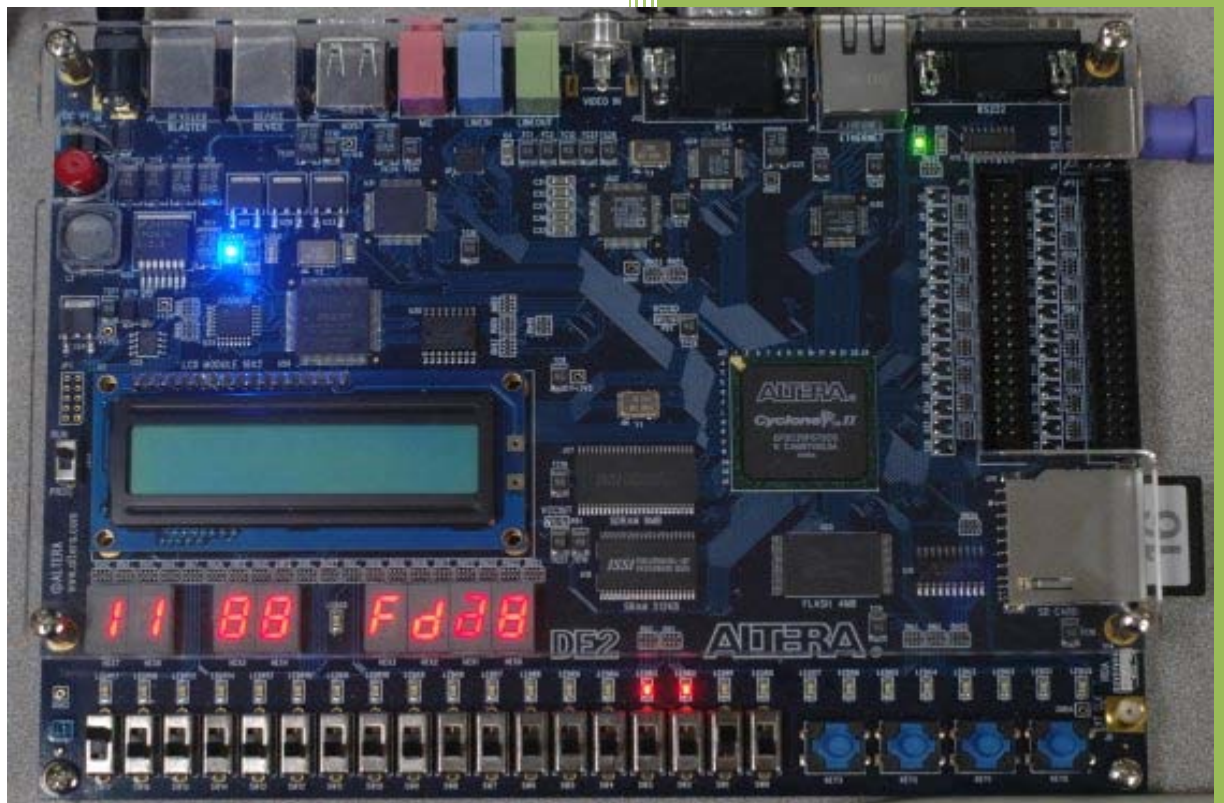


University of Sulaimani  
College of Engineering  
Electrical Engineering Department

2010-2011

Course Book of

# Advanced Electronics Lab.



Mr. Araz Sabir Ameen  
M.Sc. in  
Electronics & Communications

### ALTERA DE2 Development and Education Board

#### **DE2 Package:**

The DE2 package contains all components needed to use the DE2 board in conjunction with a computer that runs the Microsoft Windows software and Quartus II software installed on it.

#### **Package Contents**

Figure 1 shows a photograph of the DE2 package.



**Figure 1: The DE2 package contents.**

The DE2 package includes:

- DE2 board.
- USB Cable for FPGA programming.
- 9V DC wall-mount power supply

#### **Layout and Components:**

A photograph of the DE2 board is shown in Figure 2. It depicts the layout of the board and indicates the location of the connectors and key components.

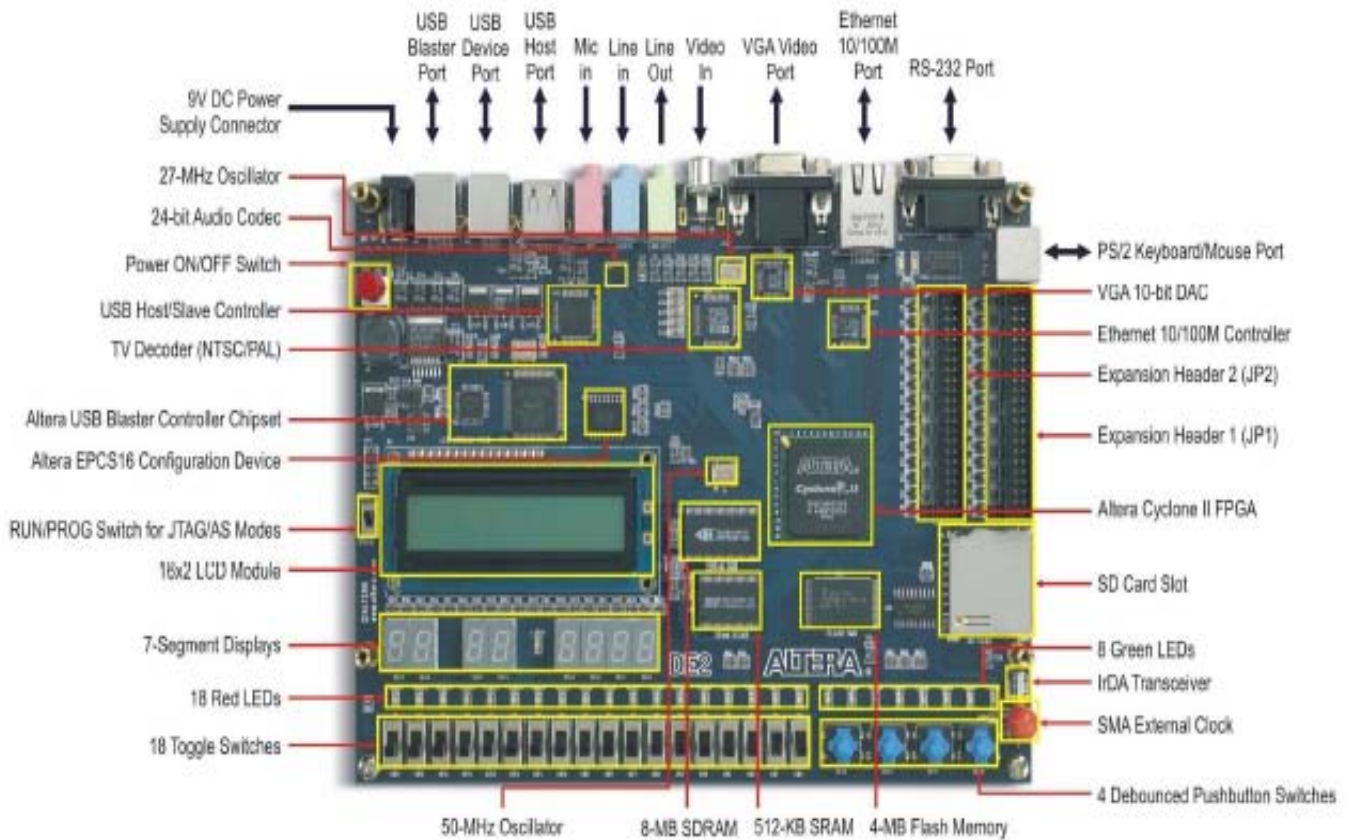


Figure 2: The DE2 board.

The DE2 board has Altera Cyclone® II 2C35 FPGA device in addition to all other input and output devices (peripherals) that are tied to the FPGA pins.

The following hardware (input and output devices) is used in the experiments:

- 4 push button switches
- 18 toggle switches
- 18 red user LEDs
- 9 green user LEDs
- 8 Common Anode Seven Segment Display
- 50-MHz oscillator and 27-MHz oscillator for clock sources

In order to use the DE2 board, the user has to be familiar with the Quartus II software. The necessary knowledge can be acquired from the first experiment (*Quartus II Introduction Using VHDL Design*).

**Pin Assignments:**

A lists of the pin names on the Cyclone II FPGA that are connected to the push button switches, toggle switches, LED's, 7-segment displays, and oscillators is given in the following tables.



**Table 1: Pin assignments for the pushbutton switches.**

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

**Table 2: Pin assignments for the toggle switches.**

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

**Table 3: Pin assignments for the LED's.**

Description	FPGA Pin No.	Signal Name
LED Red[0]	PIN_AE23	LEDR[0]
LED Red[1]	PIN_AF23	LEDR[1]
LED Red[2]	PIN_AB21	LEDR[2]
LED Red[3]	PIN_AC22	LEDR[3]
LED Red[4]	PIN_AD22	LEDR[4]
LED Red[5]	PIN_AD23	LEDR[5]
LED Red[6]	PIN_AD21	LEDR[6]
LED Red[7]	PIN_AC21	LEDR[7]



LED Red[8]	PIN_AA14	LEDR[8]
LED Red[9]	PIN_Y13	LEDR[9]
LED Red[10]	PIN_AA13	LEDR[10]
LED Red[11]	PIN_AC14	LEDR[11]
LED Red[12]	PIN_AD15	LEDR[12]
LED Red[13]	PIN_AE15	LEDR[13]
LED Red[14]	PIN_AF13	LEDR[14]
LED Red[15]	PIN_AE13	LEDR[15]
LED Red[16]	PIN_AE12	LEDR[16]
LED Red[17]	PIN_AD12	LEDR[17]
LED Green[0]	PIN_AE22	LEDG[0]
LED Green[1]	PIN_AF22	LEDG[1]
LED Green[2]	PIN_W19	LEDG[2]
LED Green[3]	PIN_V18	LEDG[3]
LED Green[4]	PIN_U18	LEDG[4]
LED Green[5]	PIN_U17	LEDG[5]
LED Green[6]	PIN_AA20	LEDG[6]
LED Green[7]	PIN_Y18	LEDG[7]
LED Green[8]	PIN_Y12	LEDG[8]

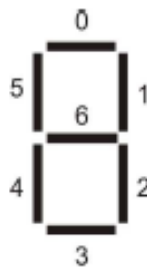


Figure 3: Position and index of each segment in a 7-segment display.

Table 4: Pin assignments for the seven segment displays

Description	FPGA Pin No.	Signal Name
Seven Segment Digit 0[0]	PIN_AF10	HEX0[0]
Seven Segment Digit 0[1]	PIN_AB12	HEX0[1]
Seven Segment Digit 0[2]	PIN_AC12	HEX0[2]
Seven Segment Digit 0[3]	PIN_AD11	HEX0[3]
Seven Segment Digit 0[4]	PIN_AE11	HEX0[4]
Seven Segment Digit 0[5]	PIN_V14	HEX0[5]
Seven Segment Digit 0[6]	PIN_V13	HEX0[6]



Seven Segment Digit 1[0]	PIN_V20	HEX1[0]
Seven Segment Digit 1[1]	PIN_V21	HEX1[1]
Seven Segment Digit 1[2]	PIN_W21	HEX1[2]
Seven Segment Digit 1[3]	PIN_Y22	HEX1[3]
Seven Segment Digit 1[4]	PIN_AA24	HEX1[4]
Seven Segment Digit 1[5]	PIN_AA23	HEX1[5]
Seven Segment Digit 1[6]	PIN_AB24	HEX1[6]
Seven Segment Digit 2[0]	PIN_AB23	HEX2[0]
Seven Segment Digit 2[1]	PIN_V22	HEX2[1]
Seven Segment Digit 2[2]	PIN_AC25	HEX2[2]
Seven Segment Digit 2[3]	PIN_AC26	HEX2[3]
Seven Segment Digit 2[4]	PIN_AB26	HEX2[4]
Seven Segment Digit 2[5]	PIN_AB25	HEX2[5]
Seven Segment Digit 2[6]	PIN_Y24	HEX2[6]
Seven Segment Digit 3[0]	PIN_Y23	HEX3[0]
Seven Segment Digit 3[1]	PIN_AA25	HEX3[1]
Seven Segment Digit 3[2]	PIN_AA26	HEX3[2]
Seven Segment Digit 3[3]	PIN_Y26	HEX3[3]
Seven Segment Digit 3[4]	PIN_Y25	HEX3[4]
Seven Segment Digit 3[5]	PIN_U22	HEX3[5]
Seven Segment Digit 3[6]	PIN_W24	HEX3[6]
Seven Segment Digit 4[0]	PIN_U9	HEX4[0]
Seven Segment Digit 4[1]	PIN_U1	HEX4[1]
Seven Segment Digit 4[2]	PIN_U2	HEX4[2]
Seven Segment Digit 4[3]	PIN_T4	HEX4[3]
Seven Segment Digit 4[4]	PIN_R7	HEX4[4]
Seven Segment Digit 4[5]	PIN_R6	HEX4[5]
Seven Segment Digit 4[6]	PIN_T3	HEX4[6]
Seven Segment Digit 5[0]	PIN_T2	HEX5[0]
Seven Segment Digit 5[1]	PIN_P6	HEX5[1]
Seven Segment Digit 5[2]	PIN_P7	HEX5[2]
Seven Segment Digit 5[3]	PIN_T9	HEX5[3]
Seven Segment Digit 5[4]	PIN_R5	HEX5[4]
Seven Segment Digit 5[5]	PIN_R4	HEX5[5]



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

Table 5: Pin assignments for the clock inputs

Description	FPGA Pin No.	Signal Name
27 MHz clock input	PIN_D13	CLOCK_27
50 MHz clock input	PIN_N2	CLOCK_50

Syllabus

	Experiment	Page	Time(hour)
1	Quartus II Introduction Using VHDL Design	5	2
2	Basic Structure of VHDL Code	21	2
3	BCD to Seven Segment Decoder	24	2
4	N-Bit Binary Adder	28	2
5	BCD Adder	32	2
6	Two Digit BCD Adder	36	2
7	Latches and Flip Flops	39	2
8	Shift Registers	43	2
9	Counters	46	2
10	State Machine Design		2



## Experiment number 1 Quartus II Introduction Using VHDL Design

**Apparatus Required:** DE2 Board, PC.

**Task:** This experiment 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 VHDL hardware description language. It makes use of the graphical user interface to invoke the Quartus II commands. Doing this experiment requires the following six steps:

1. Creating a project
2. Design entry using VHDL code.
3. Compiling a designed circuit.
4. Simulating the designed circuit.
5. Assigning the circuit inputs and outputs to specific pins on the FPGA.
6. Programming and configuring the FPGA chip on Altera's DE2 board.

### **Getting Started:**

Each logic circuit, or subcircuit, 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 experiment, we will use a directory (**exp1**). The running example for this experiment is a simple circuit for two-way light control.

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

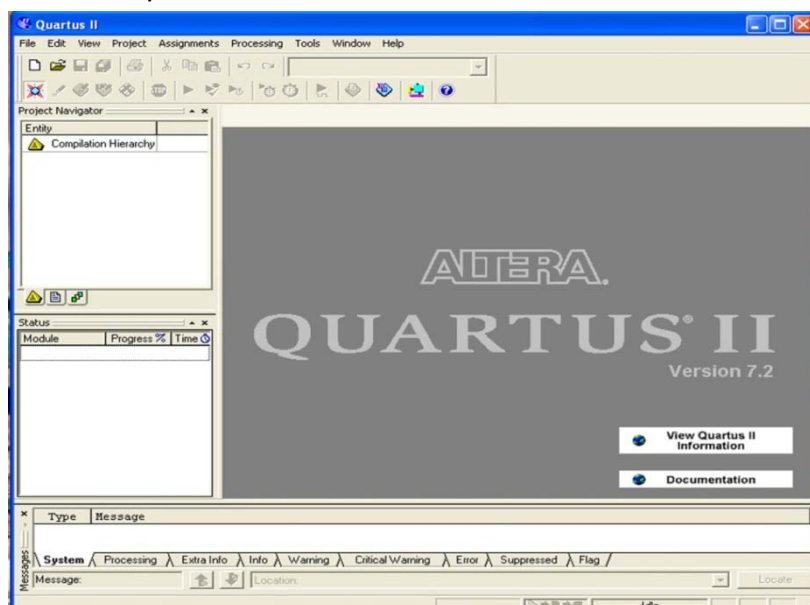


Figure 1: The main Quartus II display.



