SR-latch module defined in **SR2.v** and the basic gate modules defined in **lib.v**.

3. Use the **dff1.vwf** vector waveform file to perform **functional** simulation for your module in dff1.v (i.e. dff1.v should be set as top-level entity). Validate that the outputs' values are correct.

4. Paste your dff1.v code and a snap shot of the simulation report in the spaces given below.

Paste your "dff1.v" code here:

Paste a snapshot of your “dff1.v” simulation report here:

Part 3: JK-FF Implementation and Simulation

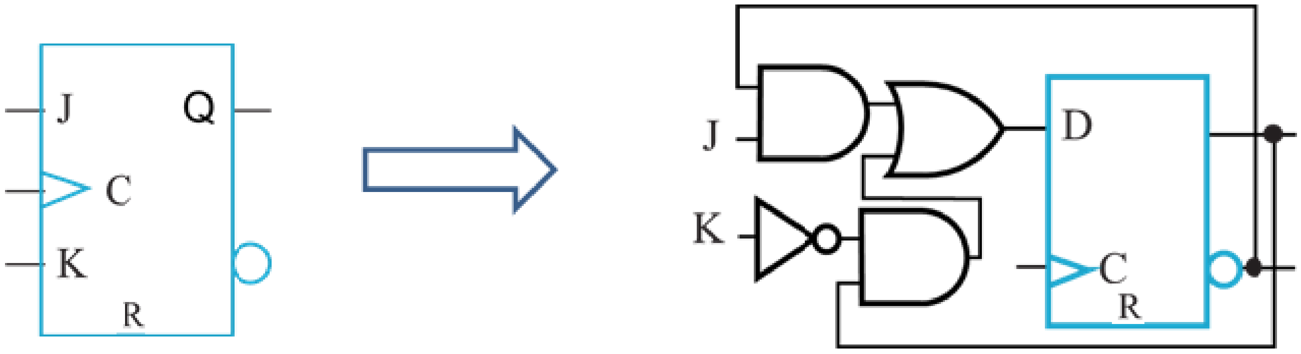


Figure 3: JK-FF Implementation

1. Fill the function table below with the correct operation for the positive-edge triggered JK-FF.

R	C	J	K	Operation
1	x	x	x	
0	↑	0	0	
0	↑	0	1	
0	↑	1	0	
0	↑	1	1	

2. Inside the **jkff1.v** file, write a **structural** Verilog module to implement the JK-FF circuit given in Figure 3. You need to use the D-FF module defined in **dff1.v** and the basic gate modules defined in **lib.v**.

3. Use the **jk.vwf** vector waveform file to perform **functional** simulation for your module in jkff1.v (i.e. jkff1.v should be set as top-level entity). Validate that the outputs’ values are correct.

4. Paste your jkff1.v code and a snap shot of the simulation report in the spaces given below.

Paste your “jkff1.v” code here:

Paste a snapshot of your “jkff1.v” simulation report here:

Part 4: T-FF Implementation and Simulation

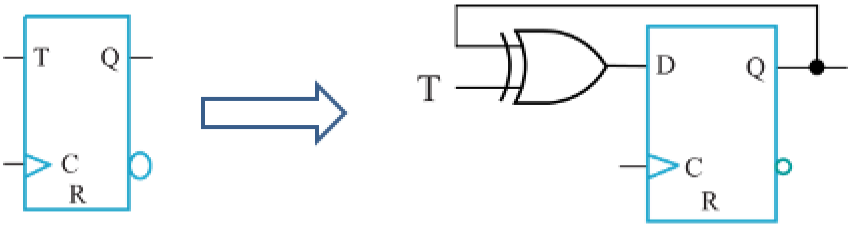


Figure 4: T-FF Implementation

1. Fill the function table below with the correct operation for the positive-edge triggered T-FF.

R	C	T	Operation
1	x	x	
0	↑	0	
0	↑	1	

2. Inside the **tff1.v** file, write a **structural** Verilog module to implement the T-FF circuit given in Figure 4. You need to use the D-FF module defined in **dff1.v** and the basic gate modules defined in **lib.v**.

