Workshop 5: Design and Implementation of a Full Adder

Objective: To design, implement and verify a full adder as an example of a combinational logic circuit

Outcome: After successful completion of this session, the student will be able to

1. Design combinational logic circuits using truth tables
2. Implement combinational logic circuits using digital ICs

Equipment Required:

1. A personal computer.
2. Intel Quartus Prime Lite Edition Design Software Version 20.1.1
3. ModelSim Intel FPGA Edition Version 20.1.1
4. Intel Cyclone IV Device Support file
5. DC power supply
6. Digital multimeter
7. Logic probe
8. Breadboard and wires

Components Required:

1. Logic gate ICs - 7486(XOR), 7408(AND), 7432(OR), 7483(Full adder) (1 No. each)
2. Resistors - 330 Ω (4 Nos.)
3. LEDs - Yellow (4 Nos.), Red (4 Nos.)

5.1 Introduction

This experiment focuses on designing and implementing combinational logic circuits. We consider a full adder as an example of a combinational logic circuit. The full adder will be designed using hierarchical design approach, where basic building blocks (half adders in this case) are used to design a complex circuit. The initial design done using the truth tables will be simulated using *Quartus Prime* and *ModelSim* software tools. Later, the designed full adder will be implemented using digital ICs. Finally, the functionality of a commercially available full adder IC will be observed.

A single-bit half adder simply performs the addition of two bits and outputs the sum and the carry-out. Examples: $1 + 0 \implies \text{sum} = 1, \text{carry-out} = 0$; $1 + 1 \implies \text{sum} = 0, \text{carry-out} = 1$. However, we cannot extend half adders to add multi-bit numbers. Hence, we construct a full adder, which has an additional input named carry-in, using 2 half adders. Unlike half adders, full adder blocks can be repeatedly connected together to achieve addition of multi-bit numbers.
Example:

$$\begin{array}{cccc} & & \text{carry-in=1} & \\ & 1 & 0 & 1 \\ + & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 \\ \hline\hline & \text{carry-out=1} & & \end{array} \tag{5.1}$$

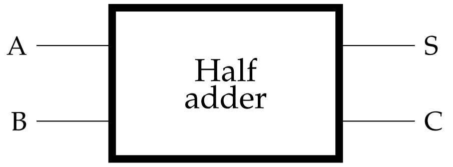


Figure 5.1: Block diagram of a half adder. A and B are input bits, S is the sum and C is the carry-out.

5.2 Pre-Lab

5.2.1 Truth Tables

Task 1. *Complete the truth table of the half adder (Table 5.1). Refer Figure 5.1.*

First bit (A)	Second bit (B)	Sum (S)	Carry-out (C)
0	0		
0	1		
1	0		
1	1		

Table 5.1: Truth table of the half adder

Task 2. *Derive and simplify boolean expressions for the outputs, S and C, in terms of the input bits, A and B.*

Task 3. *Complete the truth table of the full adder (Table 5.2).*

First bit (A)	Second bit (B)	Carry-in (C_{in})	Sum (S)	Carry-out (C_{out})
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Table 5.2: Truth table of the full adder

Task 4. *Derive boolean expressions for the outputs, S and C_{out}, in terms of the inputs A, B and C_{in}. Factorize the expressions so that the full adder can be implemented using 2 half adder blocks. Hint: Refer Task 2.*

Task 5. *Draw the block diagram of a full adder constructed using 2 half adders.*

5.2.2 Simulation Using Quartus and ModelSim

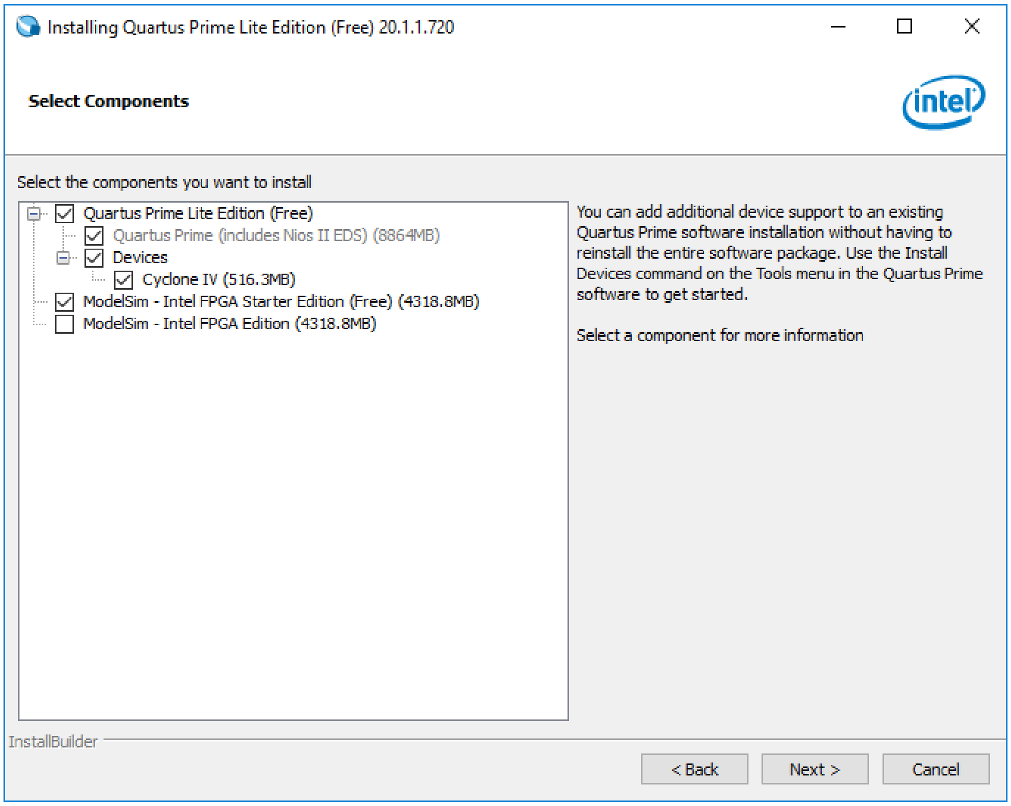
This section provides a step-by-step guide for installing Quartus and ModelSim, and simulating the full adder designed in the previous section.
Note: Required setup files have to be downloaded before starting the section.

Setting-up the Environment

Download setup files corresponding to

- Intel Quartus Prime Lite Edition Design Software Version 20.1.1
- ModelSim Intel FPGA Edition Version 20.1.1
- Intel Cyclone IV Device Support file

into a single directory and run the Quartus installer. Proceed until the installer prompts for selecting components to install. Select all the components in the list except ModelSim paid version (see Figure 5.2).