### Step 1: 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*. Create a new project as follows:

1. Select **File > New Project Wizard** to reach the window shown in fig.2, which indicates the capability of this wizard. You can skip this window in subsequent projects by checking the box (**Don't show me this introduction again**).

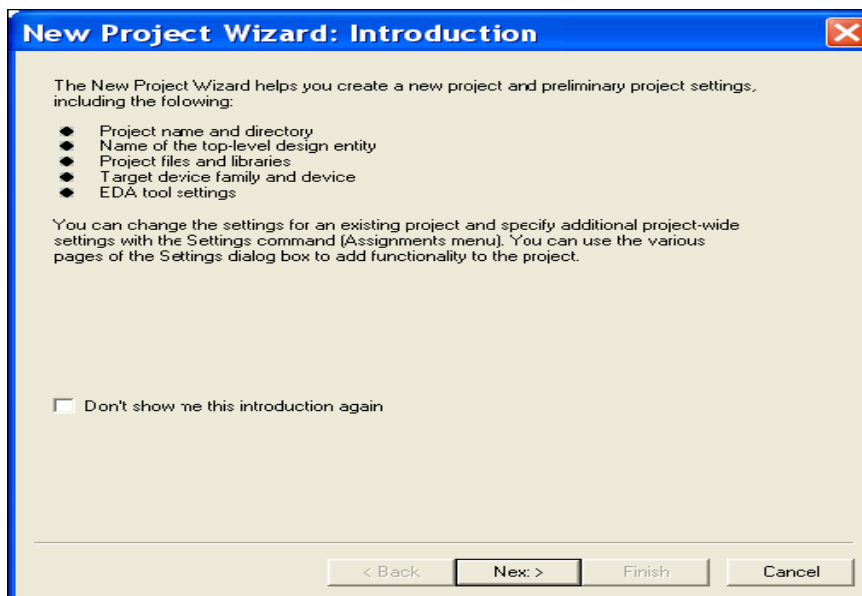


Figure 2: Tasks performed by the wizard.

2. Press **Next** to get the window in fig.3:

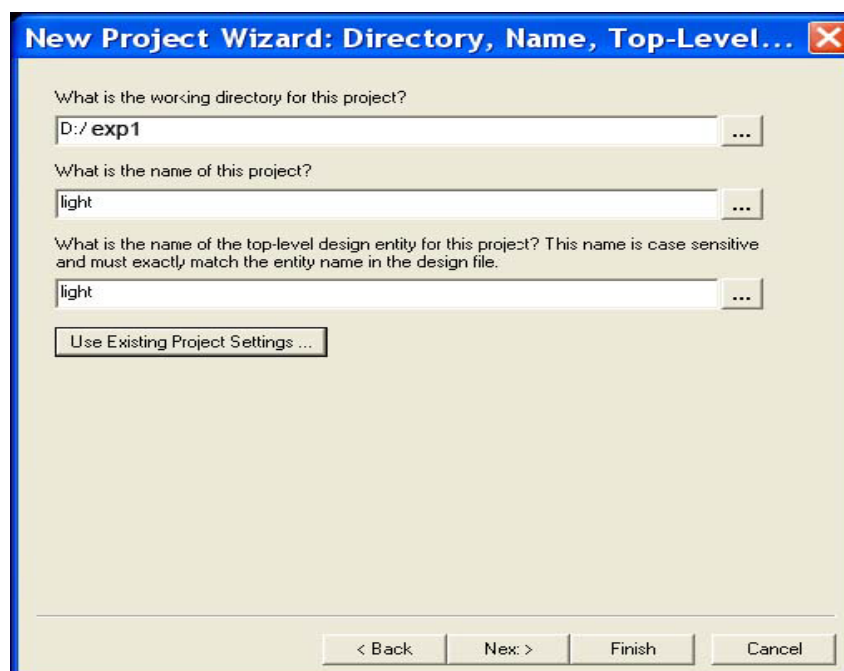


Figure 3: Creation of a new project.

3. Set the working directory to be *exp1*. The project must have a name, which is usually the same as the top-level design entity that will be included in the project. Choose *light* as the name for both the project and the top-level entity, as shown below. Press **Next**. Since we have not yet created the directory *exp1*, Quartus II software displays the pop-up box in figure 4 asking if it should create the desired directory. Click Yes, which leads to the window in figure 5.

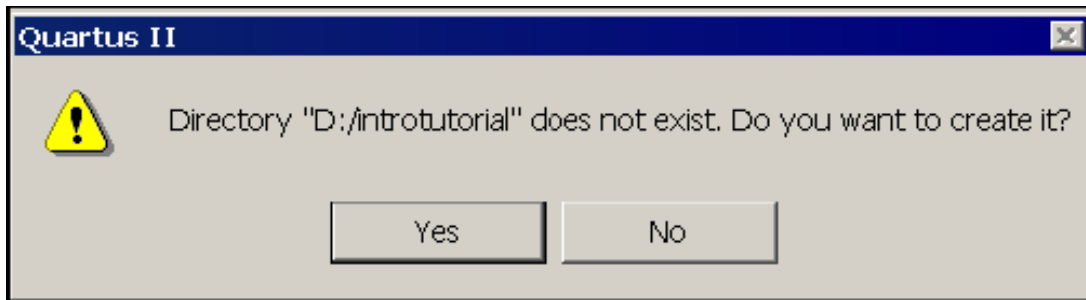


Figure 4: Quartus II software can create a new directory for the project.

4. 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 6.

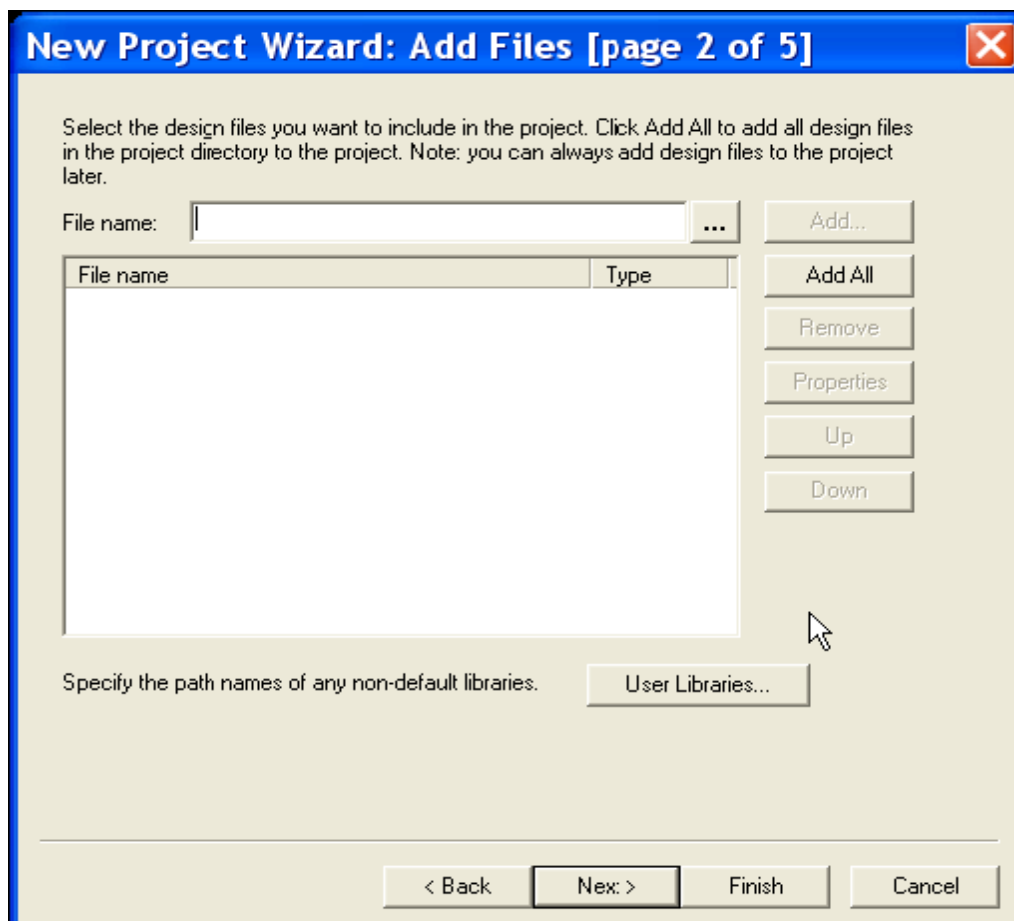


Figure 5: The wizard can include user-specified design files.

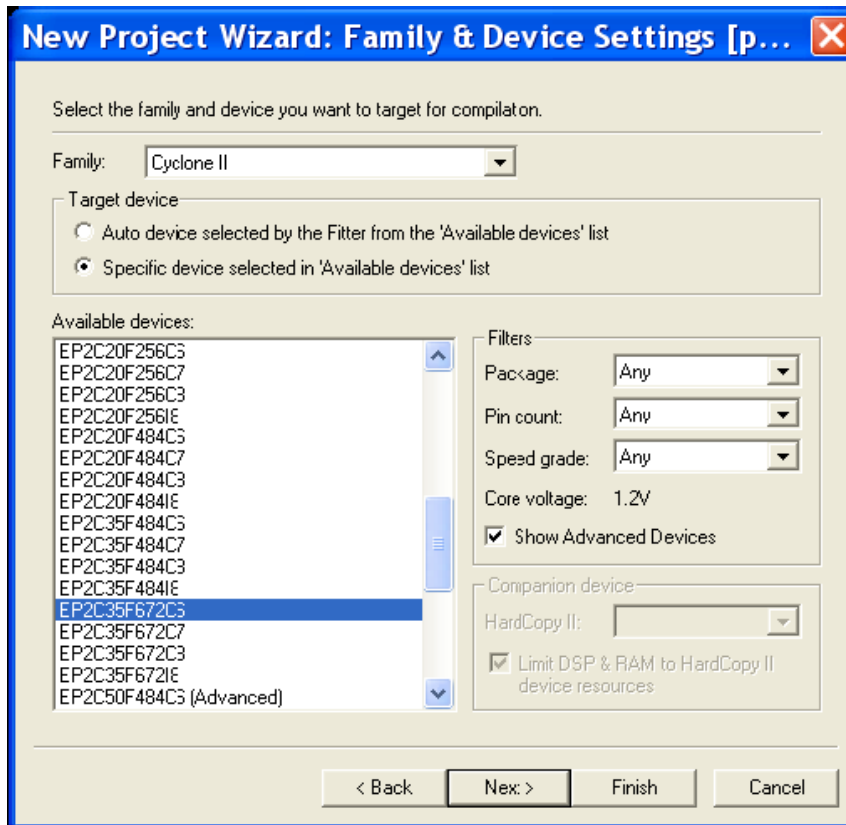


Figure 6: Choose the device family and a specific device.

5. We have to specify the type of device in which the designed circuit will be implemented. Choose **CycloneII** as the target device family. From the list of available devices, choose the device called **EP2C35F672C6** which is the FPGA used on Altera’s DE2 board. Press **Next**, which opens the window in figure 7.

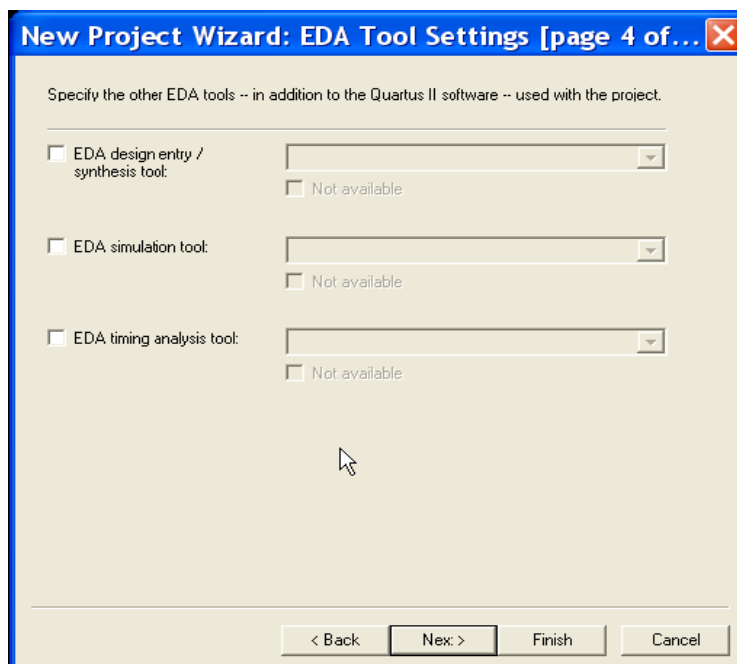


Figure 7: Other EDA tools can be specified.



- The user can specify any third-party tools that should be used. Since we will rely solely on Quartus II tools, we will not choose any other tools. Press **Next**.
- A summary of the chosen settings appears in the screen shown in figure 8.

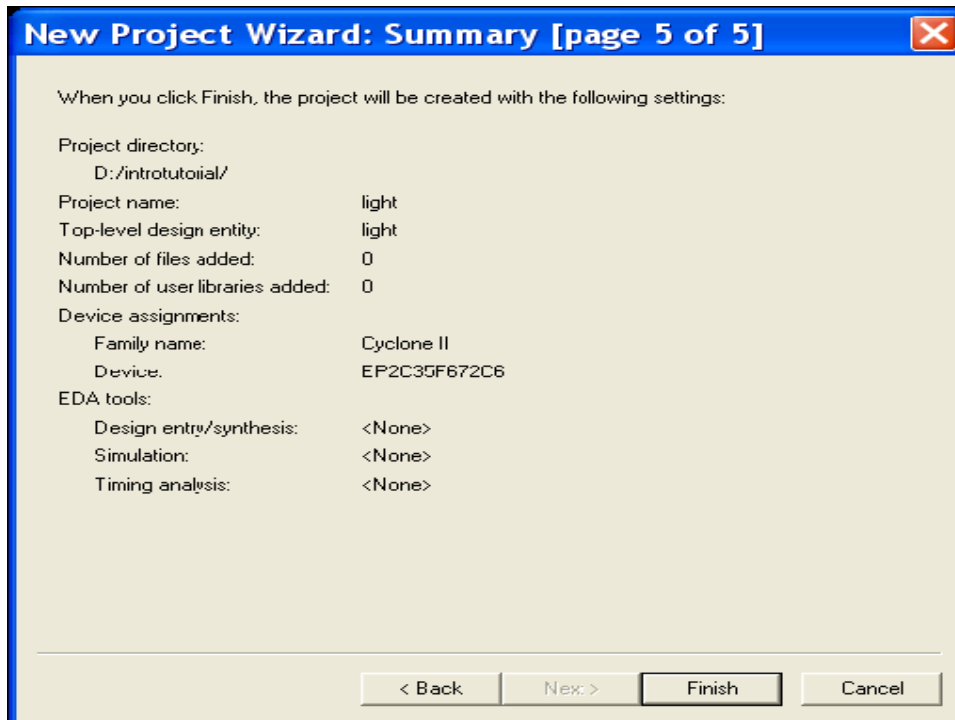


Figure 8: Summary of the project settings.

- Press **Finish**, which returns to the main Quartus II window, but with *light* specified as the new project, in the display title bar, as indicated in figure 9.