3. Use the **tff1.vwf** vector waveform file to perform **functional** simulation for your module in tff1.v (i.e. tff1.v should be set as top-level entity). Validate that the outputs' values are correct.

4. Paste your tff1.v code and a snap shot of the simulation report in the spaces given below.

Paste your "tff1.v" code here:

Paste a snapshot of your "tff1.v" simulation report here:

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.

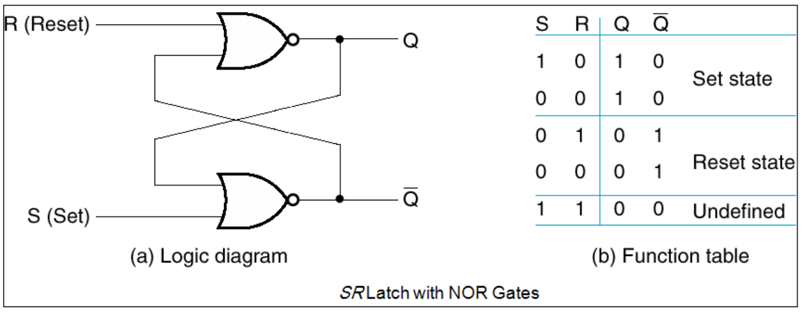
Objectives:

- 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`.



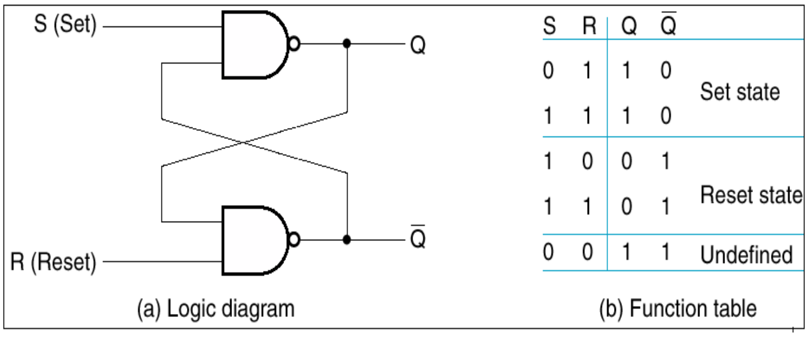
S	R	Q	$\bar{Q}$	
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

(a) Logic diagram (b) Function table

SR Latch with NOR Gates

Figure(1)

Similar SR latch can be made from NANDs as follow:



S	R	Q	$\bar{Q}$	
0	1	1	0	Set state
1	1	1	0	
1	0	0	1	Reset state
1	1	0	1	
0	0	1	1	Undefined

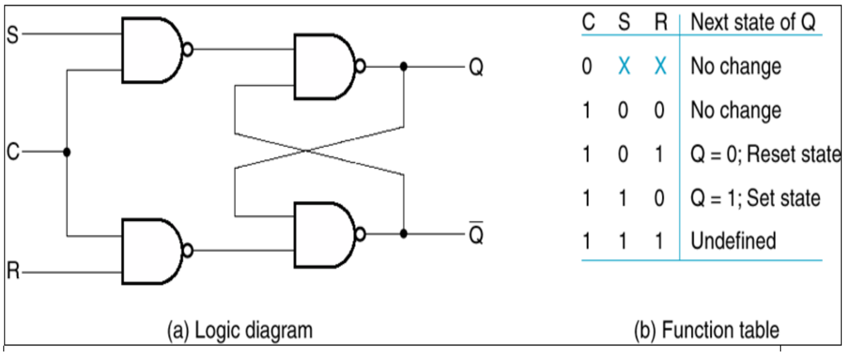
(a) Logic diagram (b) Function table

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).