Installing Quartus Prime Lite Edition (Free) 20.1.1.720

Select Components

Select the components you want to install

- ☑ Quartus Prime Lite Edition (Free)
  - ☑ Quartus Prime (includes Nios II EDS) (8864MB)
  - ☑ Devices
    - ☑ Cyclone IV (516.3MB)
- ☑ ModelSim - Intel FPGA Starter Edition (Free) (4318.8MB)
- □ ModelSim - Intel FPGA Edition (4318.8MB)

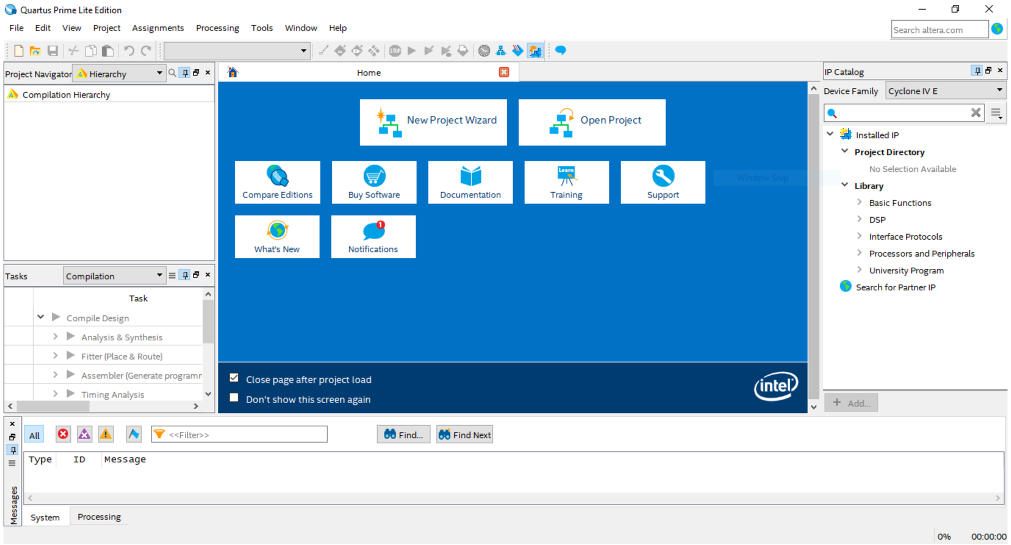
You can add additional device support to an existing Quartus Prime software installation without having to reinstall the entire software package. Use the Install Devices command on the Tools menu in the Quartus Prime software to get started.

Select a component for more information

Figure 5.2: Quartus installer: selecting components to install

After setting-up the required software tools, open *Quartus Prime Lite 20.1.1* Design Software. It will bring you to the home screen of Quartus illustrated in Figure 5.3. To verify that the ModelSim has been integrated correctly, complete the steps mentioned below;

1. Click **Tools > Options** from the menu bar. This will open up the **Options** dialog box.
2. Select **EDA Tool Options** listed under **General** category in the left-hand side pane.
3. There should be file paths mentioned in-front of **ModelSim** and **ModelSim-Altera** (see Figure 5.4).
4. From your file manager/explorer, navigate to the mentioned location and check whether **modelsim.exe** file exists. If so, you are good to go. Press **Cancel** button to exit without modifying anything.
5. If not, find the location of the **modelsim.exe** file and replace the current paths by the correct path. Press **OK** to save and exit.