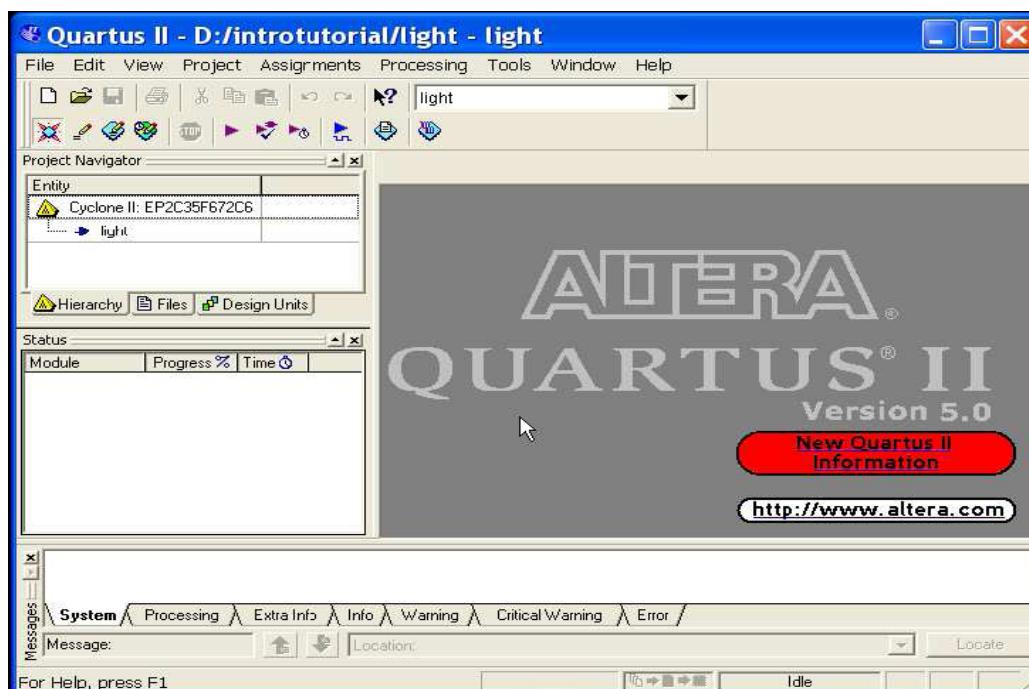
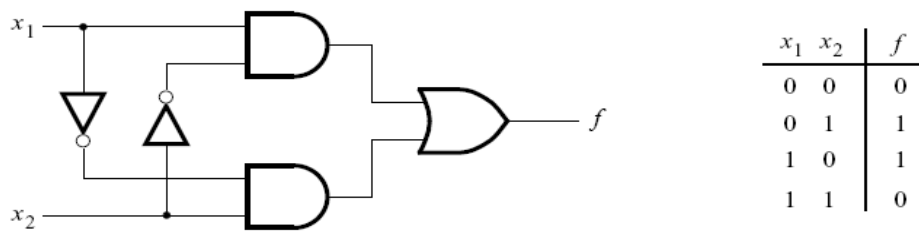


Figure 9: The Quartus II display for the created project.



**Step 2: Design Entry Using VHDL Code**

As a design example, we will use the two-way light controller circuit shown in figure 10. The circuit can be used to control a single light from either of the two 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. Note that this is just the Exclusive-OR function of the inputs x1 and x2, but we will specify it using the gates shown.



**Figure 10: The light controller circuit.**

The required circuit is described by the VHDL code in figure 11. Note that the VHDL entity is called *light* to match the name given in figure 3, which was specified when the project was created. This code can be typed into a file by using the Quartus II text editing facilities. While the file can be given any name, it is a common designers' practice to use the same name as the name of the top-level VHDL entity. The file name must include the extension *vhd*, which indicates a VHDL file. So, we will use the name *light.vhd*.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY light IS
    PORT (x1, x2      : IN  STD_LOGIC;
          f           : OUT STD_LOGIC);
END light;
ARCHITECTURE behavior OF light IS
    BEGIN
        f<= (x1 AND NOT x2) OR (NOT x1 AND x2);
    END behavior;
    