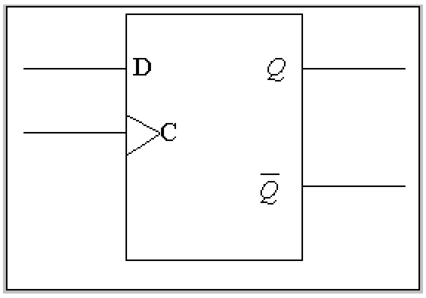
C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set state
1	1	1	Undefined

(a) Logic diagram

(b) Function table

Figure(3)

3- D Flip-Flop

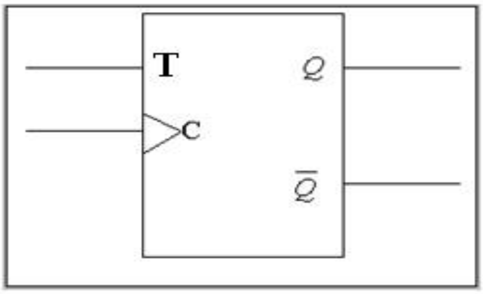


D	C	Q	$\overline{Q}$	Operation
0	Rising edge	0	1	Reset (stores 0)
1	Rising edge	1	0	Set (stores (1)

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

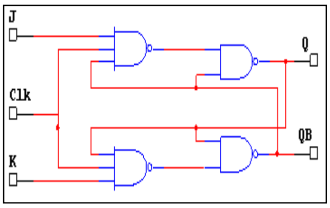


T	C	Q	Operation
0	Rising edge	0	Q (No change)
1	Rising edge	1	$\overline{Q}$ (complement)

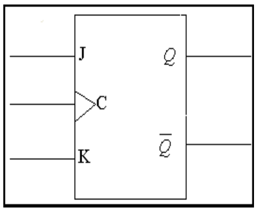
Figure(5)

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)

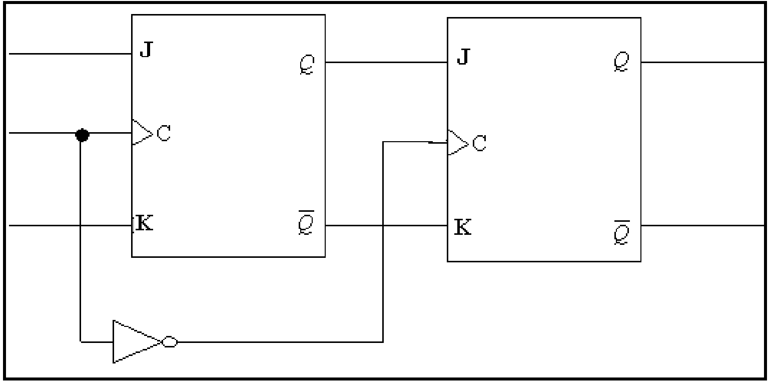


Figure(7)

J	K	C	Q	$\overline{Q}$	Operation
0	0	Rising edge	Q_0	$\overline{Q_0}$	Hold (no change)
0	1	Rising edge	0	1	Reset
1	0	Rising edge	1	0	Set
1	1	Rising edge	$\overline{Q_0}$	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-salve arrangement.

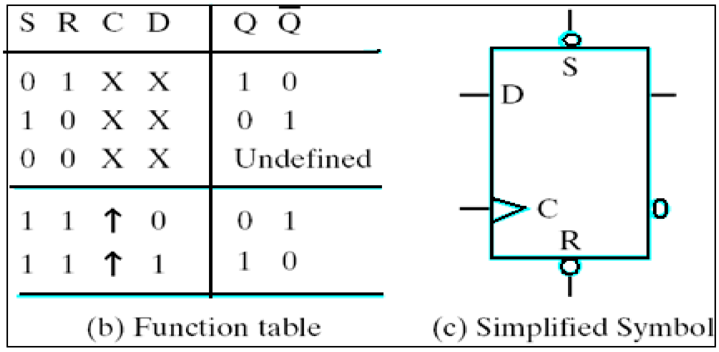


Figure(8)