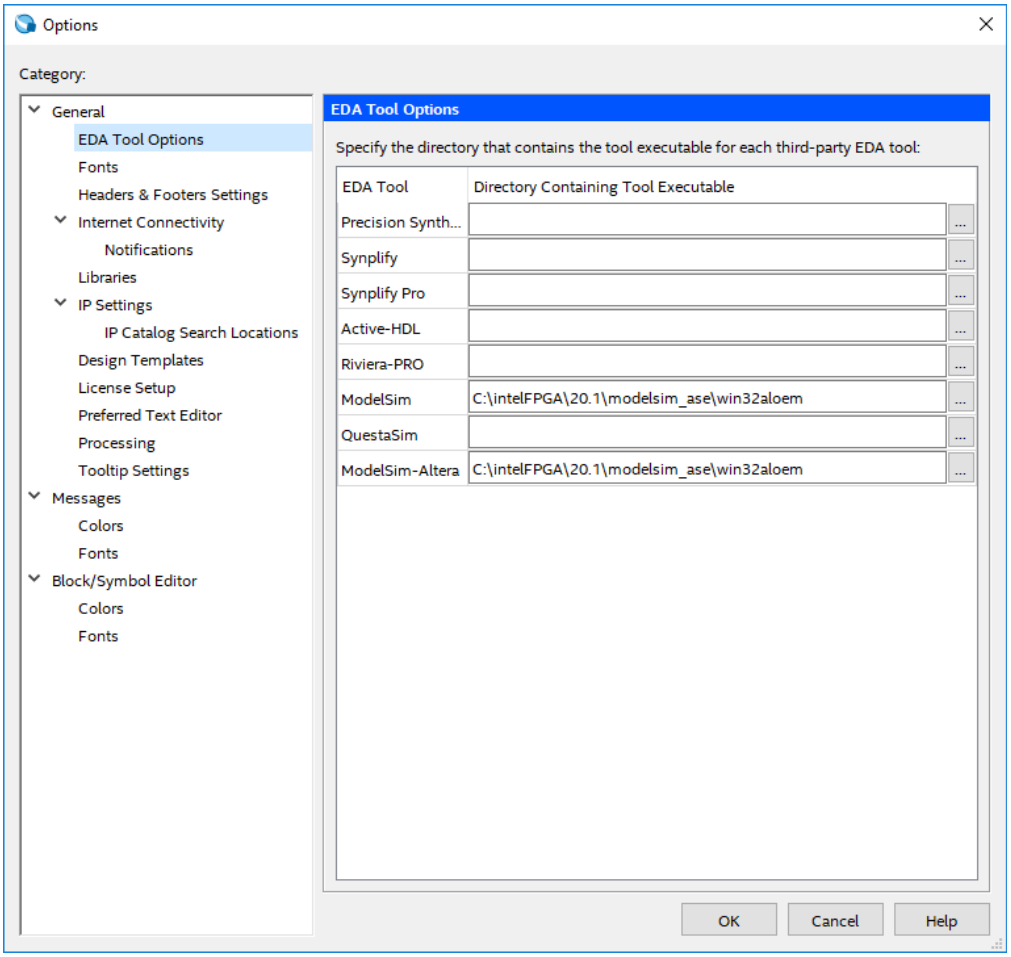


Device Family: Cyclone IV E

☑ Close page after project load

□ Don't show this screen again

Figure 5.3: Quartus home screen



EDA Tool Options

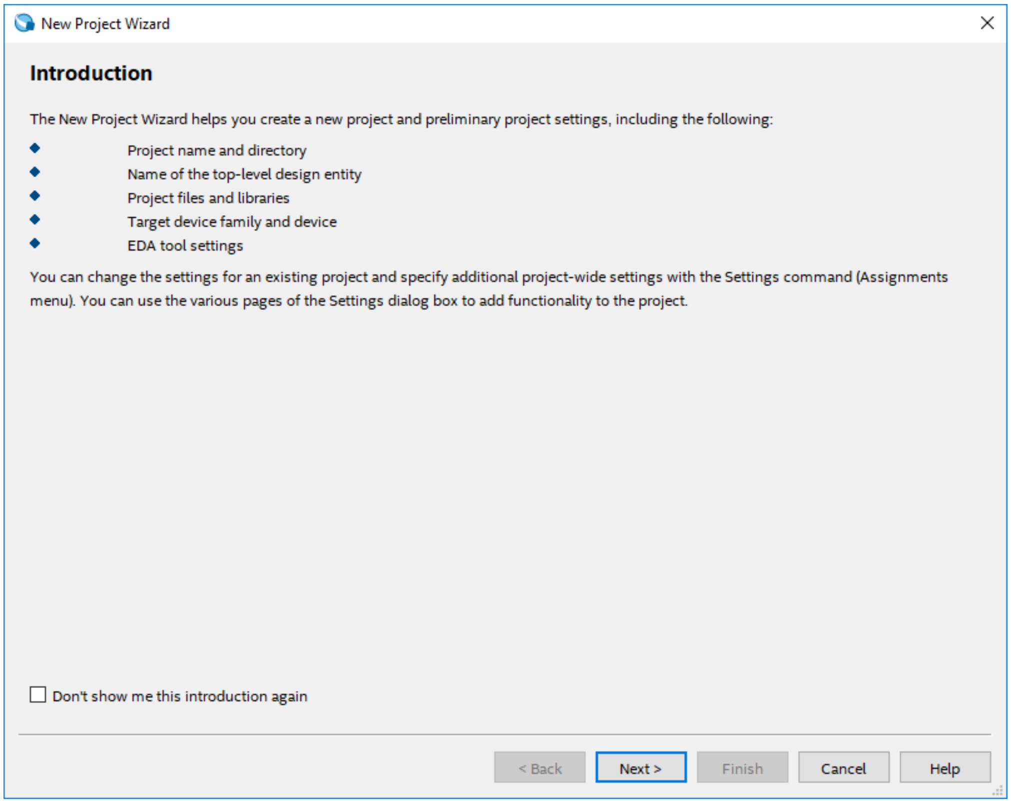
Specify the directory that contains the tool executable for each third-party EDA tool:

EDA Tool	Directory Containing Tool Executable
Precision Synth...	
Synplify	
Synplify Pro	
Active-HDL	
Riviera-PRO	
ModelSim	C:\intelFPGA\20.1\modelsim_ase\win32aloem
QuestaSim	
ModelSim-Altera	C:\intelFPGA\20.1\modelsim_ase\win32aloem

Figure 5.4: Setting ModelSim path

Creating a New Project

To start the **New Project Wizard**, either click on the "New Project Wizard" button on the home screen or goto **File > New Project Wizard**. This will open up the window illustrated in Figure 5.5.



New Project Wizard

Introduction

The New Project Wizard helps you create a new project and preliminary project settings, including the following:

- Project name and directory
- Name of the top-level design entity
- Project files and libraries
- Target device family and device
- EDA tool settings

You can change the settings for an existing project and specify additional project-wide settings with the Settings command (Assignments menu). You can use the various pages of the Settings dialog box to add functionality to the project.

□ Don't show me this introduction again

Figure 5.5: New Project Wizard

Follow the instructions listed below to create a new project.

1. Click **Next** to navigate to the project details form.

2. Specify a working directory and a suitable name for the project and click **Next**.

3. Specify the project type. Since we are creating this project from scratch, select **Empty project**. Click **Next**.

4. If we need to use any already-existing files, we can specify them here. Since there are no such files, click **Next** to continue.

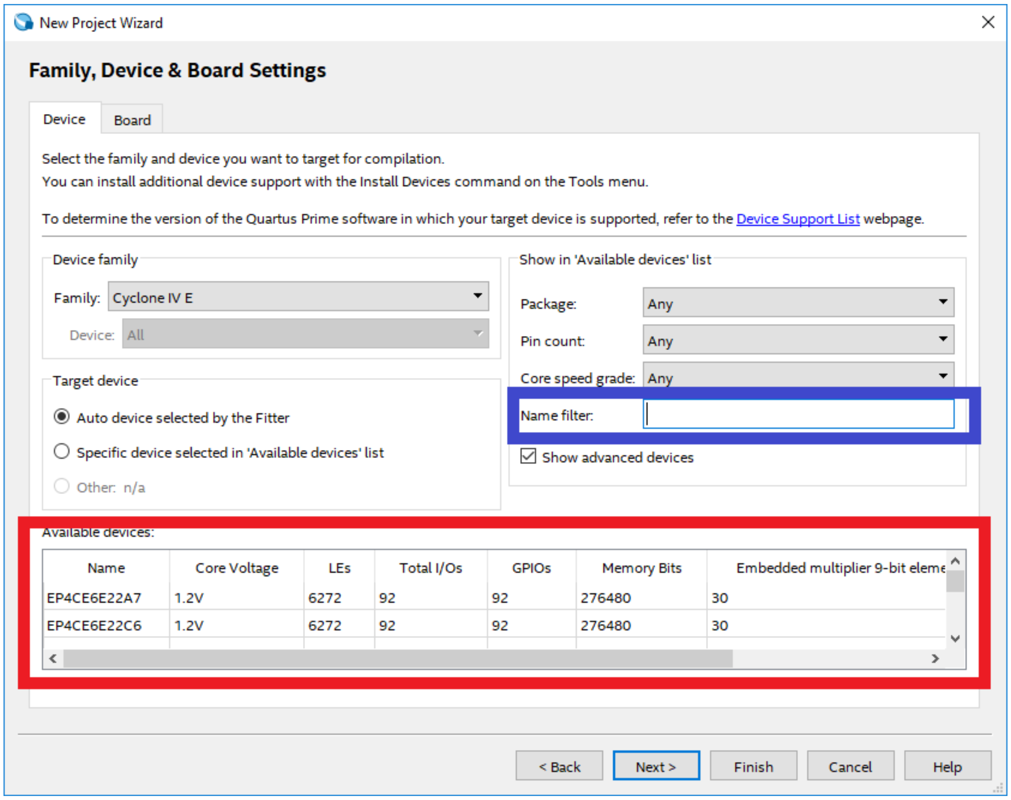
5. Carefully select the device family and model. In order to use *DE0-Nano* development board (which will be used in a later experiment), select the following;

   - Family: Cyclone IV E
   - Device: EP4CE22F17C6N

   See Figure 5.6. Click **Next** to continue.

6. If we need to use any additional third part software tools along with Quartus, we can specify them here. Since we are not going to use any, click **Next** to move on.

7. Finally, a summary of all the attributes of the new project will be shown. If everything is correct, click **Finish**. This will create the new project and will bring you to the screen shown in Figure 5.7.



New Project Wizard

Family, Device & Board Settings

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

To determine the version of the Quartus Prime software in which your target device is supported, refer to the Device Support List webpage.