```

**Figure 11: VHDL code for the circuit in figure 10.**

1. Select **File > New** to get the window in figure 12, choose VHDL File, and click **OK**. This opens the **Text Editor** window.
2. Specify a name for the file that will be created. Select **File > Save As** to open the pop-up box depicted in figure 13. In the box labeled **Save as type** choose VHDL File. In the box labeled **File name** type *light*. Put a checkmark in the box **Add file** to current project.

3. Click **Save**, which puts the file into the directory *exp1* and leads to the **Text Editor** window shown in figure 14. Maximize the **Text Editor** window and enter the VHDL code in figure 11 into it. Save the file by typing **File > Save**.

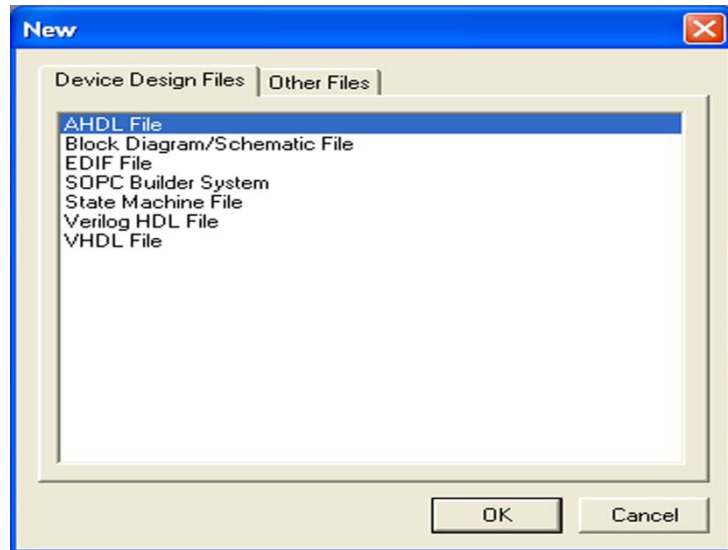


Figure 12: Choose to prepare a VHDL file.

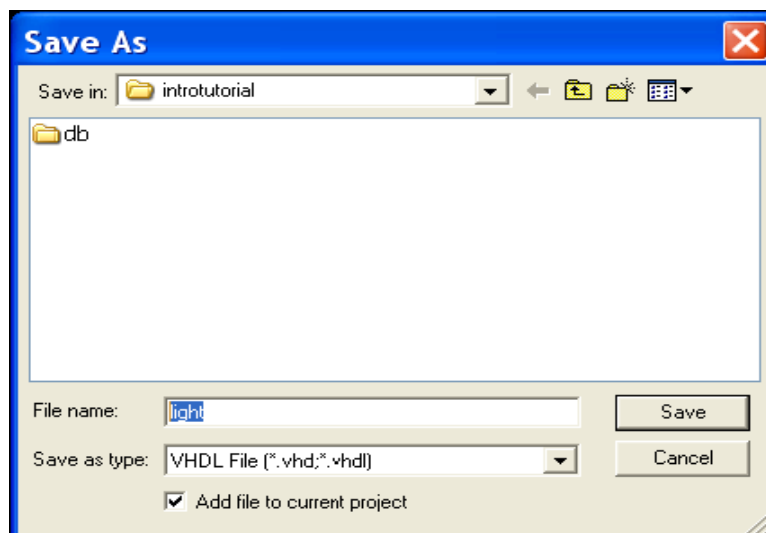


Figure 13: Name the file.

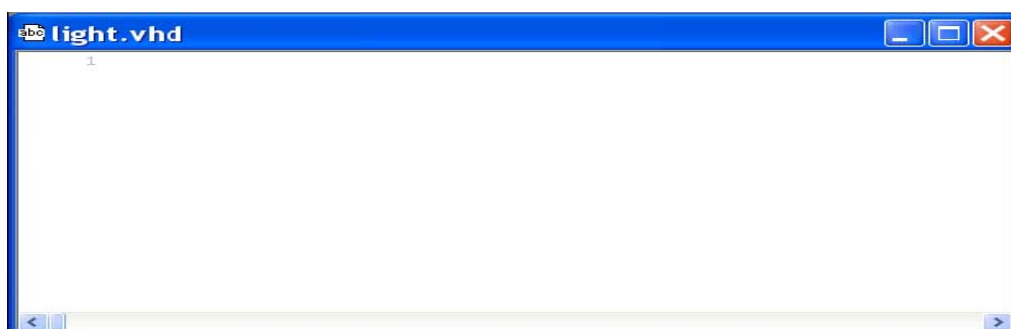



Figure 14: Text Editor Window.

### Step 3: Compiling a Designed Circuit

The VHDL code in the file *light.vhd* is processed by several Quartus II tools that analyze the code, synthesize the circuit, and generate an implementation of it for the target chip. These tools are controlled by the application program called the *Compiler*.

1. Run the Compiler by selecting **Processing > Start Compilation**, or by clicking on the toolbar icon that looks like a purple triangle .
2. The compilation moves through various stages, its progress is reported in a window on the left side of the Quartus II display.
3. 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 15.
4. In the message window, at the bottom of the figure, various messages are displayed. In case of errors, there will be appropriate messages given.
5. When the compilation is finished, a compilation report is produced.

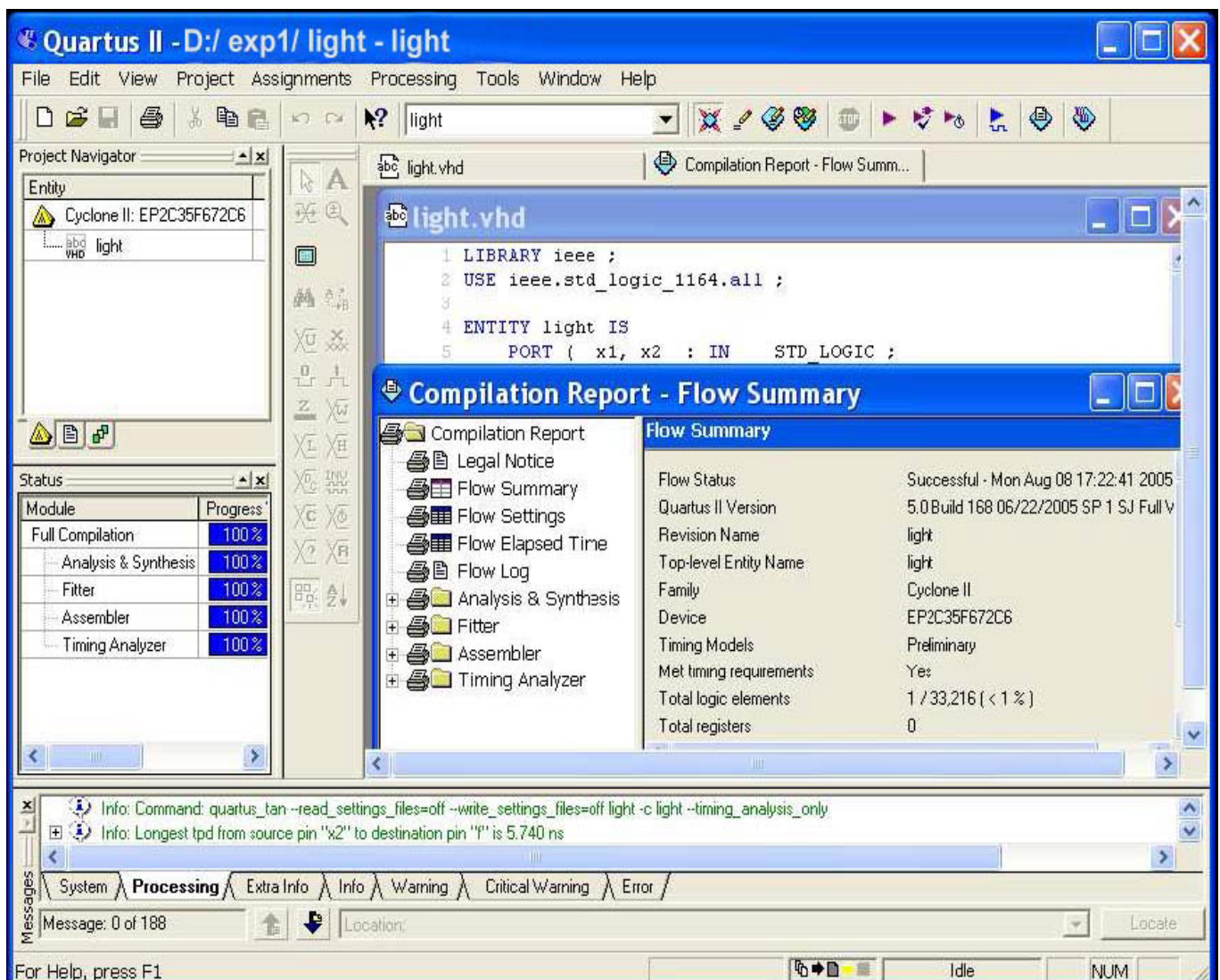


Figure 15: Display after a successful compilation.

#### **Step 4: 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. A designed circuit can be simulated in two ways. The simplest way is to assume that logic elements and interconnection wires in the FPGA are perfect, thus causing no delay in propagation of signals through the circuit. This is called *functional simulation*. A more complex alternative is to take all propagation delays into account, which leads to *timing simulation*. Typically, functional simulation is used to verify the functional correctness of a circuit as it is being designed.

1. Open the Waveform Editor window by selecting **File > New**, which gives the window shown in figure 16.

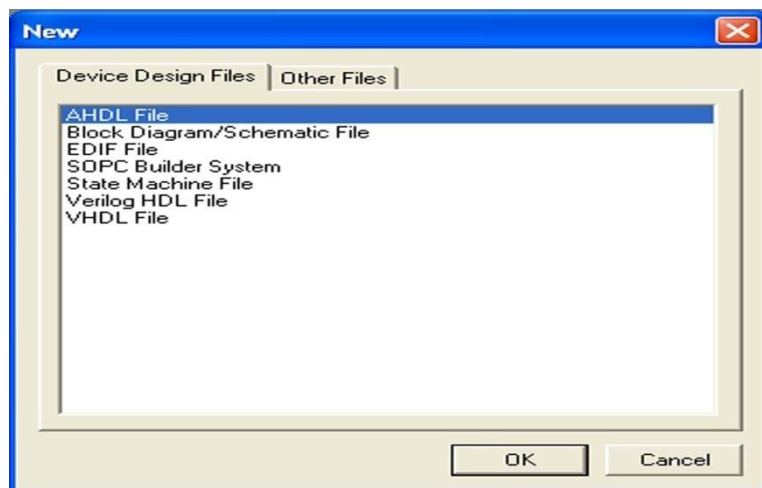


Figure 16: Need to prepare a new file.

2. Click on the **Other Files** tab to reach the window displayed in figure 17. Choose **Vector Waveform File** and click **OK**

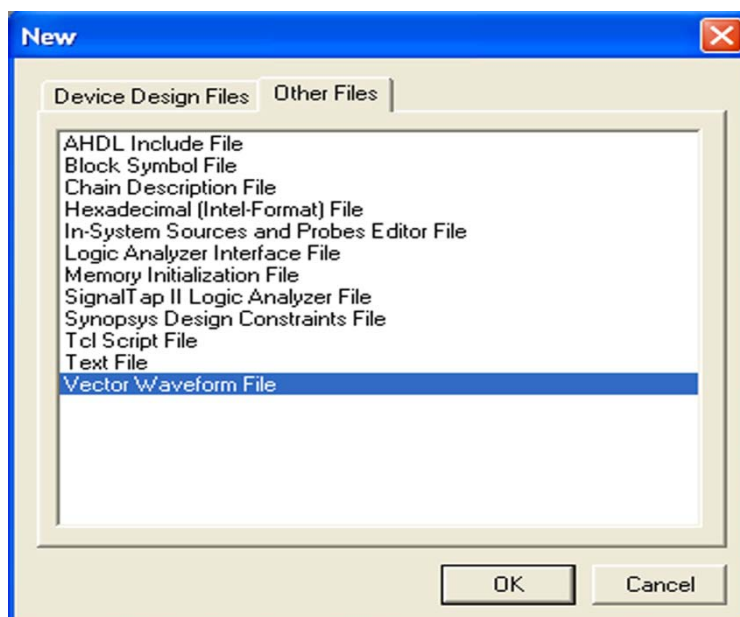


Figure 17: Choose to prepare a test-vector file.





- The **Waveform Editor** window is depicted in figure 18. **Save** the file under the name *light.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.

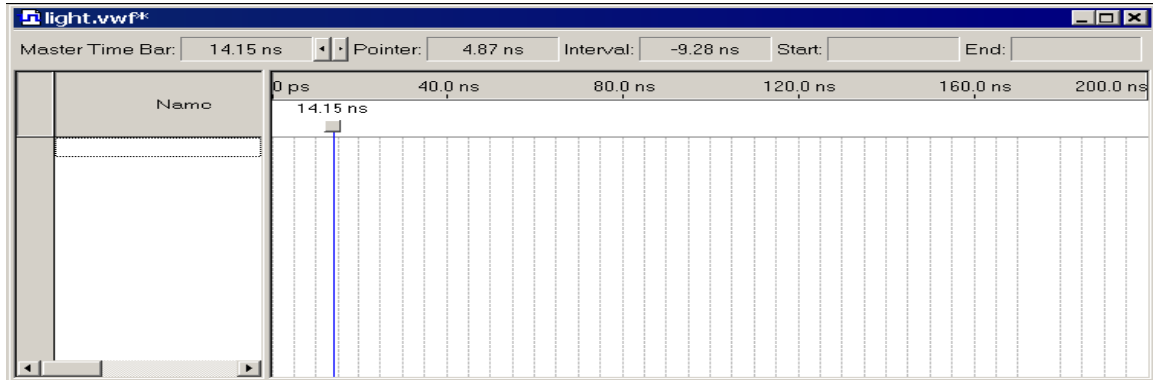


Figure 18: The Waveform Editor Window.

- To include the input and output nodes of the circuit to be simulated. Click **Edit > Insert Node or Bus** to open the window in figure 19.

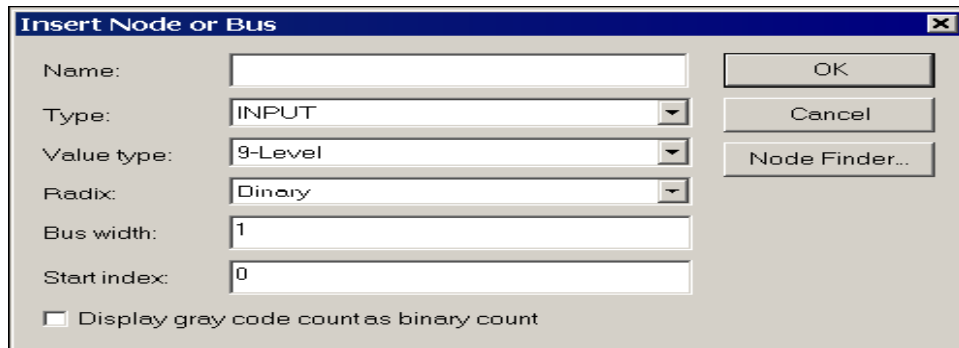


Figure 19: The Insert Node or Bus dialogue.

- Click on the button labeled **Node Finder** to open the window in Figure 20. The Node Finder utility has a filter used to indicate what type of nodes are to be found. Since we are interested in input and output pins, set the filter to **Pins: all**. Click the **List** button to find the input and output nodes as indicated on the left side of the figure.

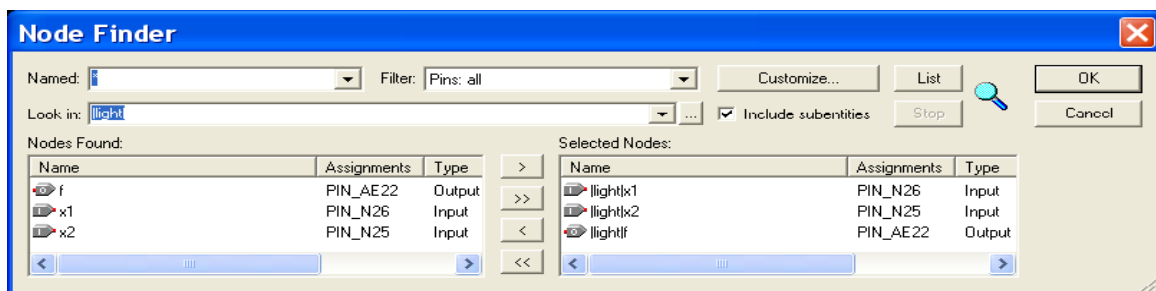


Figure 20: Selecting nodes to insert into the Waveform Editor.

- Click on the  $x1$  signal in the **Nodes Found** box in figure 20, 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 19. This leaves a fully displayed Waveform Editor window, as shown in figure 21.

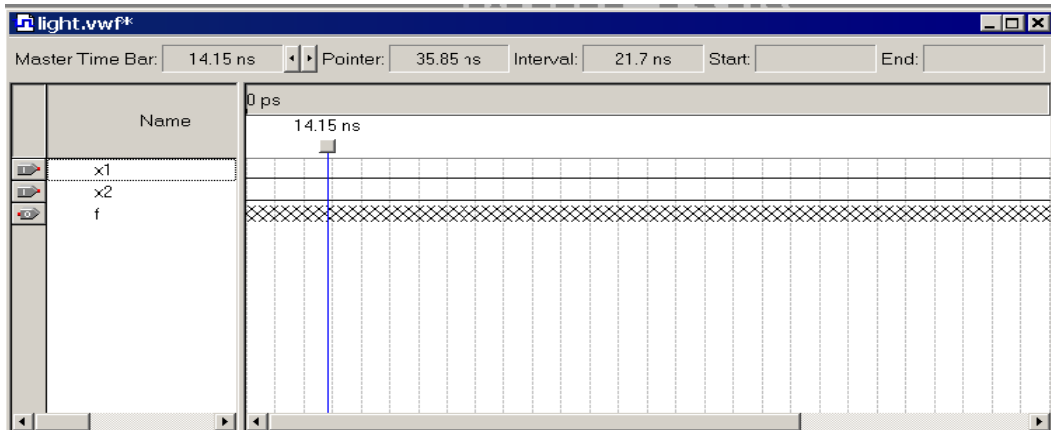


Figure 21: The nodes needed for simulation.

- 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.
- 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 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 10. This should produce the image in figure 22. 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.

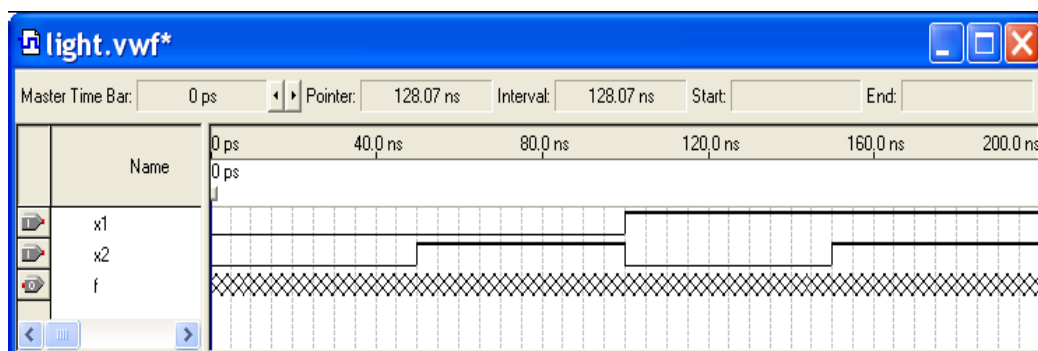


Figure 22: Setting of test values.

10. Select **Assignments > Settings** to open the Settings window. On the left side of this window click on **Simulator** to display the window in figure 23, choose **Functional** as the simulation mode, and click **OK**.

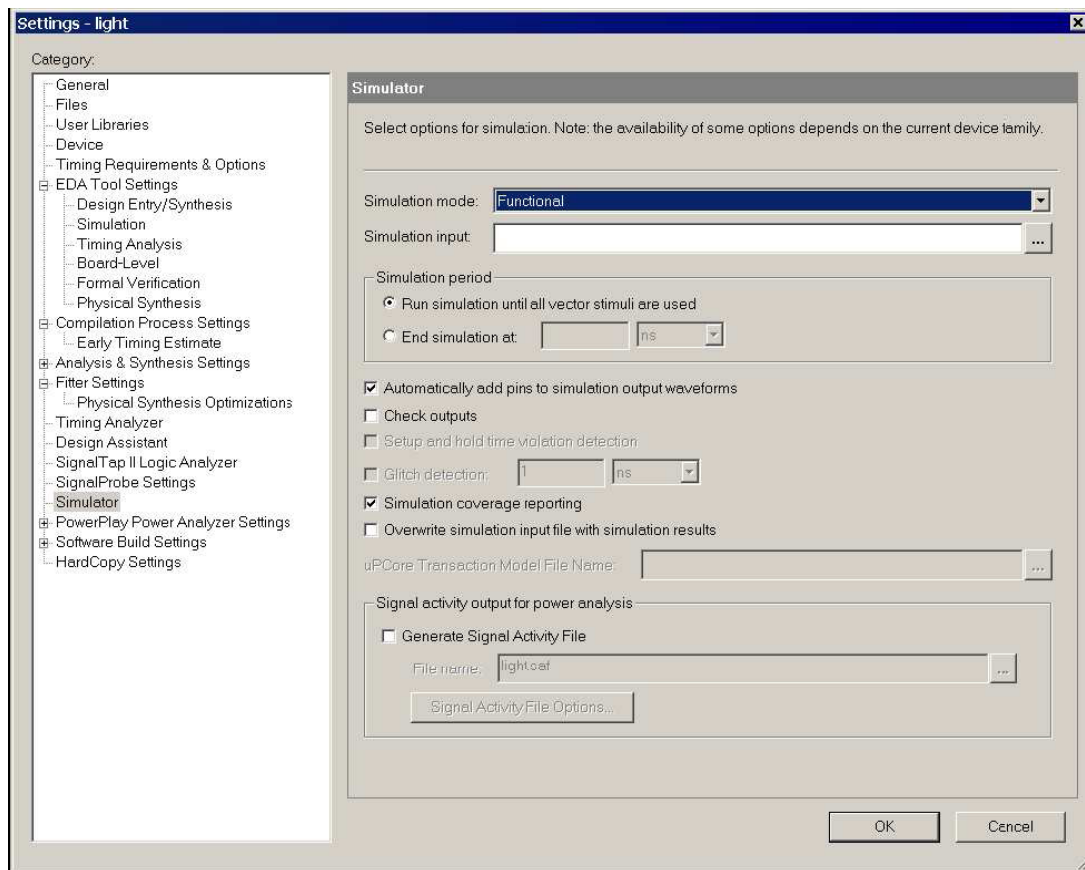


Figure 23: Specifying the simulation mode.

11. Select **Processing > Generate Functional Simulation Netlist**.

12. A simulator run is started by **Processing > Start Simulation**. Quartus II software indicates its successful completion and displays a **Simulation Report** illustrated in figure 24. 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 10.

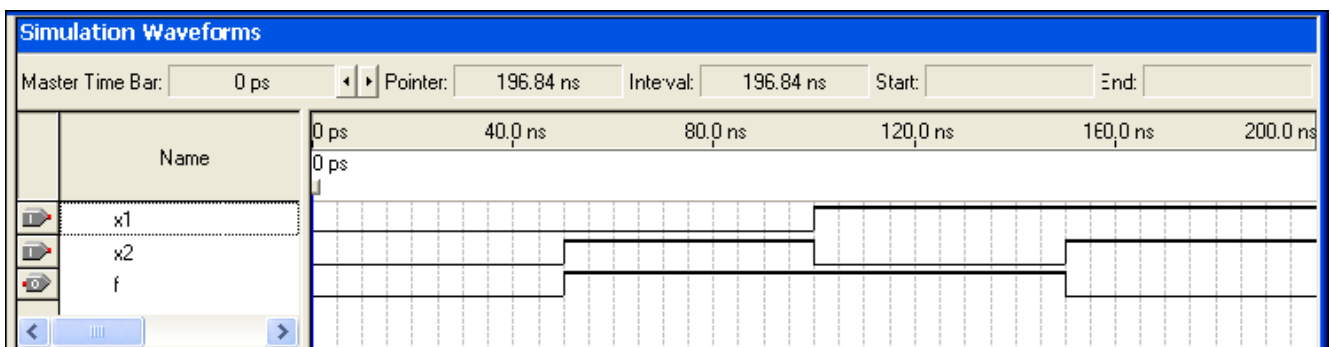


Figure 24: The result of Functional Simulation.

### Step 5: Pin Assignment

During the compilation, 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 **SW0** and **SW1**, to provide the external inputs x1 and x2. These switches are connected to the FPGA pins **N25** and **N26**, respectively. We will connect the output f to the green light-emitting diode labeled LEDG0, which is hardwired to the FPGA pin AE22.

1. Select **Assignments > Assignment Editor** to reach the window in figure 25.

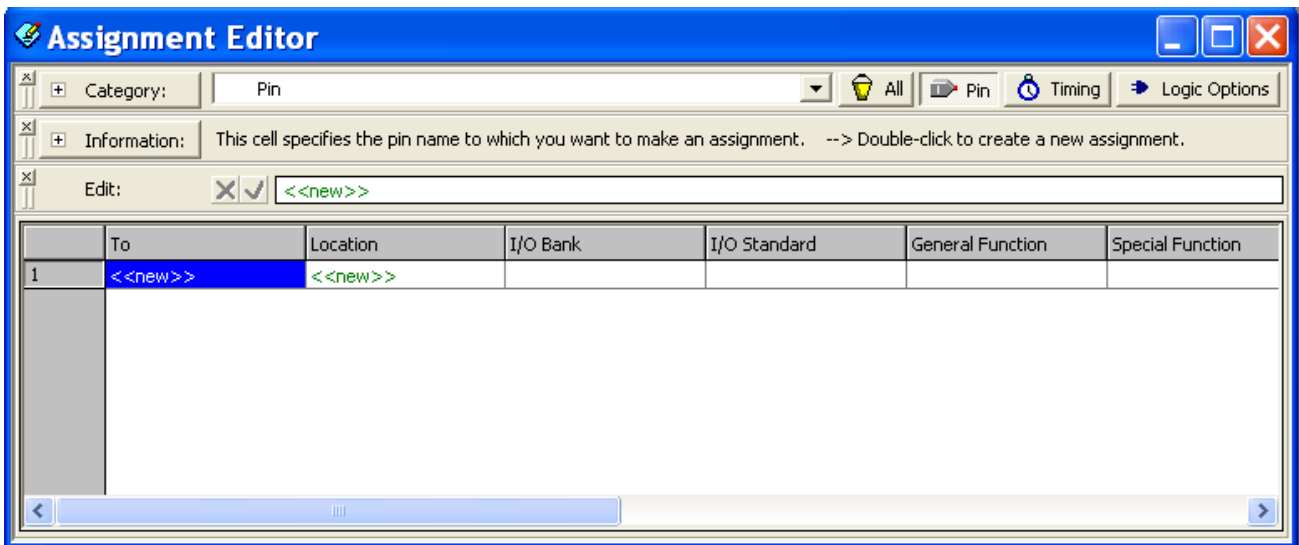


Figure 25: The Assignment Editor Window

2. 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 26 will appear. Click on **x1** as the first pin to be assigned; this will enter **x1** in the displayed table.

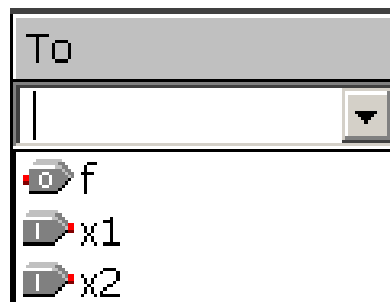


Figure 26: The drop-down menu displays the input and output names.

3. Double-click on the box to the right of this new **x1** entry, in the column labeled **Location**. Type the name of the pin (**N25**) in the **Location** box. Use the same procedure to assign input **x2** to pin **N26** and output **f** to pin **AE22**, which results in the image in figure 27. To save the assignments made, choose **File > Save**.
4. Recompile the circuit, so that it will be compiled with the correct pin assignments.

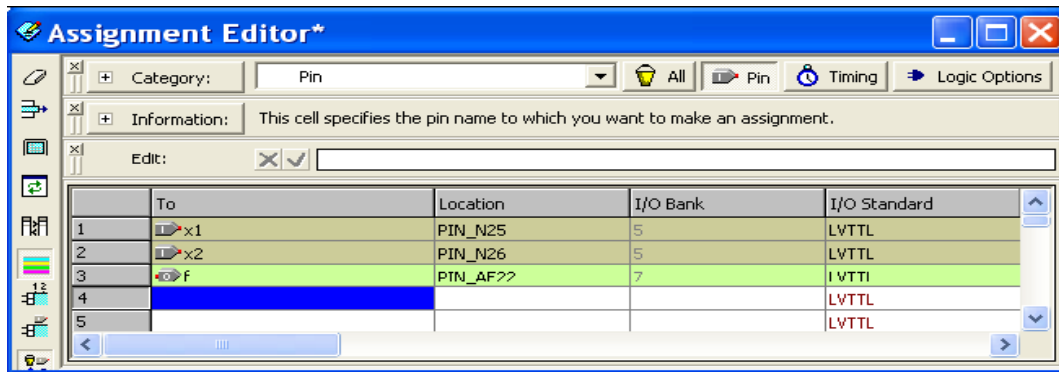


Figure 27: The Complete Assignment

### Step 6: 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. Altera’s DE2 board allows the configuration to be done in two different ways, known as **JTAG** and **AS** modes. We will use the **JTAG** mode in our experiments. In the JTAG mode, the configuration data is loaded directly into the FPGA device. 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.

1. Make sure that the USB cable is properly connected.
2. Turn on the power supply switch on the board.
3. Flip the **RUN/PROG** switch into the **RUN** position.
4. Select **Tools > Programmer** to reach the window in figure 28. 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. Observe that the configuration file *light.sof* is listed in the window in Figure 28.
5. Click on the **Program/Configure** check box in figure 28.
6. Press **Start** in the window in figure 28. An **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, then check to ensure that the board is properly powered on.

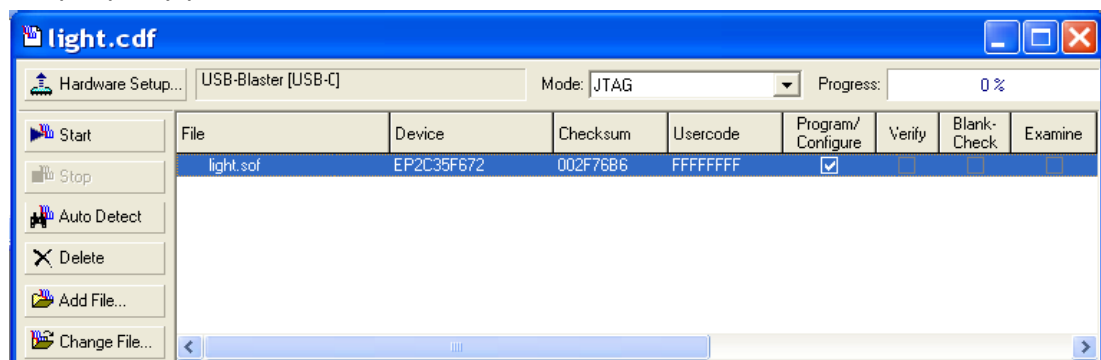


Figure 28: The Programmer Window

## Experiment number 2 Basic Structure of VHDL Code

**Apparatus:** PC and DE2 Board

**Task:** Learn the basic parts of VHDL components

**Theory:**

VHDL is a *hardware description language*; it describes the behavior of an electronic circuit or system from which the physical circuit or system can be implemented. VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an abbreviation for Very High Speed Integrated Circuit. A circuit or sub-circuit described with VHDL code is called a *design entity* or just *entity*. Figure 1 shows the general structure of an entity. It has three main parts:

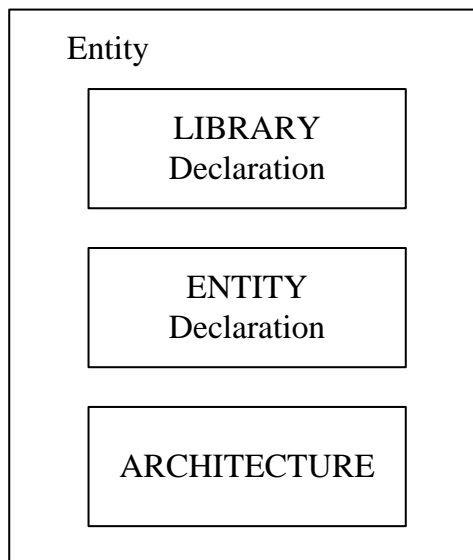


Figure 1: The general structure of a VHDL design entity

- 1- **LIBRARY** declarations: Contains a list of all libraries to be used in the design.

General form:     **LIBRARY** library\_name ;  
                      **USE** library name.package\_name.all ;

- 2- **ENTITY** declaration: Specifies the input and output pins of the circuit. The name of the entity can be any legal VHDL name. The input and output signals are specified using the keyword **PORT**. Whether each port (pin) is an input, output, or bidirectional are specified by the *mode* of the port. The available modes are summarized in table 1.

General form:     **ENTITY** entity\_name **IS**  
                      **PORT** (signal\_name, signal\_name, ...     : **mode**     **type\_name**;  
                                  signal\_name, signal\_name, ...     : **mode**     **type\_name**);  
                      **END** entity\_name;



**3- ARCHITECTURE** provides the circuit details for an entity. It has two main parts: *the declarative region* and *architecture body*.

- The *declarative region* appears preceding the BEGIN keyword. It can be used to declare signals, user defined data types, constants, components, and attributes.
- The functionality of the entity is specified in the *architecture body*, which follows the BEGIN keyword. This specification involves statements that define the logic function in the circuit.