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



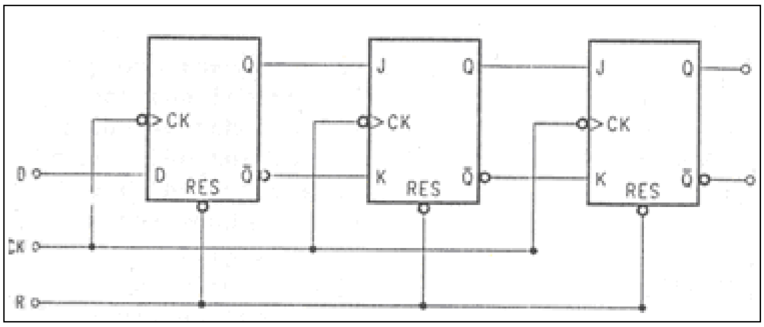
S	R	C	D	Q	Q̄
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	Undefined	
1	1	↑	0	0	1
1	1	↑	1	1	0

(b) Function table (c) Simplified Symbol

Figure(9)

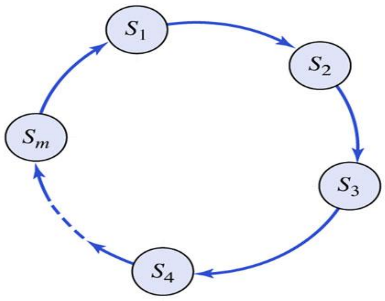
8- 3-Stage Shift Register

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.



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 Si 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 Q2 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 Q2 has been divided by eight or 2n were 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$_0$, Q1, Q2 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.

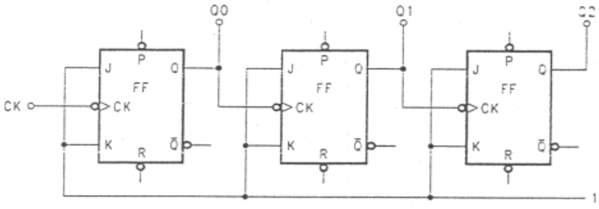


Figure (11)

- 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.

3) BCD Counter (74LS160)

Since some digital functions are performed in BCD, the decade counter is often used.

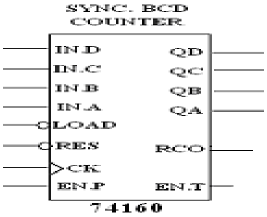


Figure 5

- CK: The counter clock.
- EN.P, EN.T: Are the two active high enable signals.
- QAQBQCQD: 4-bit counter output.
- RES: Active low reset input; when RES = 0 then the output QAQBQCQD = 0000.
- LOAD: active low load input; if:
  - LOAD = 0 then the counter start counting from the value on the inputs (IN.A, IN.B, IN.C, IN.D) to 9 each clock cycle.
  - LOAD = 1 then the counter start counting from the value on QAQBQCQD to 9 each clock cycle.
- IN.A, IN.B, IN.C, IN.D: The starting count value when LOAD = 0.

Ex1:

IN.A IN.B IN.C IN.D = 0101

CLOCK	Counter output	LOAD
1	0	1
2	1	1
3	2	1
4	3	1
5	4	1
6	5	1
7	6	1
8	7	1
9	8	1
10	9	1
11	0	1

EX2:

IN.A IN.B IN.C IN.D = 0101

CLOCK	Counter output	LOAD
1	5	0
2	5	0
3	6	1
4	7	1
5	8	1
6	9	1
7	0	1
8	5	0
9	5	0
10	5	0
11	6	1

EX3:

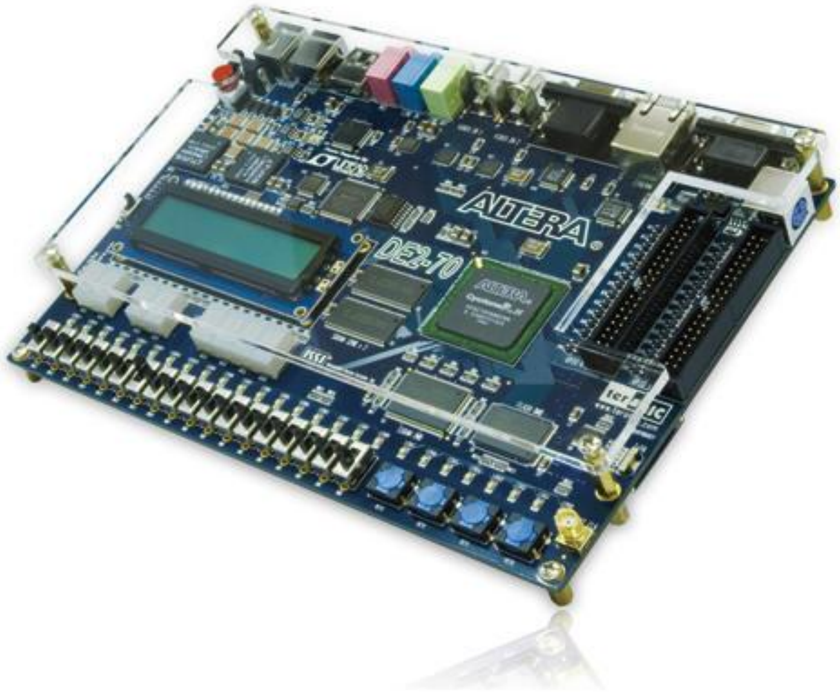
IN.A IN.B IN.C IN.D = 0101

CLOCK	Counter output	LOAD
1	5	0
2	6	1
3	7	1
4	8	1
5	9	1
6	5	0
7	6	1
8	7	1
9	8	1
10	9	1
11	5	0

- Note that if you want to count like 5,6,7,8,9,5,6,…as in EX3 above then the LOAD input should be 0 when counter output reaches 9 to take the loaded value (5) not (0).
- RCO: The output that become “1” when the value on QAQBQCQD = 1001 and “0” otherwise.
  This input is used to ensure that the load input become “0” when the counter output reaches “9” by connecting RCO to inverter (because LOAD IS ACTIVE LOW) and the output of the inverter to the LOAD input.

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

Labsheet8: Shift Register & Counter Design



Name:

Student ID:

Section:

Problem Description:

In this experiment, you are required to implement the circuit shown in Figure1 . The circuit consists of two components: a 3-bit synchronous counter and a 3-bit shift register. Note that the *"**clk**"* input of the counter is obtained from the external *"**clk**"* input. On the other hand, the *"**clk**"* input of the 3-bit register is obtained from the "***zero count***" output of the 3-bit counter. The "***zero count***" output is 0 when counter outputs C2, C1, and C0 equal 001, 010, 011, 100, 101, 110, or 111. On the other hand, the "***zero count***" output is 1 when counter outputs C2, C1, and C0 equal 000.