Device family

Family: Cyclone IV E

Device: All

Target device

● Auto device selected by the Fitter

○ Specific device selected in 'Available devices' list

○ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Core speed grade: Any

Name filter:

☑ Show advanced devices

Available devices:

Name	Core Voltage	LEs	Total I/Os	GPIOs	Memory Bits	Embedded multiplier 9-bit eleme
EP4CE6E22A7	1.2V	6272	92	92	276480	30
EP4CE6E22C6	1.2V	6272	92	92	276480	30

Figure 5.6: Family, device and board selection. Red square: select device type here. Blue square: type a part of the device name to filter the list

Constructing the Half Adder as a Schematic Design

As the first step of creating the schematic, we need to add a new file under the new project. We will construct our half adder module as a schematic design, where we can draw the block diagram of the module using basic logic gates. To add a schematic design file, follow the steps mentioned below;

1. Click **File > New**. This will open up the **New** window.

2. Select **Block Diagram/Schematic file** under **Design Files** category and click **OK**.

This will open the **Graphic Editor** window, where we can draw our schematic. In order to place logic components on the canvas, click on icon from the tool bar. It will open the **Symbol** window. Inside the **Libraries** pane, expand into **primitives > logic** and select a logic gate to place it on the canvas. While the **Repeat-insert mode** is checked, you can repeatedly click on the canvas to place several instances of the selected logic components.

Similarly, to place input/output ports of the module, select icon from the tool bar. To draw wires connecting the components, select icon. Any component or wire can be removed by first selecting the component/wire and then pressing **Delete** on keyboard. To change the properties of a component (name, default value etc.) double click on the component itself. This will open up a properties window.

Task 6. *Construct the half adder as a schematic design. Refer Task 2. Rename all the ports correctly (A, B, S and C) and save the file in the current working directory (make sure to select* ***Add file to current project*** *in the* ***Save As*** *dialog box).*

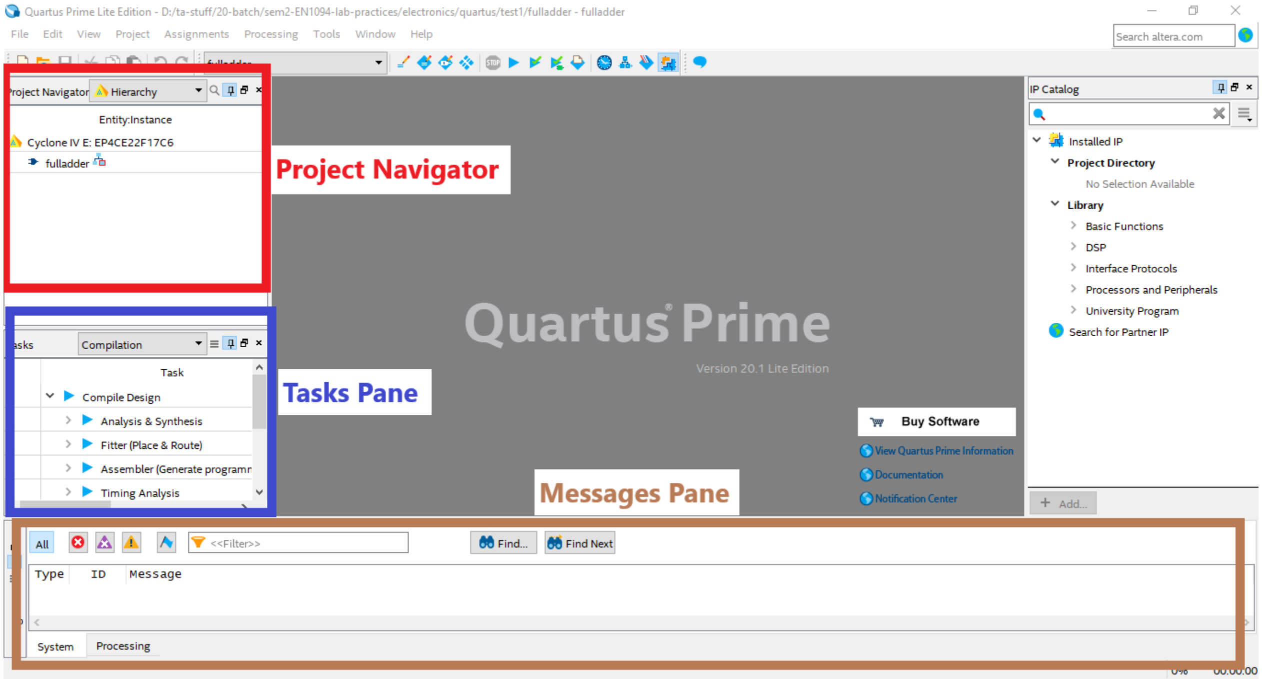


Figure 5.7: Project display. *Project Navigator* displays different files/modules of the project organized in different forms. *Tasks Pane* displays different tasks of the project flow and their status. *Messages Pane* displays messages corresponding to different tools and tasks.

Simulating the Half Adder

Prior to simulating or programming a design into a Field Programmable Gate Array (FPGA), the design must be compiled. In order to do so, we have to first specify the outer-most module of the design. Hence, before simulating the half adder created above, we have to first specify it as the top level entity of this project and then compile. To do so, follow the instructions given below;

1. In **Project Navigator** pane, select **Files** from the top-most drop down list. This will display all the files included in the project.

2. Right click on the half adder file and select **Set as Top-Level Entity**.

3. Compile the project by selecting **Processing > Start Compilation** from the menu bar.

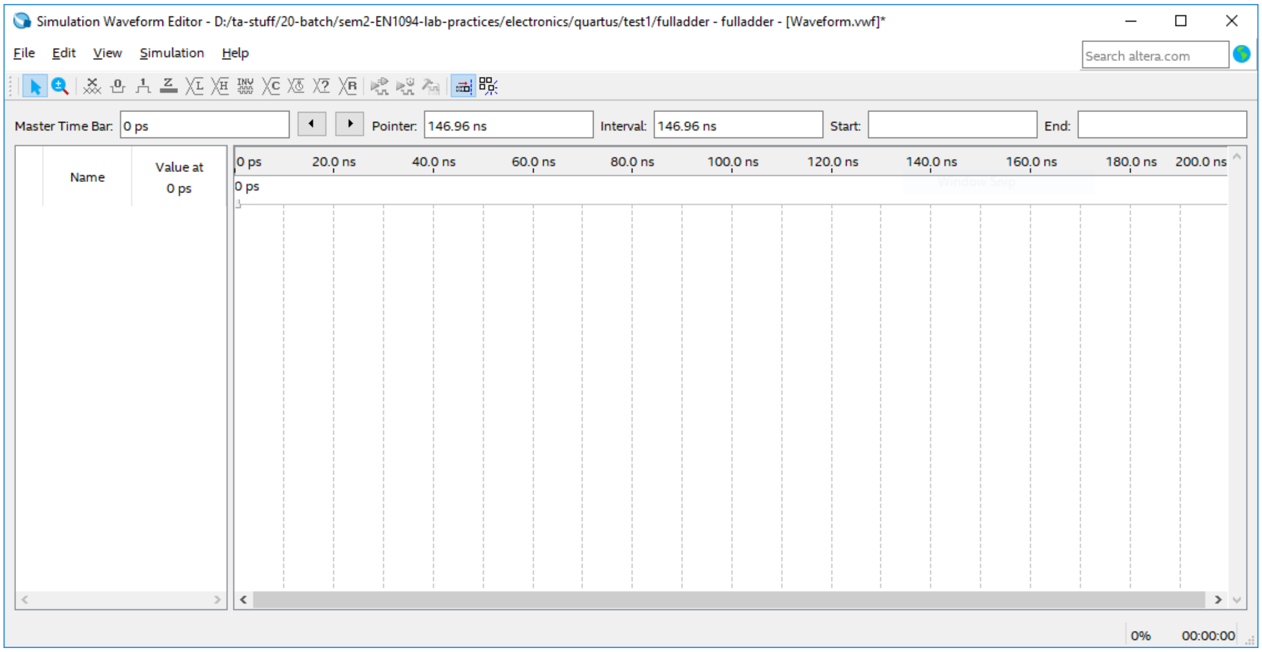
Messages related to the compilation process will appear on the **Messages** pane. A **Compilation Report** will be displayed in the end. If the last message on the **Messages** pane informs that the compilation has been successful, you can proceed to simulation. Otherwise, scroll the pane to find out the error. Double clicking on an error will indicate its origin.