```

General form:  ARCHITECTURE architecture_name OF entity_name IS
                [SIGNAL declarations]
                [CONSTANT declarations]
                [TYPE declarations]
                [COMPONENT declarations]
                [ATTRIBUTE declarations]
                BEGIN
                COMPONENT instantiation statements;
                CONCURRENT ASSIGNMENT statements;
                PROCESS statements;
                GENERATE statements;
                END architecture_name;
    
```

**Table 1: The possible modes for signals that are entity ports**

Mode	Purpose
IN	Used for a signal that is an input to an entity.
OUT	Used for signal that is an output from an entity. The value of the signal can not be used inside the entity. This means that in an assignment statement, the signal can appear only to the left of the <= operator.
INOUT	Used for a signal that is both an input to an entity and an output from the
BUFFER	Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement, the signal can be appear both on the left and right sides of the <= operator.

**Design Example:**

Figure 2 shows the block diagram, and logic diagram of a full adder circuit. Figure 3 gives the VHDL code for the full adder of figure2.

The entity declaration specifies the input and output signals. The input port **Cin** is the carry-in, and the bits to be added are the input ports **x** and **y**. The output ports are the sum, **s**, and the carry-out, **Cout**. The entity represents the block diagram of the full adder (figure 2a).

The architecture defines the functionality of the full adder using logic equations. The logic equations used in the architecture body is the simple type of concurrent assignment statements. The architecture represents the logic diagram of the full adder (figure 2b).

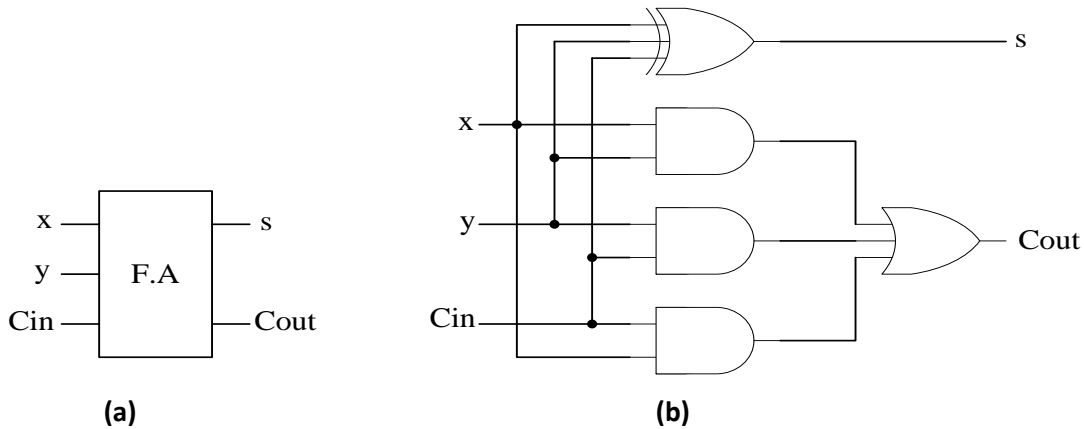


Figure 2: a) Block diagram of full adder  
 b) Logic diagram of full adder

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY fulladder IS
PORT ( cin , x , y      :IN          STD_LOGIC ;
      s , cout         :OUT         STD_LOGIC );
END fulladder ;
ARCHITECTURE behavior OF fulladder IS
BEGIN
    s<= x XOR y XOR cin ;
    cout<=(x AND y) OR (x AND cin) OR (y AND cin) ;
END behavior;
    
```

Figure 3: VHDL code of a full adder.

**Procedure:**

1. Create a new Quartus II project for the full adder circuit. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in Figure 3 and include it in your project.
3. Compile the design, then select **Tools > Netlist Viewer > RTL Viewer**.
4. Include in your project the following pin assignments for the DE2 board, and recompile the project.

Port Name	FPGA Pin No.	Description on DE2 Board
cin	PIN N25	Toggle Switch[0]
x	PIN N26	Toggle Switch[1]
y	PIN P25	Toggle Switch[2]
s	PIN AE23	LED Red[0]
cout	PIN AF23	LED Red[1]





5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit (verify its truth table) by toggling the switches and observing the LEDs.

**Discussion:**

1. Specify the three basic parts of the VHDL code of the full adder.
2. Write discrete VHDL codes for (half adder, half subtracter, and full subtracter).



### Experiment Number 3 BCD to Seven Segment Decoder

**Apparatus:** PC and DE2 Board

**Task:** Learning the following: Using STD\_LOGIC\_VECTOR types, the use of Selected *Signal Assignment* and OTHERS, and Seven segment display.

#### **Theory:**

##### **1. STD\_LOGIC\_VECTOR type:**

To use this type, we must include the two statements in the VHDL code

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;
```

These statements provide access to the std\_logic\_1164 package in the library ieee, which defines the STD\_LOGIC data type. The following values are legal for STD\_LOGIC: 0, 1, Z, -, L, H, U, X, and W. Only the first four values are useful for logic circuits. The value Z represent high impedance, and - stands for don't care. The STD\_LOGIC\_VECTOR represents an array of STD\_LOGIC.

**Example:** SIGNAL c : STD\_LOGIC\_VECTOR (1 TO 4);  
SIGNAL x : STD\_LOGIC\_VECTOR (3 DOWNTO 0);

c and x are both 4-bit number. Now if c= 0101 and x=0100 then:

- c (1) = 0, c (2) = 1, c (3) = 0, c (4) = 1, x (3) = 0, x (2) = 1, x (1) = 0, x (0) = 0.

##### **2. Selected Signal Assignment and OTHERS:**

The selected signal assignment is a type of concurrent assignment statement; it is used to set the value of a signal to one of several alternatives based on a selection criterion. The general form is:

```
WITH expression SELECT  
    signal name <= expression WHEN constant value ,  
    expression WHEN constant value ,  
    .  
    .  
    expression WHEN constant value ;
```

The statement(S <= OTHERS='0') set each bit of S to 0.

**Example:** SIGNAL x1, x2, Sel, f : STD\_LOGIC;  
WITH Sel SELECT  
 f <= x1 WHEN '0' ,  
 x2 WHEN OTHERS ;

This Example describes a 2 to 1 multiplexer with Sel as the select input. In a selected signal assignment, all possible values of the select input, Sel in this case, must be explicitly listed in the code. The word **OTHERS** provides an easy way to meet this requirement. **OTHERS** represent all possible values not already listed. In this case the other possible values are 1, Z, -, and so on.



**3. Seven Segment Display:**

Seven segment displays are used in many type of products. These displays are used with logic circuits that decode BCD number and activate the appropriate digit on the display. There are two types of seven segment display; *common anode* and *common cathode*.

- In *common anode* type, the segments will light if logic 0 applied to its terminals.
- In *common cathode* type, the segments will light if logic 1 applied to its terminals.

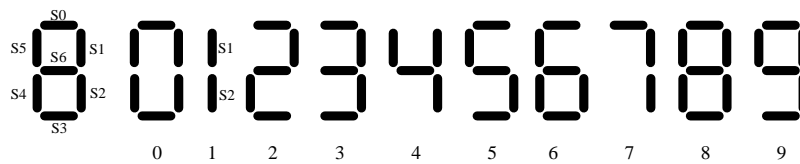
The seven segment displays on the DE2 board are of common anode type.

**Design Example:**

A BCD to seven segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in a display indicator used for displaying the decimal digit in a familiar form. The seven outputs of the decoder (S0, S1, S2, S3, S4, S5, and S6) select the corresponding segments in the display, as shown in figure1. The six invalid combinations should result in blank display.

Figure2 shows the VHDL code for the design example, the logic expressions for the seven segments outputs (S0, S1, S2, S3, S4, S5, S6) must be derived using Karnaugh map.

Another VHDL code style for the design example is shown in figure3; this style uses selected signal assignment instead of simple assignment statement. This style represents the truth table of the functions (table1), therefore the minimization step using Karnaugh map not required.



**Figure 1: Seven Segment Display**

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY sevensegment IS
    PORT (a          : IN          STD_LOGIC_VECTOR (3 DOWNTO 0) ;
          s          :OUT         STD_LOGIC_VECTOR(0 TO 6));
END sevensegment;
ARCHITECTURE behavior OF sevensegment IS
    BEGIN
        s(0)<= .....;
        s(1)<= .....;
        s(2)<= .....;
        s(3)<= .....;
        s(4)<= .....;
        s(5)<= .....;
        s(6)<= .....;
    END behavior;
    
```

**Figure 2: VHDL Code for BCD to Seven Segment Decoder using Simple Assignment Statement**



```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY sevensegment IS
    PORT (a          : IN          STD_LOGIC_VECTOR (3 DOWNTO 0);
          s          : OUT          STD_LOGIC_VECTOR(0 TO 6));
END sevensegment;
ARCHITECTURE behavior OF sevensegment IS
    BEGIN
        WITH a SELECT
            s<= "0000001" WHEN "0000" ,
               "1001111" WHEN "0001" ,
               "0010010" WHEN "0010" ,
               "0000110" WHEN "0011" ,
               "1001100" WHEN "0100" ,
               "0100100" WHEN "0101" ,
               "0100000" WHEN "0110" ,
               "0001111" WHEN "0111" ,
               "0000000" WHEN "1000" ,
               "0000100" WHEN "1001" ,
               "1111111" WHEN OTHERS;
    END behavior;
    
```

Figure 3: VHDL Code for BCD to Seven Segment Decode using Selected Signal Assignments

Table 1: Truth table of BCD to common anode seven segment display

BCD Code				Seven Segment outputs						
A3	A2	A1	A0	S0	S1	S2	S3	S4	S5	S6
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1



**Procedure:**

1. Create a new Quartus II project for the BCD to seven segment decoder circuit. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in Figure 3 and include it in your project.
3. Compile the project.
4. Make the following pin assignments for the DE2 board, and recompile the project.

Port Name	FPGA Pin No.	Description on DE2 Board
a(0)		Toggle Switch[0]
a(1)		Toggle Switch[1]
a(2)		Toggle Switch[2]
a(3)		Toggle Switch[3]
s(0)		Seven Segment Digit 0[0]
s(1)		Seven Segment Digit 0[1]
s(2)		Seven Segment Digit 0[2]
s(3)		Seven Segment Digit 0[3]
s(4)		Seven Segment Digit 0[4]
s(5)		Seven Segment Digit 0[5]
s(6)		Seven Segment Digit 0[6]

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the seven segment display.