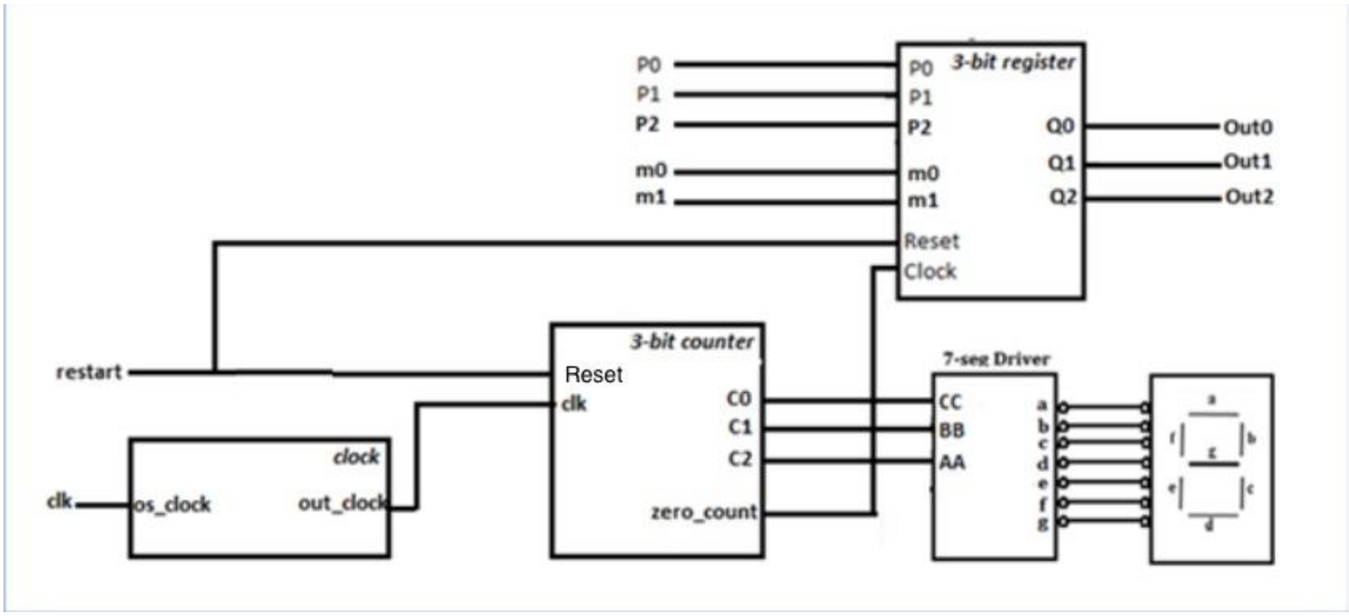


Figure1 : A 3-bit Shift Register Controlled by a 3-bit Down Counter

Part 1: 3-bit Synchronous Counter

In this part, you are required to design a 3-bit count-down counter that counts 7, 6, 5, 4, 3, 2, 1, 0, 7, 6, 5, 4, ... and so on using D flip-flops.

1. Fill the following state table according to the required counter design. (PreLab)

Present State			Next State					
C2	C1	C0	C2	C1	C0	D2	D1	D0
0	0	0						
0	0	1						
0	1	0						
0	1	1						
1	0	0						
1	0	1						
1	1	0						
1	1	1						

2. Use the k-maps below to derive the optimized input equations of the three flip flops. (PreLab)

D0=

D1=

D2 =

3. Draw the sequential circuit that implements the 3-bit counter. Add to your circuit the gate(s) required to generate the output signal "***zero_count***".

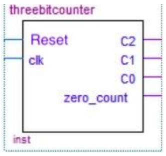
4. In the file "***threebitcounter.v***", write a Verilog module that implements the 3-bit counter with the "***zero_count***" signal **structurally** using the modules defined in "***lib.v***" and "***dff1.v***".

5. Set "***threebitcounter.v***" as your top-level entity, compile, and perform functional simulation using the file "***Counter.vwf***". Make sure that your counter design works as expected, then paste your code and simulation report below.

Paste your "threebitcounter.v" code here:

Paste a snapshot of your "threebitcounter.v" simulation report here:

6. Convert the 3-bit counter you built to a block as in the figure below. This block will be used in the final circuit implementation in part 3. Do the following:
   File → Create/Update → Create Symbol Files for Current Files



Part 2: 3-bit Shift-Register

1. In the file "***reg3b.v***", write a Verilog module that implements the 3-bit shift register shown in Figure 2 **structurally** using the modules defined in "***mux1.v***" and "***dff1.v***". Notice that the shift register has four modes of operation according to the following table. The m1 and m0 bits represent the "***Mode***" bits used as select lines for the multiplexers.

m1	**m0**	**Operation**
0	0	Hold
0	1	Parallel Load
1	0	Rotate Right
1	1	Rotate Left