Simulation Waveform Editor will be used as the simulation tool. It can provide different waveforms specified by the user to the input ports of a compiled module and display the output waveforms. It will invoke the *ModelSim* simulation tool to generate the waveforms. To simulate the half adder, follow the instructions given below;

1. Click **File > New** from menu bar and add a new **University Program VWF** file listed under the **Verification/Debugging Files**. It will open up the **Simulation Waveform Editor** window (see Figure 5.8).

2. Set the simulation duration to 80ns by clicking **Edit > Set End Time...**.

3. Click **Edit > Insert > Insert Node or Bus...**.



Simulation Waveform Editor - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/test1/fulladder - fulladder - [Waveform.vwf]*

Master Time Bar: 0 ps

Pointer: 146.96 ns

Interval: 146.96 ns

Start:

End:

Figure 5.8: Simulation Waveform Editor

4. From the opened dialog box, click on **Node Finder...** button. It will open up the **Node Finder** dialog box.

5. In the **Node Finder**, set the **Filter:** to **Pins: all** from the drop down list and press **List**. A list of all the input/output nodes will be shown on the **Nodes Found:** pane (on the left hand side).

6. Add all the input/output nodes of the half adder to the **Selected Nodes:** pane (on the right hand side) by clicking on the **>** or **>>** buttons in-between the two panes.

7. Once all the nodes are included in the **Selected Nodes:** pane, click **OK**. This will insert a waveform for each node into the waveform editor.

Since we have made some changes, we can now save our waveform file by clicking **File > Save**. As the next step, we need to specify a waveform for each input node. To do this, follow the instructions given below;

1. On the waveform editor, click on an empty area within the time slot that you need to specify the value for. While pressing the mouse button, move the mouse so that the entire region where you need to specify the value is covered by the selection square (blue colour). See Figure 5.9.

2. After selecting the required part of the wave, press 0 or 1 button from the toolbar to make the selected part logic 0 or 1 respectively.

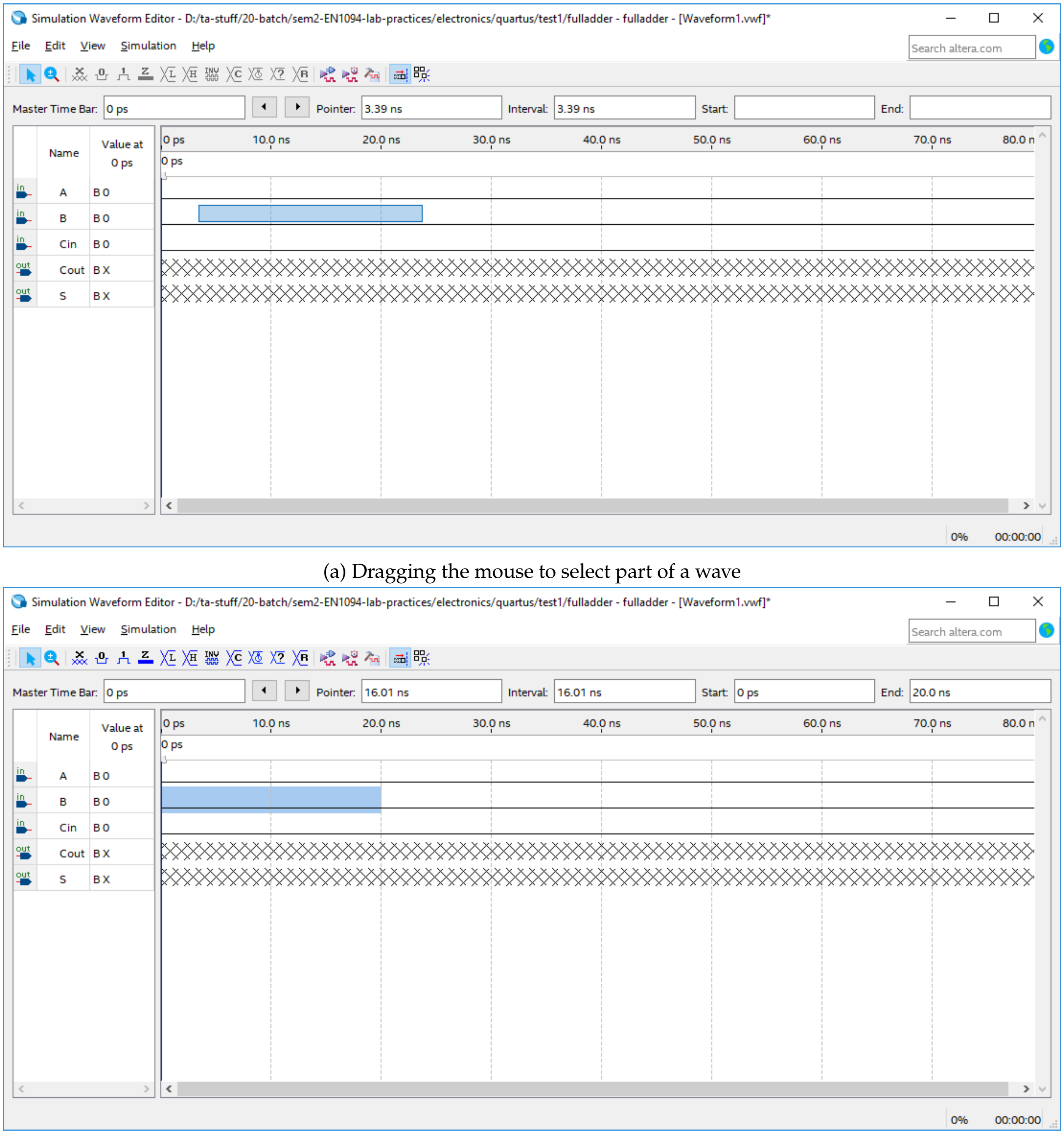
After all the input waveforms have been specified correctly, you can now run the simulation by following the instructions listed below;

1. Click **Simulation > Simulation Settings** on the menu bar. This will open up a settings window which includes scripts for running the simulation.