**Discussion:**

1. Derive the logic expressions for the seven segment display outputs and complete the VHDL code of figure 2. Which code style you prefer (figure 2 or figure 3), why?
2. Write a VHDL code for a circuit that converts gray code to BCD code using two the styles of figure 2 and figure 3.



## Experiment Number 4 N-Bit Binary Adder

**Apparatus:** PC and DE2 Board

**Task:** Learning the following:

1. COMPONENT declaration and instantiation.
2. FOR GENERATE statement.
3. Defining an ENTITY with GENERIC

### Theory:

#### 1. COMPONENT Declaration and Instantiation:

A VHDL code defined in one source code file can be used as a subcircuit in another source code file. In VHDL jargon the subcircuit is called a component. A subcircuit must be declared using a *component declaration*. This statement specifies the name of the subcircuit and gives the names of its input and output ports. The component declaration can appear either in the declaration region of an architecture or in package declaration.

*General form:*

```
COMPONENT component_name
  GENERIC ( parameter_name: integer: = default value ;
           parameter_name: Integer: = default value);
  PORT (signal_name, signal_name, ... : mode type_name;
        signal_name, signal_name, ... : mode type_name);
END COMPONENT;
```

Once a component declaration is given, the component can be *instantiated* as a subcircuit. This done using *component instantiation* statement.

*General form:*

#### 1- Named Association:

Instance\_name : component name

**PORT MAP** ( formal\_name=> actual\_name, formal\_name=> actual\_name , ..... ) ;

#### 2- Positional Association:

Instance\_name : component name

**PORT MAP** ( actual\_name, actual\_name , ..... ) ;

Each *formal name* is the name of a port in the subcircuit. Each *actual name* is the name of a signal in the code that instantiate the subcircuit. The signal names following PORT MAP keyword can be written in two ways:



- IN *name association*, the order of the signal listed after PORT MAP keyword does not have to be the same as the order of the ports in the corresponding COMPONENT declaration.
- IN *positional association*, the signal names following the PORT MAP keyword are given in the same order as in the COMPONENT declaration, and then the formal name is not needed.

The VHDL code of figure1 represents the design entity for 4-bit adder which uses the VHDL code of full adder in experiment number2 as a component.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY adder IS
    PORT (
        cin      : IN  STD_LOGIC;
        x, y     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
        s        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        cout     : OUT STD_LOGIC);
END adder;
ARCHITECTURE behavior OF adder IS
    SIGNAL c : STD_LOGIC_VECTOR(1 TO 3);
    COMPONENT fulladder
        PORT (cin, x, y      : IN  STD_LOGIC;
              s, cout      : OUT STD_LOGIC);
    END COMPONENT;
BEGIN
    stage0: fulladder PORT MAP ( cin , x(0), y(0), s(0), c(1) );
    stage1: fulladder PORT MAP ( c(1), x(1), y(1), s(1), c(2) );
    stage2: fulladder PORT MAP ( c(2), x(2), y(2), s(2), c(3) );
    stage3: fulladder PORT MAP ( c(3), x(3), y(3), s(3), cout);
END behavior;
```

Figure 1: VHDL code for a four bit adder, using component instantiation

## 2. FOR –GENERATE statement:

The FOR-GENERATE statement provides a convenient way of repeating either a logic expression or a component instantiation. The code of figure2 illustrates its use for component instantiation. The code in the figure is equivalent to the code given in figure1.



*General form:*      generate\_label :  
                           FOR index\_variable IN range GENERATE  
                                           statement ;  
                                           statement ;  
                           END GENERATE;

### 3. Defining an ENTITY with GENERICS:

The code in figure2 represent 4-bit binary adder. It is possible to make this code more general by introducing a parameter in the code that represents the number of bits in the adder. In VHDL jargon such parameter is called a GENERIC. Figure3 gives the code for an n-bit adder entity named addern. The GENERIC keyword is used to define the number of bits, n, to be added. This parameter is used in the code, both in the definitions of the signals X, Y, and S and in the FOR-GENERATE statement that instantiate the n full adders.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY adder IS
    PORT (cin      : IN      STD_LOGIC;
          x, y     : IN      STD_LOGIC_VECTOR (3 DOWNTO 0);
          s        : OUT     STD_LOGIC_VECTOR (3 DOWNTO 0);
          cout     : OUT     STD_LOGIC );
END adder;
ARCHITECTURE behavior OF adder IS
    SIGNAL c: STD_LOGIC_VECTOR (0 TO 4);
    COMPONENT fulladder
        PORT (cin, x, y      : IN      STD_LOGIC;
              s, cout      : OUT     STD_LOGIC);
    END COMPONENT;
BEGIN
    c(0) <= cin ;
    Generate_label:
    FOR i IN 0 TO 3 GENERATE
        bit: fulladd PORT MAP ( c(i), x(i), y(i), s(i), c(i+1) ) ;
    END GENERATE;
    cout <= c(4) ;
END behavior ;
```

**Figure2: VHDL code for 4-bit adder using FOR-GENERATE statement**





```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY adder IS
    GENERIC (n : INTEGER := 4) ;
    PORT (
        Cin      : IN  STD_LOGIC ;
        X, Y     : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
        S        : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
        Cout     : OUT STD_LOGIC ) ;
END adder ;
ARCHITECTURE behavior OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;
    COMPONENT fulladd
        PORT (Cin, x, y      : IN  STD_LOGIC ;
              s, Cout      : OUT STD_LOGIC) ;
    END COMPONENT ;
    BEGIN
        C(0) <= Cin ;
        Generate_label:
        FOR i IN 0 TO 3 GENERATE
            bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
        END GENERATE ;
        Cout <= C(n) ;
    END behavior ;
```

Figure 3: VHDL code for n-bit adder

**Procedure:**

1. Create a new Quartus II project for the n-bit binary adder. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in Figure 3 and include it in your project.
3. Compile the project.
4. Make the following pin assignments for the DE2 board, and recompile the project.



Port Name	FPGA Pin No.	Description on DE2 Board
X(0)		Toggle Switch[0]
X(1)		Toggle Switch[1]
X(2)		Toggle Switch[2]
X(3)		Toggle Switch[3]
Y(0)		Toggle Switch[4]
Y(1)		Toggle Switch[5]
Y(2)		Toggle Switch[6]
Y(3)		Toggle Switch[7]
Cin		Toggle Switch[8]
S(0)		LED Red [0]
S(1)		LED Red [1]
S(2)		LED Red [2]
S(3)		LED Red [3]
Cout		LED Red [4]

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit for the following values of Cin, X, Y

Cin	X	Y	Result
0	0	0	
0	4	5	
0	6	2	
0	10	8	
0	11	9	
1	12	10	
1	8	15	
1	5	3	
1	7	8	
1	15	15	

**Discussion:**

1. Rewrite the code of figure1 using name association.
2. Write a VHDL code for a 4-bit binary adder using only simple assignment statement (note: do not use component).



### Experiment Number 5 BCD Adder

**Apparatus:** PC and DE2 Board

**Task:** Learning the use of DE2 Pin assignment file to simplify pin assignment step.

**Theory:**

A BCD adder is a logic circuit that adds two BCD digits(A and B) and produces two digit BCD sum ( K S ). The addition of two BCD numbers, together with a possible carry from a previous stage, using a 4- bit binary adder will produce a binary result in the range (0 to 19). To obtain the results that exceeds 9, correction logic circuit is required to add (0110) to the binary results if the results is greater than (1001), and nothing is added when the binary result is less than or equal to (1001). Figure1 shows the circuit diagram of the BCD adder.

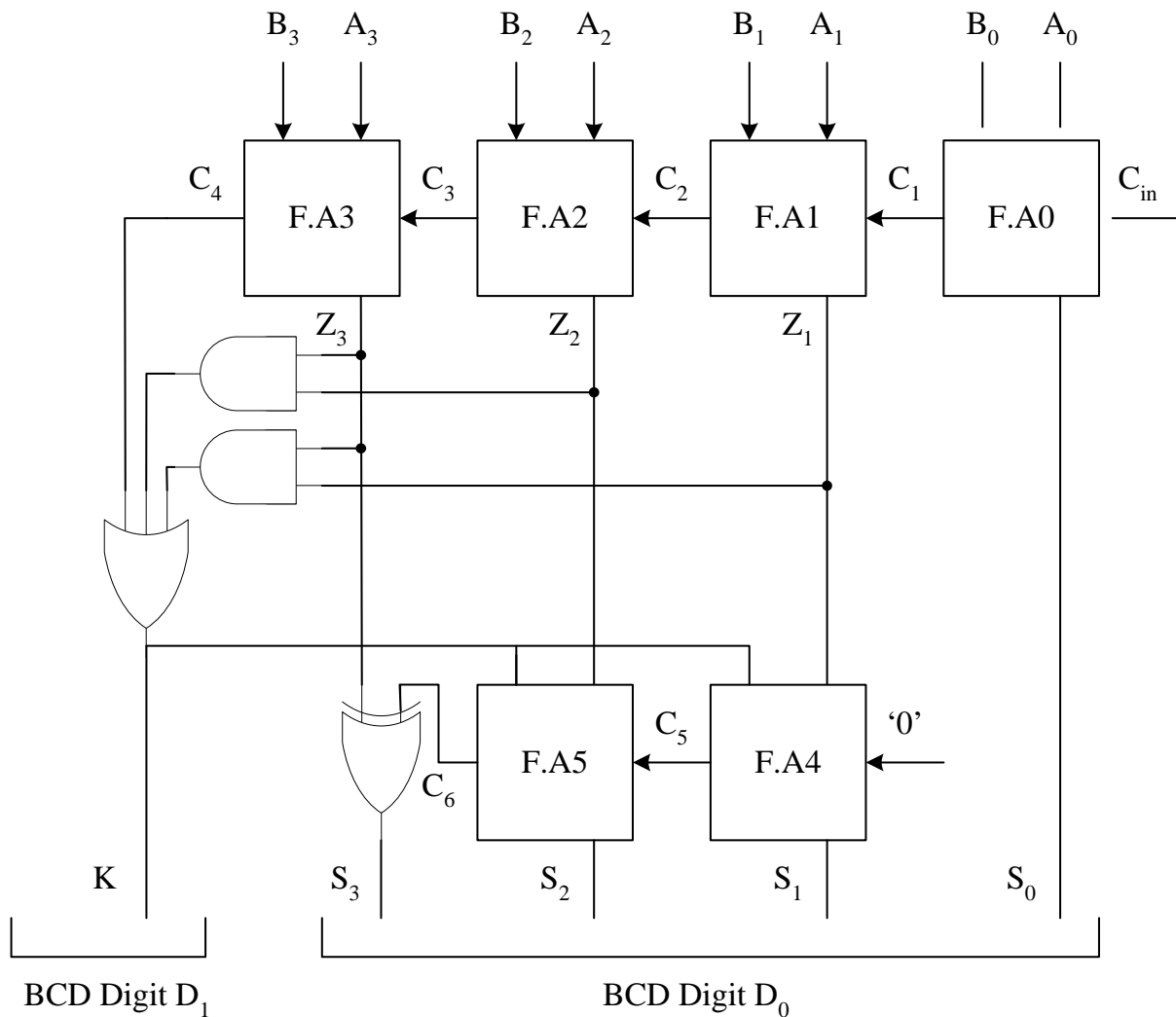


Figure 1: Circuit diagram of BCD adder.



**DE2 Pin Assignment File:**

A useful Quartus II feature allows the user to both export and import the pin assignments from a special file format, rather than creating them manually using the Assignment Editor. A simple file format that can be used for this purpose is the *comma separated value (CSV)* format, which is a common text file format that contains comma-delimited values.

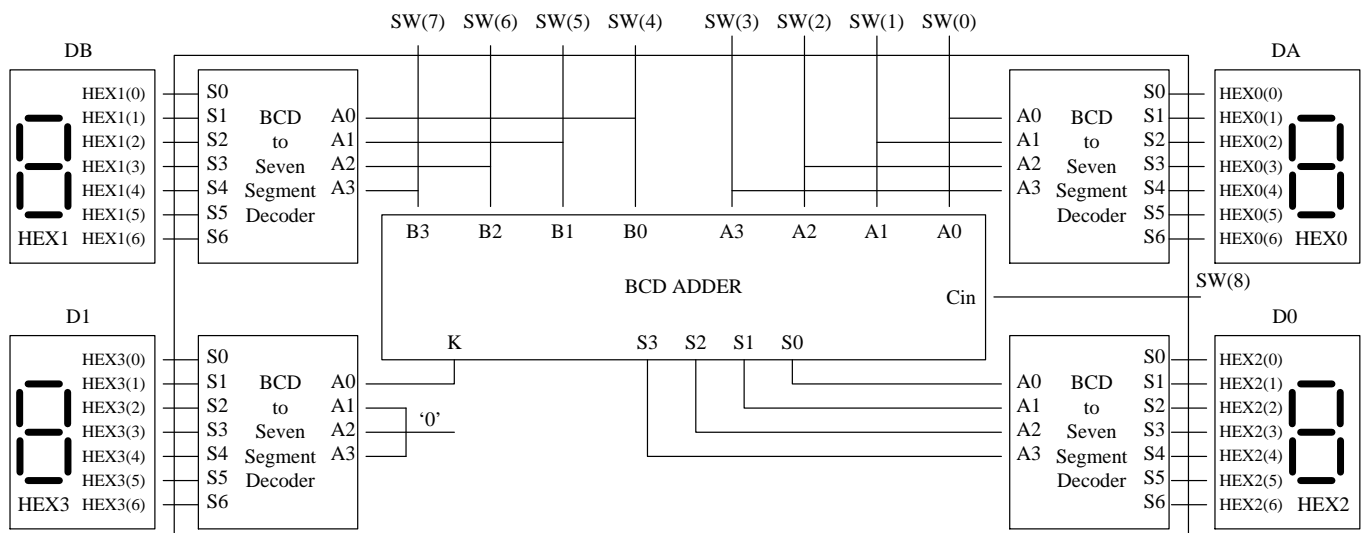
A good way to make the required pin assignments is to import into the Quartus II software the file called "**DE2\_pin\_assignments.csv**", which is provided with the DE2 board and available on your experiments folder.

It is important to realize that the pin assignments in the "**DE2\_pin\_assignments.csv**" file are useful only if the pin names given in the file are exactly the same as the port names used in your VHDL entity.

**Design Example:**

The design example for this experiment is building the BCD adder shown in figure1 with the following requirements:

1. Using full adder as a sub-circuit.
2. Using seven segment display as a sub-circuit to display the decimal value of the BCD inputs (A and B) and BCD outputs ( $D_1$  and  $D_0$ ).



**Figure 2: Block diagram for the design example.**

The VHDL code for the above requirement needs the following ports in the entity:

- 1- Input ports: 9 Toggle Switches (four switches for BCD input A, four switches for BCD input B, and one switch for Cin).
- 2- Output ports: Four Seven Segment Displays, with each seven segment has seven ports, the total ports will be (28 ports).

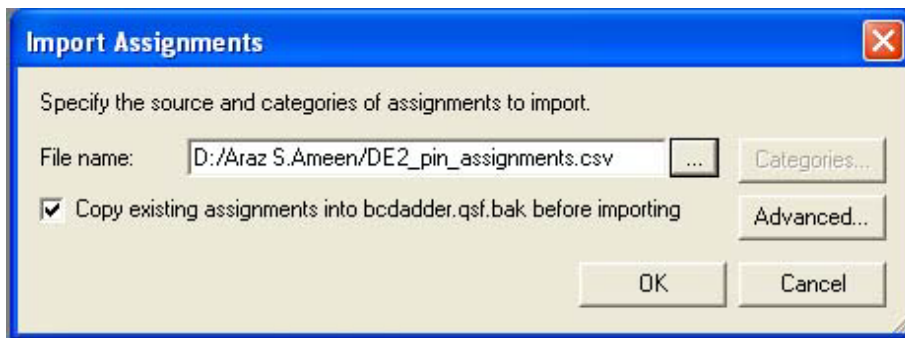
During Pin Assignments step, we must make assignments for 37 port(9 + 28) which is tedious work for doing it manually therefore we will use the "**DE2\_pin\_assignments.csv**" file.