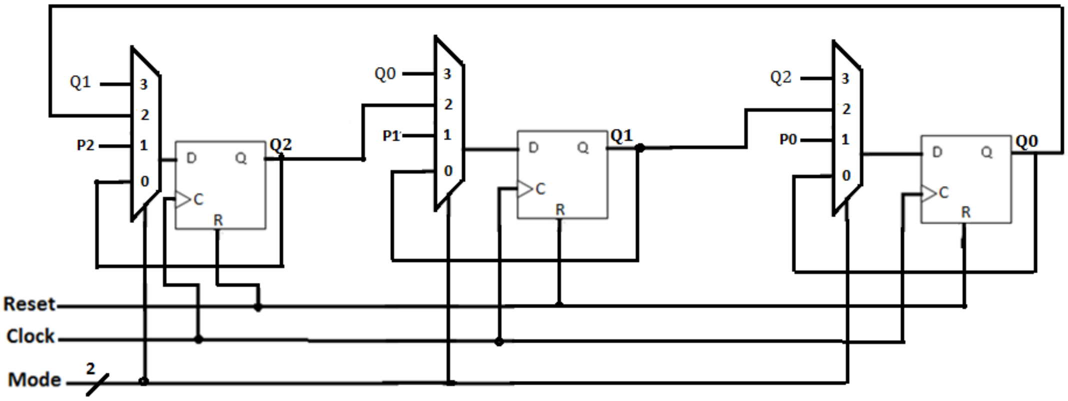


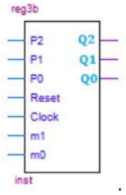
Figure 2: 3-bit Shift Register

2. Set "***reg3b.v***" as your top-level entity, compile, and perform functional simulation using the file "***Reg.vwf***". Make sure that your shift register design works as expected, then paste your code and simulation report below.

Paste your "reg3b.v" code here:

Paste a snapshot of your “reg3b.v” simulation report here:

3. Convert the 3-bit shift register you built to a block as in the figure below. This block will be used in the final circuit implementation in part 3. Do the following:
   File → Create/Update → Create Symbol Files for Current Files



Part 3: Final Circuit

1. Create a new **schematic file** and save it as “***circuit1.bdf***”. In the file, build the schematic diagram shown in Figure 1. This can be done by adding the segdecoder symbol, clock symbol, 3-bit counter (threebitcounter) symbol, and 3- bit register (reg3b) symbol to your bdf file. In order to add these symbols to your design, select the symbol of AND gate on the tool bar then expand the project menu. If you cannot find the symbols under the project menu, you can add them by typing the symbol name in the text box below.

2. Connect the symbols and add input (i.e. clk, restart, P0, P1, P2, m0, m1) and output (i.e. Out0, Out1, Out2, a, b, c, d, e, f, g) ports.

3. Important Notes:
   a. The block “clock.bsf” has one input “os_clock” and one output “out_clock”. This module implements a frequency division circuit which will be used to divide the high frequency clock of the FPGA oscillator (28 MHz) to obtain a slower clock (10 Hz) so that changes in the counter value can be detected on the 7-segment display and provide sufficient time for register setup. Hence, in your schematic diagram you are required to connect the “os_clock” to an input pin called “clk” (which will be assigned to the oscillator clock when you do pins assignment) and connect “out_clock” to the counter clock input.
   b. In your schematic diagram connect the clock input of the register to the output zero_count of the counter. This means that the state of the register will be updated each time the counter reaches count 0 according to the mode setting.

4. Assign the following pins to the inputs and outputs in the “***circuit1.bdf***” file, download your design on the FPGA, and test it.

Input Switches		
clk		PIN_E16
restart	iSW[1]	PIN_AB26
m0	iSW[2]	PIN_AB25
m1	iSW[3]	PIN_AC27
P0	iSW[4]	PIN_AC26
P1	iSW[5]	PIN_AC24
P2	iSW[6]	PIN_AC23
Outputs of the segdecoder		
a	oHEX0_D[0]	PIN_AE8
b	oHEX0_D[1]	PIN_AF9
c	oHEX0_D[2]	PIN_AH9
d	oHEX0_D[3]	PIN_AD10
e	oHEX0_D[4]	PIN_AF10
f	oHEX0_D[5]	PIN_AD11
g	oHEX0_D[6]	PIN_AD12
Outputs of the Shift Register		
Out0	oLEDR[1]	PIN_AK5
Out1	oLEDR[2]	PIN_AJ5
Out2	oLEDR[3]	PIN_AJ4