2. If not already selected, select **Verilog** as the **HDL Language**.

3. On the **Functional Simulation Settings** tab, locate the line **“vsim -novopt -c ....."** inside the **ModelSim Script (Functional Simulation)** pane. See Figure 5.10.

4. Carefully delete the **-novopt** command from the line. Keep everything else unmodified. If you made a mistake, press **Restore Defaults** button on the bottom of the window to reset.


Simulation Waveform Editor - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/test1/fulladder - fulladder - [Waveform1.vwf]*

Master Time Bar: 0 ps Pointer: 3.39 ns Interval: 3.39 ns Start: End:

Name	Value at 0 ps
A	B 0
B	B 0
Cin	B 0
Cout	B X
S	B X

(a) Dragging the mouse to select part of a wave

Simulation Waveform Editor - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/test1/fulladder - fulladder - [Waveform1.vwf]*

Master Time Bar: 0 ps Pointer: 16.01 ns Interval: 16.01 ns Start: 0 ps End: 20.0 ns

Name	Value at 0 ps
A	B 0
B	B 0
Cin	B 0
Cout	B X
S	B X

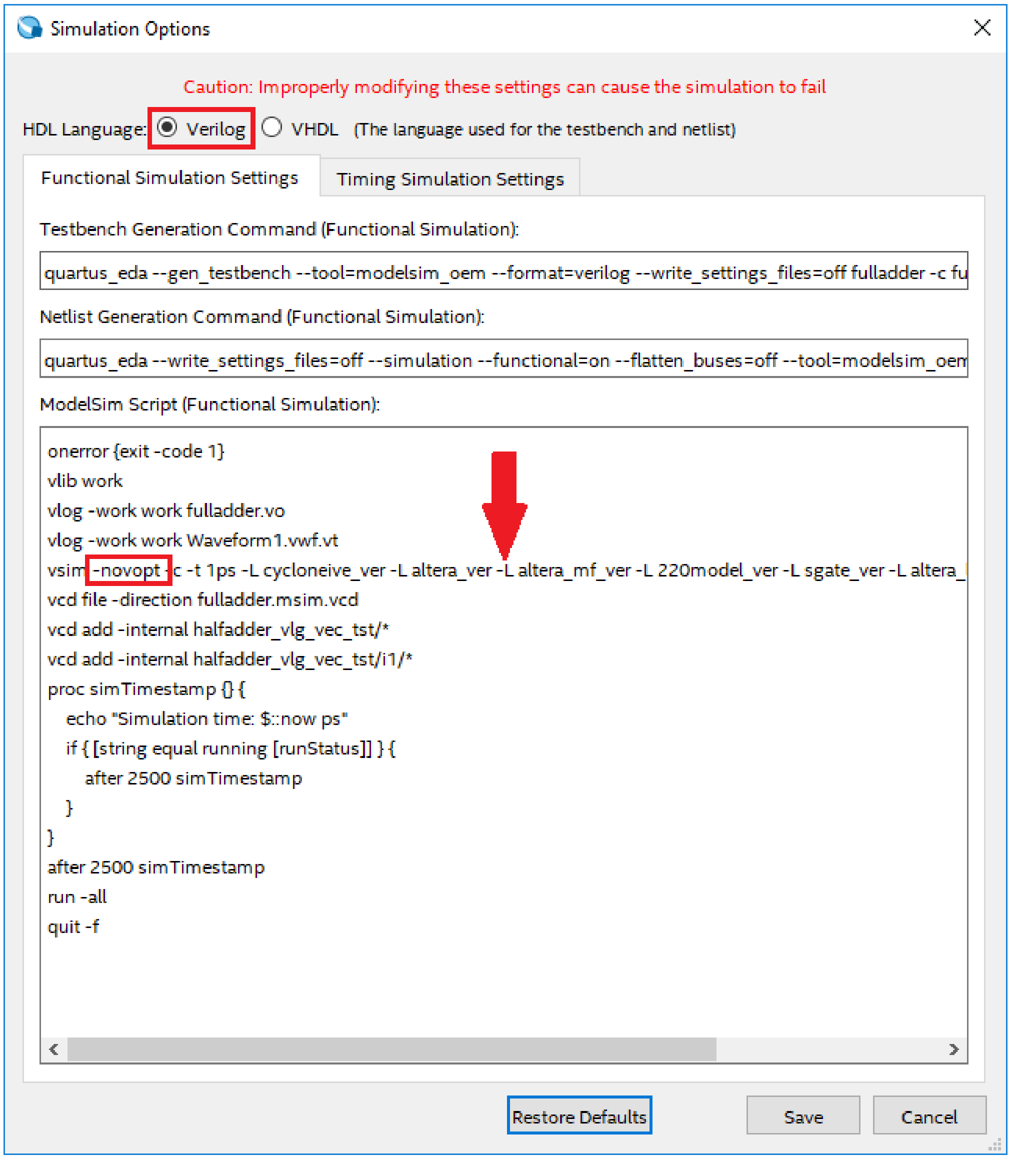
(b) Selected part of the wave after releasing the mouse button

Figure 5.9: Selecting part of a wave to specify the value

5. Press **Save** button to save settings and close the window.

6. To run the simulation, click **Simulation > Run Functional Simulation** from the menu bar. It will open up a new window displaying the status of different sub-tasks required for the simulation. In the end, if there are no errors, the output waveforms will be displayed on the **Simulation Waveform Editor**. An example is given in Figure 5.11.

Task 7. *Simulate the half adder constructed under Task 6. Verify the functionality against Table 5.1.*



Caution: Improperly modifying these settings can cause the simulation to fail

HDL Language: ● Verilog ○ VHDL (The language used for the testbench and netlist)

Testbench Generation Command (Functional Simulation):

```
quartus_eda --gen_testbench --tool=modelsim_oem --format=verilog --write_settings_files=off fulladder -c fu
```

Netlist Generation Command (Functional Simulation):

```
quartus_eda --write_settings_files=off --simulation --functional=on --flatten_buses=off --tool=modelsim_oem
```

ModelSim Script (Functional Simulation):

```
onerror {exit -code 1}
vlib work
vlog -work work fulladder.vo
vlog -work work Waveform1.vwf.vt
vsim -novopt -c -t 1ps -L cycloneive_ver -L altera_ver -L altera_mf_ver -L 220model_ver -L sgate_ver -L altera_
vcd file -direction fulladder.msim.vcd
vcd add -internal halfadder_vlg_vec_tst/*
vcd add -internal halfadder_vlg_vec_tst/i1/*
proc simTimestamp {} {
  echo "Simulation time: $::now ps"
  if { [string equal running [runStatus]] } {
    after 2500 simTimestamp
  }
}
after 2500 simTimestamp
run -all
quit -f
```