To use the " **DE2\_pin\_assignments.csv** " file, we must use the names of the DE2 board in the VHDL code for the circuit, as shown in figure2. The VHDL code for the circuit of figure2 is shown in figure4.

**Procedure:**

1. Create a new Quartus II project for the BCD adder. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in Figure 4 and include it in your project.
3. Compile the project.
4. Import "DE2\_pin\_assignments.csv" file by choosing **Assignments > Import Assignments**. This opens the dialogue in figure 3 to select the file to import. Browse to the folder that holds your experiments and press **OK**.



**Figure3: Import Assignment Window**

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit for the following values of Cin, A, B

Cin	X	Y	Result
0	0	0	
0	4	5	
0	6	4	
0	9	8	
0	2	9	
1	0	0	
1	1	4	
1	5	3	
1	7	8	
1	9	9	

**Discussion:**

1. Describe the function of line1, and line2 of the VHDL code of figure 3.
2. Write a VHDL code for BCD adder of figure 1 using Conditional Assignment Statement.



```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-----

ENTITY bcdadder IS
PORT(SW          :IN          STD_LOGIC_VECTOR(8 DOWNT0 0);
      HEX0,HEX1,HEX2,HEX3 :OUT          STD_LOGIC_VECTOR(0 TO 6));
END bcdadder;

-----

ARCHITECTURE behavior OF bcdadder IS
    SIGNAL C      :STD_LOGIC_VECTOR(1 TO 6);
    SIGNAL Z      :STD_LOGIC_VECTOR(1 TO 3);
    SIGNAL S,S1   :STD_LOGIC_VECTOR(3 DOWNT0 0);
    SIGNAL K,L    :STD_LOGIC;

    -----

    COMPONENT fulladder
        PORT(Cin,x,y      :IN          STD_LOGIC;
              s,Cout      :OUT          STD_LOGIC);
    END COMPONENT;

    -----

    COMPONENT sevensegment
        PORT(A          :IN          STD_LOGIC_VECTOR(3 DOWNT0 0);
              S          :OUT          STD_LOGIC_VECTOR(0 TO 6));
    END COMPONENT;

    -----

    BEGIN
    -----bcd adder part-----
    L<='0';                                --line1
    FA0:fulladder PORT MAP(SW(8),SW(0),SW(4),S(0),C(1));
    FA1:fulladder PORT MAP(C(1) ,SW(1),SW(5),Z(1),C(2));
    FA2:fulladder PORT MAP(C(2) ,SW(2),SW(6),Z(2),C(3));
    FA3:fulladder PORT MAP(C(3) ,SW(3),SW(7),Z(3),C(4));
    FA4:fulladder PORT MAP(L,Z(1),K,S(1),C(5));
    FA5:fulladder PORT MAP(C(5),Z(2),K ,S(2),C(6));
    S(3)<=Z(3) XOR C(6);
    K<= C(4) OR (Z(3) AND Z(2)) OR (Z(3) AND Z(1));

    -----seven segment-----
    S1<="000" & K;                            --line2
    DA:sevensegment PORT MAP(SW(3 DOWNT0 0),HEX0);
    DB:sevensegment PORT MAP(SW(7 DOWNT0 4),HEX1);
    D0:sevensegment PORT MAP(S ,HEX2);
    D1:sevensegment PORT MAP(S1,HEX3);

    -----

    END behavior;
    
```

Figure 4: VHDL code for figure 2.



**Experiment Number 6**  
**Two Digit BCD Adders**

**Apparatus:** PC and DE2 Board

**Task:** Learning: 1. Using unsigned package for addition and subtraction  
 2. Using Conditional Assignment Statements for circuit design.

**Theory:**

Two digit BCD adders is a logic circuit that consist of two BCD adders, it is used to add two 2-digit BCD numbers, **A1 A0** and **B1 B0** to produce the three digit BCD sum **S2 S1 S0**, where each BCD digit is a 4-bit number. The block diagram of two digit BCD adder is shown in figure1.

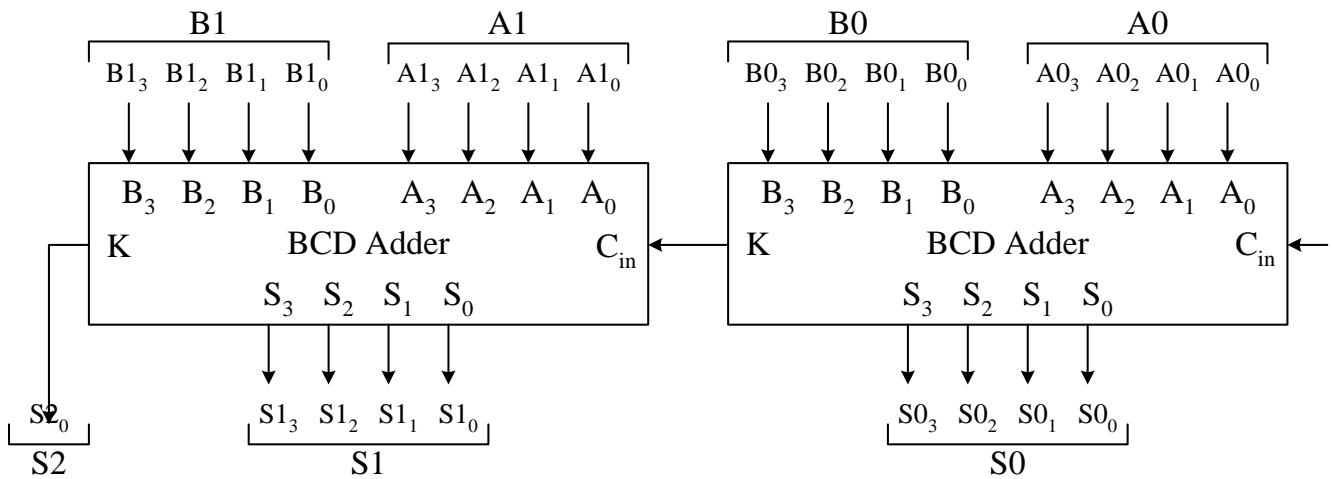


Figure 1: Block diagram of two digit BCD adder.

**Unsigned Package:**

STD\_LOGIC\_VECTOR signals can be used as binary numbers in arithmetic circuits by including in the code the following statement for unsigned arithmetic:

```
USE ieee.std_logic_unsigned.all
```

The *std\_logic\_unsigned* package specifies that it is legal to use the STD\_LOGIC\_VECTOR signal with arithmetic operators, like (+, -). The VHDL compiler should generate a circuit that works for unsigned numbers.

**Conditional Assignment Statement:**

The conditional signal assignment is used to set a signal to one of several alternative values, the general form is:

```
signal name <= expression WHEN logic expression ELSE ;
                    expression WHEN logic expression ELSE ;
                    .
                    .
                    expression ;
```

As an example, the following conditional assignment statement describes an EX-NOR gate:

```
f<='1'WHEN x1=x2 ELSE '0' ;
```



**Design Example:**

In experiment number 5, a VHDL code for a BCD adder is created by using two full adder component and simple assignment statement. The two digit BCD adder can be created using two instances of the BCD adder of experiment 5 in the way shown in figure1. A different approach for describing the two-digit BCD adder in VHDL code is to specify an algorithm like the one represented by the following pseudo-code:

1. $T0 = A0 + B0$	11. If $(T1 > 9)$ then
2. If $(T0 > 9)$ then	12. $Z1 = 10;$
3. $Z0 = 10;$	13. $C2 = 1;$
4. $C1 = 1;$	14. Else
5. Else	15. $Z1 = 0;$
6. $Z0 = 0;$	16. $C2 = 0;$
7. $C1 = 0;$	17. End if
8. End if	18. $S1 = T1 - Z1$
9. $S0 = T0 - Z0$	19. $S2 = C2$
10. $T1 = A1 + B1 + C1$	

It is reasonably straightforward to see what circuit could be used to implement this pseudo-code. Lines 1, 9, 10, and 18 represent adders, lines 2-8 and 11-17 correspond to multiplexers, and testing for the conditions  $T0 > 9$  and  $T1 > 9$  requires comparators. A VHDL code that corresponds to this pseudo-code is shown in figure2 using conditional assignment statement.

**Procedure:**

1. Create a new Quartus II project for the two digit BCD adder. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board. Use the  $SW_{3-0}$  as A0 digit,  $SW_{7-4}$  as A1 digit,  $SW_{11-8}$  as B0 digit,  $SW_{15-12}$  as B1 digit.
2. Create a VHDL entity for the code in figure 2 and include it in your project.
3. Compile the project.
4. Import "DE2\_pin\_assignments.csv" file .
5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by trying the following values for the numbers (A1 A0) and (B1 B0).

Cin	A1 A0	B1 B0	S2 S1 S0
0	00	00	
0	40	05	
0	62	47	
0	92	38	
0	99	99	
1	00	00	
1	40	05	
1	62	47	
1	92	38	
1	99	99	





```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY bcd2digit IS
    PORT(SW          :IN   STD_LOGIC_VECTOR(15 DOWNT0 0);
         HEX0, HEX1, HEX2 :OUT  STD_LOGIC_VECTOR (0 TO 6);
         HEX4,HEX5,HEX6,HEX7 :OUT  STD_LOGIC_VECTOR(0 TO 6));
END bcd2digit;
ARCHITECTURE behavior OF bcd2digit IS
    SIGNAL Z0,Z1,S2          :STD_LOGIC_VECTOR(3 DOWNT0 0);
    SIGNAL T0,T1,S0,S1      :STD_LOGIC_VECTOR(4 DOWNT0 0);
    SIGNAL c1,c2            :STD_LOGIC;
-----seven segment component-----
    COMPONENT sevensegment
        PORT(A          :IN   STD_LOGIC_VECTOR(3 DOWNT0 0);
             S          :OUT  STD_LOGIC_VECTOR(0 TO 6));
    END COMPONENT;
-----
    BEGIN
    T0<=('0' & SW(3 DOWNT0 0))+SW(11 DOWNT0 8);  --T0=A0+B0
    Z0<="1010" WHEN T0>"1001" ELSE "0000";
    c1<='1'  WHEN T0>"1001" ELSE '0';
    S0<=T0 - Z0;
    T1<=('0' & SW(7 DOWNT0 4))+SW(15 DOWNT0 12)+c1;--T1=A1+B1+C1
    Z1<="1010" WHEN T1>"1001" ELSE "0000";
    c2<='1'  WHEN T1>"1001" ELSE '0';
    S1<=T1 - Z1;
    S2<="000" & c2;
-----seven segment-----
    D0:sevensegment PORT MAP(S0(3 DOWNT0 0),HEX0);
    D1:sevensegment PORT MAP(S1(3 DOWNT0 0),HEX1);
    D2:sevensegment PORT MAP(S2(3 DOWNT0 0),HEX2);
    D4:sevensegment PORT MAP(SW(3 DOWNT0 0),HEX4);
    D5:sevensegment PORT MAP(SW(7 DOWNT0 4),HEX5);
    D6:sevensegment PORT MAP(SW(11 DOWNT0 8),HEX6);
    D7:sevensegment PORT MAP(SW(15 DOWNT0 12),HEX7);
END behavior;
    
```

Figure2: VHDL code for two digit BCD adder

**Discussion:**

1. Write a VHDL code for two digit BCD adder using the BCD adder as a component.
2. Describe briefly the task of each line in the architecture of figure 2.

## Experiment number 7 Latches and Flip Flops

**Apparatus:** PC and DE2 Board

**Task:** Learning: 1. Using Sequential Assignment Statement  
2. Simulate the design using vector waveform file.

### **Theory:**

Latches and flip-flops are storage elements consist of logic gates and feedback connection between inputs and outputs. Figure1 shows a gated D-latch circuit, a VHDL code for the D-latch can be written using simple assignment statement (as shown in figure2).

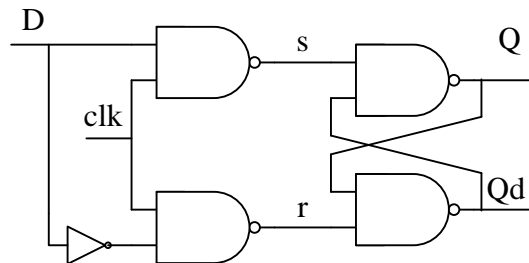


Figure1: Gated D-latch logic diagram

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
    port (D, clk      : in          std_logic ;
          Q,Qd       : buffer      std_logic );
end dlatch;
architecture behavior of dlatch is
    signal s,r       : std_logic ;
begin
    s<= D nand clk ;
    r<= (not D) nand clk ;
    Q<= s nand Qd ;
    Qd<= r nand Q ;
end behavior ;
```

Figure2: VHDL code for Gated D-latch using simple assignment statement

Altera FPGAs include flip-flops and latches that are available for implementing a user's circuit. These flip-flops and latches can be inferred using PROCESS statement.

### **PROCESS Statement:**

Since the order in which the sequential statements appear in VHDL code is significant, whereas the ordering of concurrent statement is not, the sequential statements must be separated from the concurrent statements. This is accomplished using a PROCESS statement.



The PROCESS statement appears inside an architecture body, and it encloses other statements within it. The general form of a PROCESS statement is shown in figure3.

```
General form:  PROCESS (signal_name, signal_name, ....)
                [VARIABLE declarations]
                BEGIN
                [Simple Signal Assignment Statement]
                [Variable Assignment Statement]
                [WAIT Statement]
                [CASE Statement]
                [IF Statement]
                [LOOP Statement]
                END PROCESS;
```

The VHDL code for the gated D-latch (figure2) can be rewritten using PROCESS statement as shown in figure3.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dlatch IS
    PORT( D,clk      :IN      STD_LOGIC;
          Q, Qd      :BUFFER  STD_LOGIC);
END dlatch;
ARCHITECTURE behavior OF dlatch IS
    BEGIN
        PROCESS(D,clk)
        BEGIN
            IF clk='1' THEN
                Q<=D;
            END IF;
        END PROCESS;
        Qd<= NOT Q;
    END behavior;
```

Figure3: VHDL code for Gated D latch using PROCESS statement

The VHDL code of figure3 can be converted to code for positive edged triggered D flip-flop by making some modification in the PROCESS body as follows:

```
PROCESS(clk)
    BEGIN
        IF clk'EVENT AND clk='1' THEN Q<=D;
        END IF;
    END PROCESS;
```