Figure 5.10: Simulation Settings

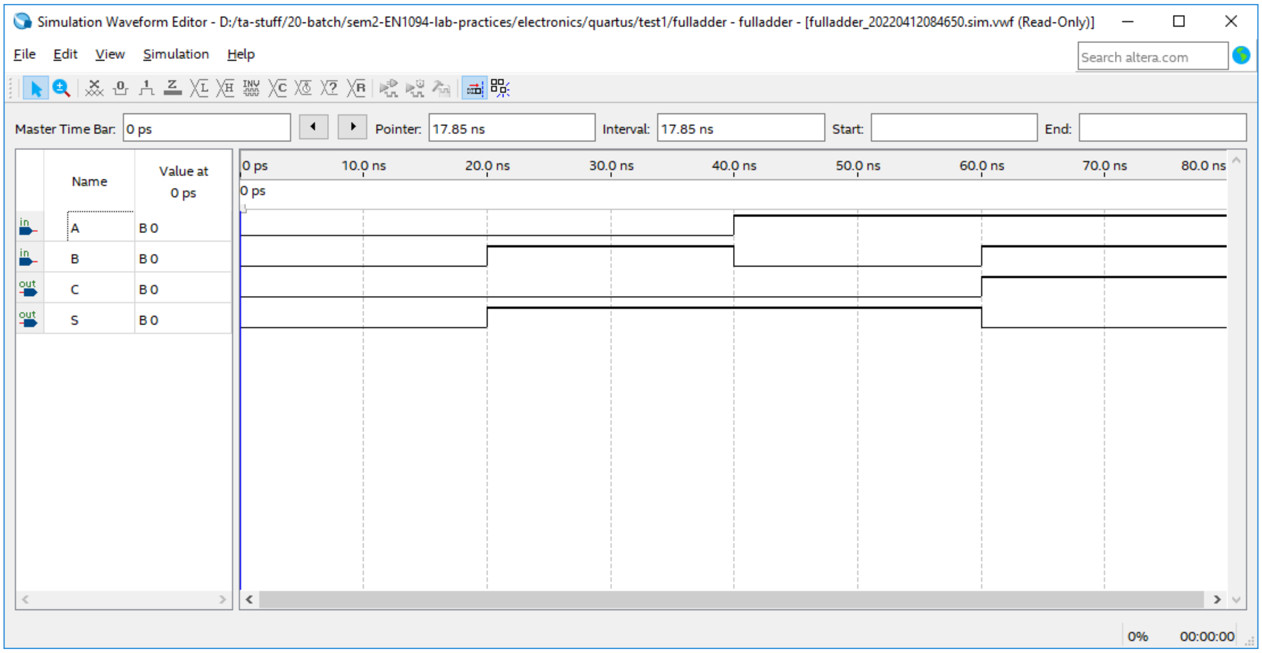
Constructing and Simulating the Full Adder

Quartus facilitates *hierarchical* designing. In other words, we can use previously designed modules as components of a new design. Consequently, we may use the already constructed half adder to create the full adder under this section. In order to use the half adder as a component of another design, we have to first create a symbol for it (note that this is only required in case of schematic designs, not for HDL designs). Follow the below steps to create a symbol for the half adder.

1. Open the half adder file by double clicking on it in the **Project Navigator**.

2. Click **File > Create / Update > Create Symbol Files for Current File**. This will open up **Create Symbol File** dialog box.

3. Give an appropriate name and save the symbol file inside the current working directory. Make sure that the file type is **Symbol File (*.bsf)**.


Simulation Waveform Editor - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/test1/fulladder - fulladder - [fulladder_20220412084650.sim.vwf (Read-Only)]

Master Time Bar: 0 ps

Pointer: 17.85 ns

Interval: 17.85 ns

Start:

End:

Name	Value at 0 ps
A	B 0
B	B 0
C	B 0
S	B 0

Figure 5.11: Simulation result

Create a **Block Diagram/Schematic File** for the new top-level design. Make sure to add the file to the project. To insert instances of the half adder, follow the steps listed below;

1. Click on icon to open the **Symbol** window.
2. Click on the button with dots, located next to the text box under the title **Name:** to browse for new symbol files. See Figure 5.12.

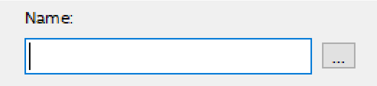


Figure 5.12: Browse button to locate new symbols

3. Locate the **.bsf** file corresponding to the half adder.
4. Now you can place it on the canvas and use it in a similar manner to an ordinary component.

Task 8. *Based on the results of Task 5, create a full adder using half adder blocks created under Task 6. Compile and simulate the design and verify the functionality with respect to Table 5.2. When compiling, make sure that the full adder file has been selected as the* ***Top-Level Entity****.*

5.3 Constructing a Full Adder Using Logic ICs

Task 9. *Identify the types and pinouts of the logic ICs using the datasheets provided.*

Task 10. *Construct two independent half adders on the breadboard using the logic ICs. Connect LEDs to the inputs (yellow) and the outputs (red) of each half adder as shown in Figure 5.13 to observe the logic levels easily. Verify the functionality with respect to Table 5.1.*

Task 11. *Connect the two half adders to construct a full adder according to the results of Task 5 and verify the functionality against Table 5.2.*

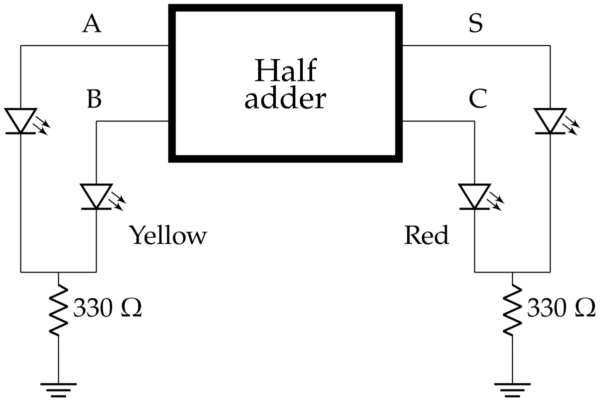


Figure 5.13: Inputs and outputs of half adder

5.4 Constructing a Full Adder Using a Full Adder IC

7483 is a commercially available full adder IC (see Figure 5.14). It can be used to add two 4-bit numbers and a carry-in bit. The result is provided as a 4-bit number, along with the carry-out. Therefore, the IC can be repeatedly connected to construct adders with a higher word length.