**Design Example:**

Figure4 shows a circuit with three different storage elements: a gated D latch, a positive-edge triggered D flip-flop, and a negative-edge triggered D flip-flop. The circuit have three inputs (D, clock, and reset), and six outputs (A, Ad, B, Bd, C, and Cd) .The VHDL code for this circuit is shown in figure5.

```

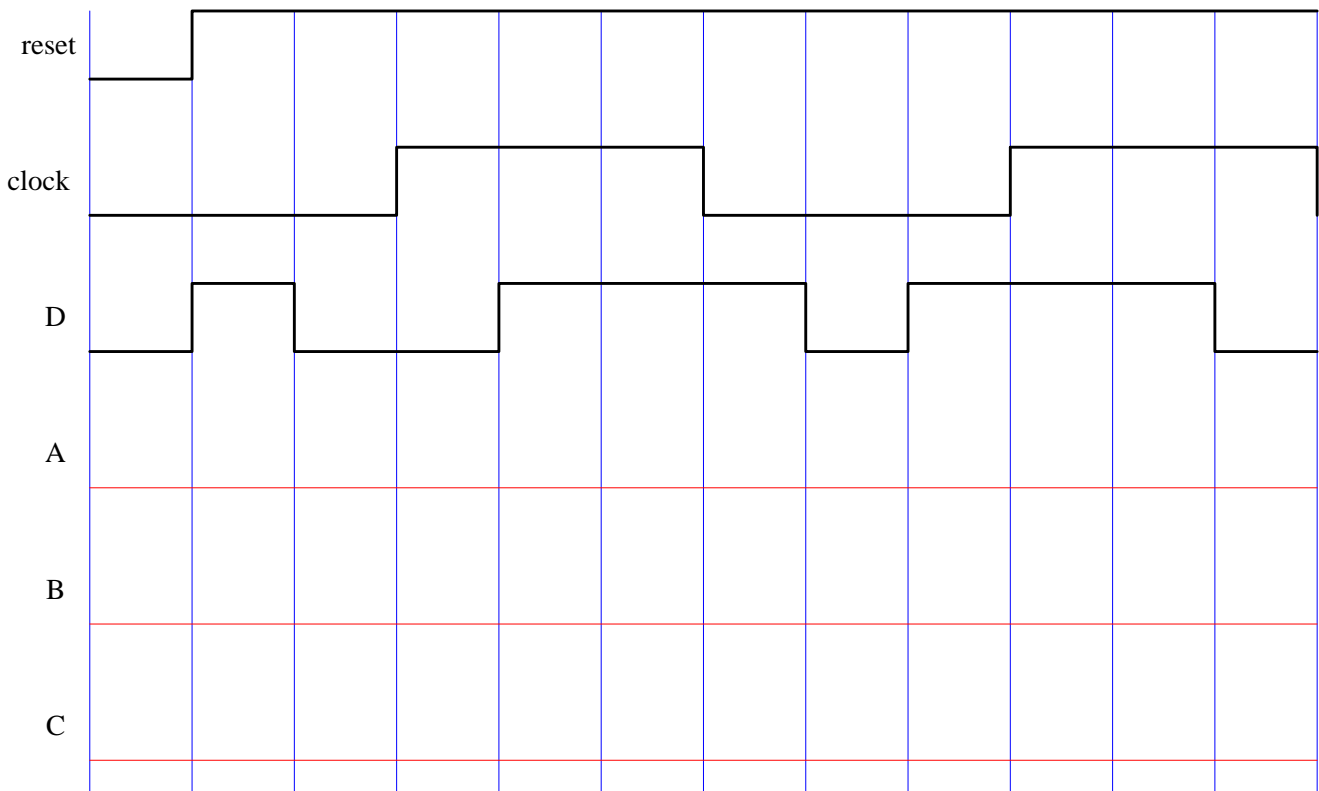
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY exp7 IS
    PORT( D,clock,reset      :IN          STD_LOGIC;
          A,Ad               :OUT          STD_LOGIC;
          B,Bd               :OUT          STD_LOGIC;
          C,Cd               :OUT          STD_LOGIC);
END exp7;
ARCHITECTURE behavior OF exp7 IS
    SIGNAL y1,y2,y3          :STD_LOGIC;
    BEGIN
        PROCESS(D,clock,reset)
            BEGIN
                IF reset='0' THEN
                    y1<='0';
                    y2<='0';
                    y3<='0';
                ELSE
                    IF clock='1' THEN y1<=D;
                    END IF;
                    IF clock'EVENT AND clock='1' THEN y2<=D;
                    END IF;
                    IF clock'EVENT AND clock='0' THEN y3<=D;
                    END IF;
                END IF;
            END PROCESS;
        A <=y1;
        Ad<=NOT y1;
        B <=y2;
        Bd<=NOT y2;
        C <=y3;
        Cd<=NOT y3;
    END behavior;
    
```

**Figure5: VHDL code for Figure4.**



**Procedure:**

1. Create a new Quartus II project for the circuit of figure4. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in figure 5 and include it in your project.
3. Compile the project.
4. Create a vector waveform file for the circuit (from file menu, select **New → Vector Waveform File**).
5. In vector waveform file window, select **Edit → End Time → 12 μsec**.
6. In vector waveform file window, select **Edit → Grid Size → 1 μsec**.
7. Insert **D, clock, A, B, C** signal into the vector waveform file and simulate the design for the following value for **D, clock** shown below( complete the timing diagram from simulation result)



**Discussion:**

1. Write a VHDL code for JK latch.
2. Write a VHDL code for JK flip-flop.



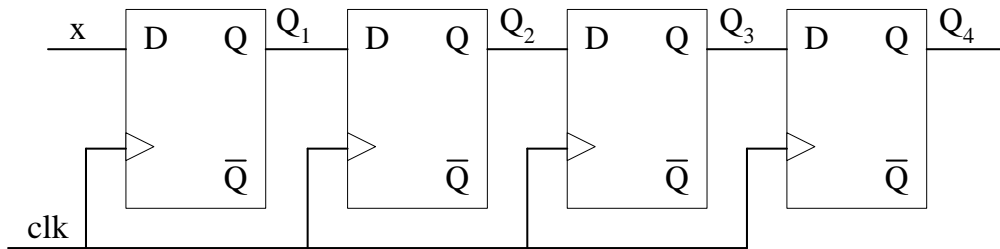
**Experiment number 8**  
**Shift Register**

**Apparatus:** PC and DE2 Board

**Task:** Using Sequential Assignment Statement to describe Shift Register.

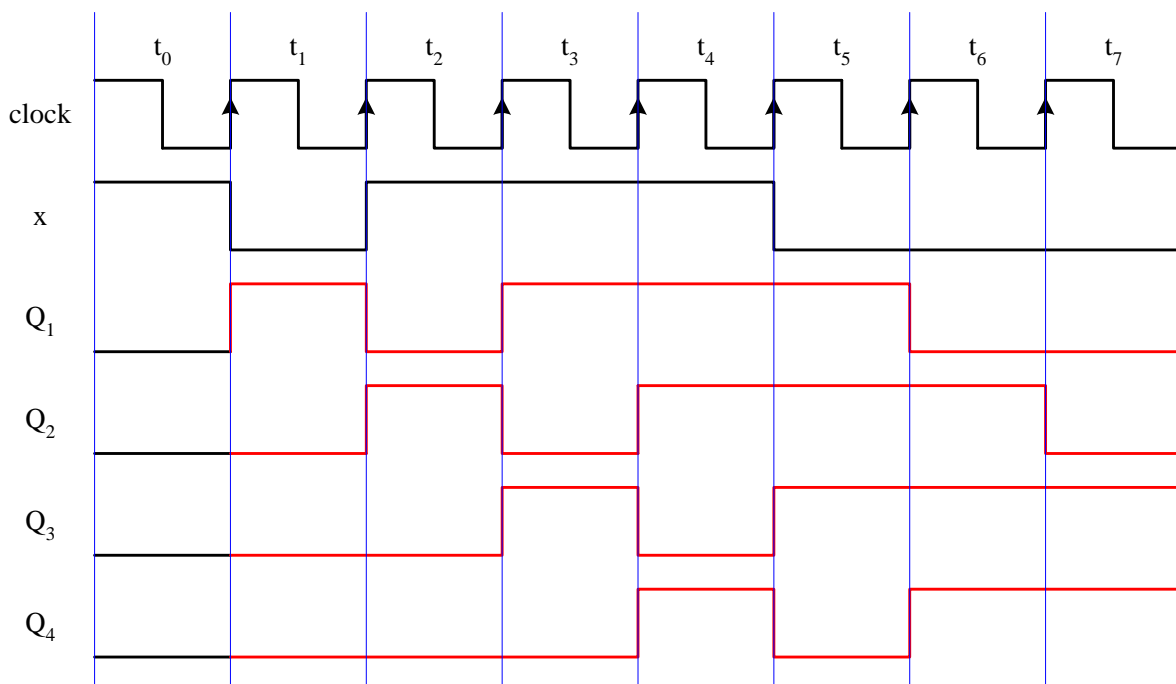
**Theory:**

A flip-flop stores one bit of information. When a set of  $n$  flip-flops is used to store  $n$  bits of information, such as  $n$ -bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register. A register that provides the ability to shift its contents is called *shift register*. Figure1 shows a four bit shift register that is used to shift its contents one bit position to the right.



**Figure1: Simple 4-bit shifts register**

The data bits are loaded into the shift register in a serial fashion using the ( $x$ ) input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. The timing diagram of the transfer is given in figure2, which shows what happens when the signal values at ( $x$ ) during eight consecutive clock cycles are (1, 0, 1, 1, 1, 0, 0, 0), assuming that the initial state of all flip-flops is 0.



**Figure2: Timing diagram for figure1.**



Figure3 shows a VHDL code that defines the 4-bit shift register of figure1. The initial state of the flip-flops is set to 0 by using a reset input. The shift register has a serial input, x, and parallel output, Q. The left-most bit in the register is Q (1), and the right-most bit is Q (4); shifting is performed in the left-to-right direction.

```

library ieee;
use ieee.std_logic_1164.all;
entity shift4 is
    port( x, clk, rst           :in           std_logic;
          Q                     :buffer      std_logic_vector(1 to 4));
end shift4;
architecture behavior of shift4 is
    begin
    process(clk,rst)
        begin
        if rst='0' then Q<="0000";
        elsif clk'event and clk='1' then
            Q(1)<=x;
            Q(2)<=Q(1);
            Q(3)<=Q(2);
            Q(4)<=Q(3);
        end if;
    end process;
end behavior;
    
```

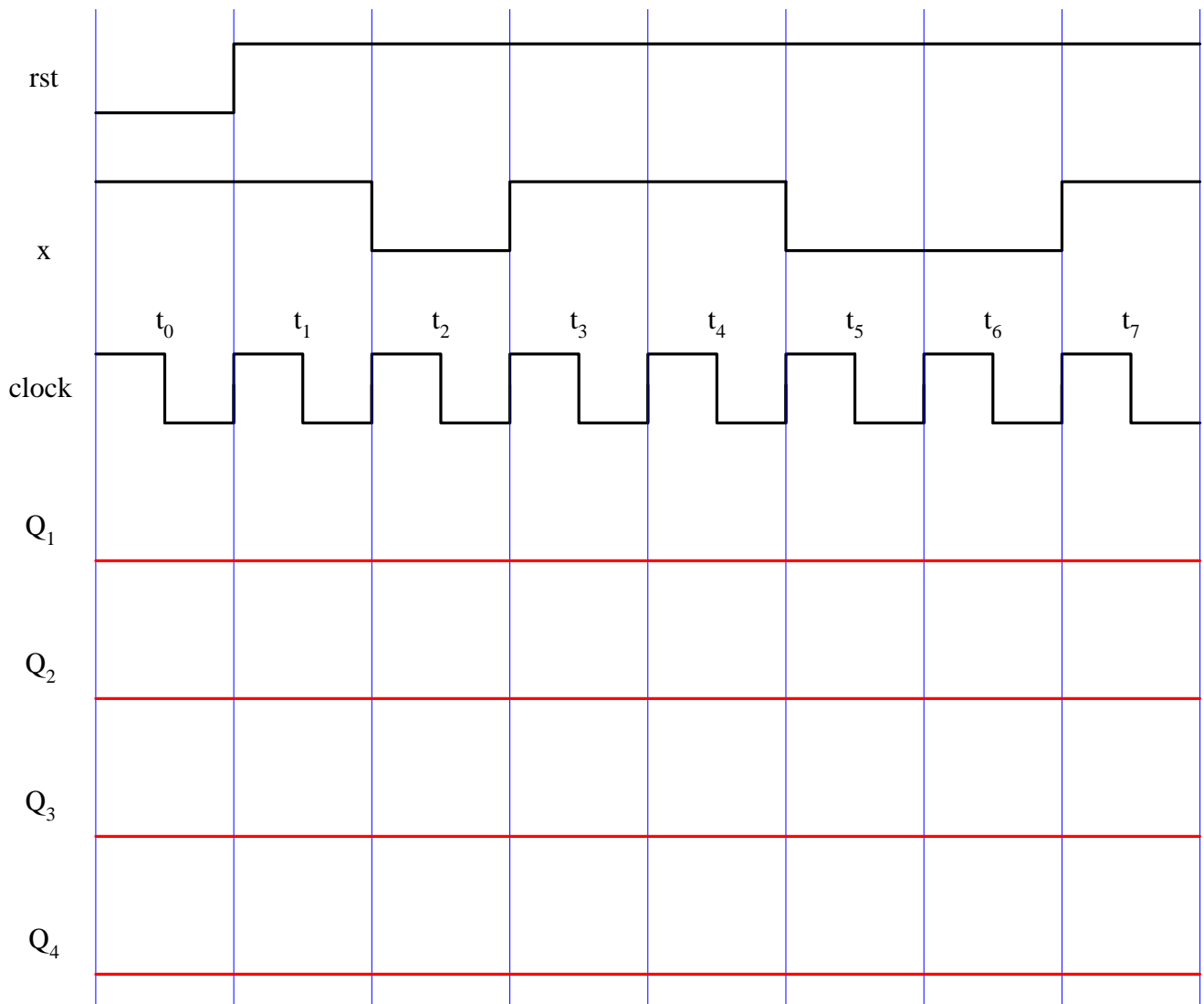
**Figure3: VHDL code for 4-bit shift register.**

**Procedure:**

1. Create a new Quartus II project for the 4-bit shift register. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in figure 3 and include it in your project.
3. Compile the project.
4. Perform the following pin assignment.

Port Name	FPGA Pin No.	Description on DE2 Board
x		Toggle Switch[0]
rst		Toggle Switch[1]
clk		Push button[0]
Q(1)		LED Red[1]
Q(2)		LED Red[2]
Q(3)		LED Red[3]
Q(4)		LED Red[4]

5. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by completing the timing diagram shown below. Note that pressing and releasing the push button switch represent one clock cycle.



**Discussion:**

1. Show by block diagrams the types of registers.
2. Write a VHDL code for each type of register.





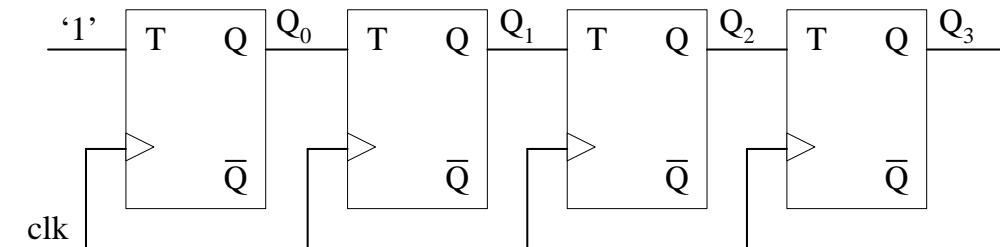
**Experiment number 9**  
**Synchronous Counters**

**Apparatus:** PC and DE2 Board

**Task:** Using Sequential Assignment Statement and arithmetic package to describe Counters.

**Theory:**

The term synchronous refers to events that have a fixed time relationship with each other. A synchronous counter is one in which all the flip-flops in the counter are clocked at the same time by a common clock pulse. Figure1 shows a 4-bit synchronous binary counter using T-type flip-flop. The LSB T flip-flop must operate in toggle state.



**Figure1: 4-bit synchronous binary counter using T flip-flop.**

A VHDL code for figure1 can be written using four instances of T flip-flop and simple assignment statement. Figure2 shows a VHDL code for the 4-bit synchronous binary counter using process statement and unsigned arithmetic package.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity count4 is
    port( clk, rst      :in      std_logic;
          Q             :out    std_logic_vector(3 DOWNTO ) );
end count4;
architecture behavior of count4 is
    signal count      : std_logic_vector(3 DOWNTO );
    begin
    process(rst,clk)
        begin
            if rst='0' then count<="0000";
            elsif clk'event and clk='1' then count<= count+1;
            end if;
        end process;
        Q<=count;
    END behavior;
    
```

**Figure2: VHDL code of 4-bit counter using process.**



**Procedure:**

1. Create a new Quartus II project for the 4-bit counter. Select *Cyclone II EP2C35F672C6* as the target chip, which is the FPGA chip on the Altera DE2 board.
2. Create a VHDL entity for the code in figure 2 and include it in your project.
3. Compile the project.
4. Perform the following pin assignment.

Port Name	FPGA Pin No.	Description on DE2 Board
rst		Toggle Switch[0]
clk		Push button[0]
Q(0)		LED Red[0]
Q(1)		LED Red[1]
Q(2)		LED Red[2]
Q(3)		LED Red[3]

5. Download the compiled circuit into the FPGA chip and test the functionality of the circuit. Note that pressing and releasing the push button switch represent one clock cycle.

**Discussion:**

1. Write a VHDL code for the circuit of figure1.
2. Modify the code of figure2 to operate in down mode