Task 12. *Identify the pinout of the 7483 full adder IC using the datasheet.*

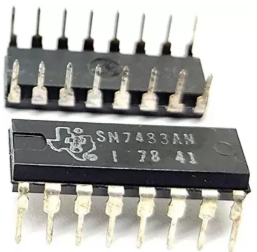


Figure 5.14: 7483 full adder IC

Task 13. *Connect LEDs along with current limiting resistors (330 Ω) to the outputs of the full adder IC (including the carry-out). Make sure that the carry-in is at logic zero. Perform the operations mentioned in Table 5.3 and record your observations.*

Operation	First input (A)				Second input (B)				Output
	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	(darken lit LEDs)
6+2									○—○—○—○—○ C_o S_3 S_2 S_1 S_0
9+8									○—○—○—○—○ C_o S_3 S_2 S_1 S_0

Table 5.3: Observations: the full adder

Task 14. *An adder-subtractor can perform both addition and subtraction by representing signed numbers in their two's complement notation. Modify the inputs of the full adder IC as shown in Figure 5.15 to construct an adder-subtractor.*

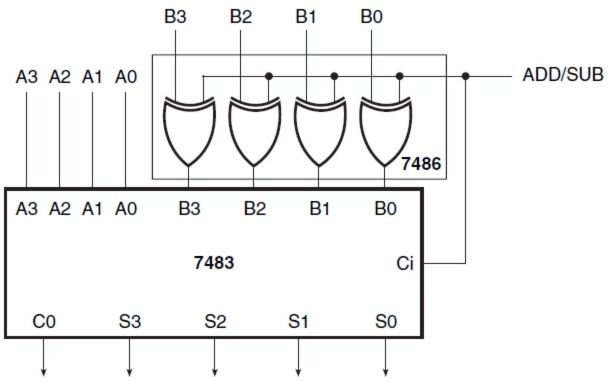


Figure 5.15: Adder-subtractor schematic

Task 15. *Perform the operations mentioned in Table 5.4 and record your observations.*

Operation	First input (A)				Second input (B)				Add/Sub	Output
	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	C_i	(darken lit LEDs)
15-3										○ ○ ○ ○ ○ C_o S_3 S_2 S_1 S_0
8-13										○ ○ ○ ○ ○ C_o S_3 S_2 S_1 S_0

Table 5.4: Observations: the adder-subtractor

♣ The End ♣

Workshop 6: Design and Implementation of a Sequence Detector

Objective: To design, implement and verify a sequence detector as an example of a sequential logic circuit, using SystemVerilog and deploy on a Field Programmable Gate Array (FPGA)

Outcome: After successful completion of this session, the student will be able to

1. Design sequential logic circuits using state diagrams
2. Use SystemVerilog to implement a sequential circuit
3. Develop a test bench and verify the hardware design
4. Perform pin planning and deploy the design on an FPGA

Equipment Required:

1. A personal computer.
2. Intel Quartus Prime Lite Edition Design Software Version 20.1.1
3. ModelSim Intel FPGA Edition Version 20.1.1
4. Intel Cyclone IV Device Support file
5. DE0-Nano Development Board
6. Oscilloscope
7. Jumper wires - male-to-female (3 Nos.)

Components Required: None

6.1 Introduction

In this experiment, we will design and implement a sequence detector on an FPGA. As the initial step, the required boolean expressions will be derived using a state diagram and truth tables. Then the detector, along with a sequence generator, will be implemented in SystemVerilog and simulated using ModelSim. Finally, the design will be programmed into a FPGA and the waveforms will be observed using the oscilloscope.

6.1.1 Finite State Machines

A Finite State Machine (FSM) is an abstract computational model which can be used to model sequential logic circuits. An FSM can take only one of the finite number of possible states at any given time. Transitions among the states are well defined, and depend on the current state and the current inputs. In addition to state transitions, FSMs also produce outputs. Depending on how the output is generated, FSMs can be categorized into two types namely;

- Mealy machines - the outputs depend on both the current state and the current inputs
- Moore machines - the outputs depend only on the current state.

The number of states of an FSM determines the number of memory elements that is required for implementing that FSM. FSMs can be illustrated graphically using state diagrams.

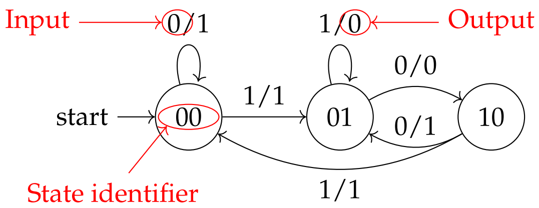


Figure 6.1: An example state diagram with three states

6.1.2 FPGAs and Development Boards

A Field Programmable Gate Array (FPGA) is an electronic chip (integrated circuit) inside which we can implement any digital circuit we want. Digital designers design their circuits, then express them in text using a Hardware Description Language (HDL) and use software tools to synthesize, place and route to implement it inside an FPGA.

When programming a microcontroller or a microprocessor, the software code we write is compiled into instructions which are to be executed by the microcontroller, which is a fixed digital circuit. On the other hand, when "programming" an FPGA, we describe our own circuit, which gets implemented. Therefore, we can design and implement our own custom processors inside an FPGA. Consequently, practical circuit design considerations such as placement of logic blocks and critical path delays must be taken into account. It is vital to understand this distinction.

FPGAs are field-programmable; that is, the circuit can be changed after deploying into a product. Therefore, they are widely used in products such as satellites, Mars rovers, scientific equipment such as particle detectors at LHC, high bandwidth network switches and video decoders. FPGA implementations of high-bandwidth algorithms are often faster and power-efficient than their generic CPU-based counterparts. For chips that can be manufactured in millions of units, like processors, it is economical to fabricate custom ICs instead, which are faster, consume less power and smaller than FPGAs.

For academic purposes and for prototyping needs, FPGA chips are available on development boards. A typical development board contains peripherals required for programming the FPGA along with switches and LEDs connected to the input/output pins of the FPGA. The development board that will be used in this experiment is the Terasic DE0-Nano board which consists of an Altera Cyclone IV FPGA.

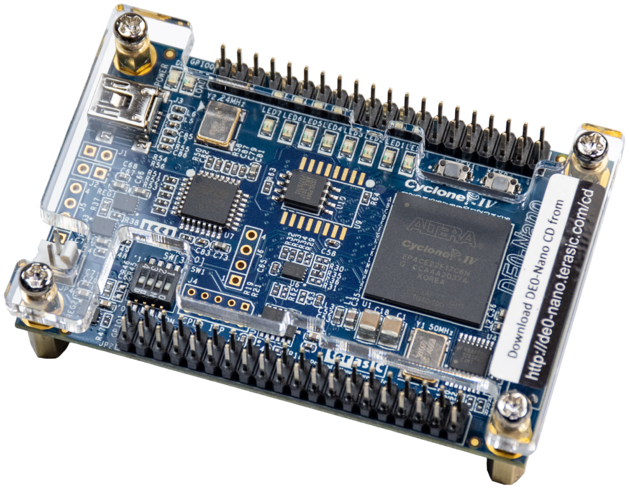


Figure 6.2: Terasic DE0-Nano development board with Altera Cyclone IV FPGA

6.1.3 HDLs and SystemVerilog

As the first step of programming an FPGA, we need to express our logic circuit in a text format. A Hardware Description Language (HDL) is a specialized computer language that can be used to describe the structure or the behaviour of logic circuits. Unlike a programming language, in an HDL, there is no control flow. That is, the statements are not executed one after the other. Instead, our hardware description is synthesized into a physical electronic circuit. However, some HDLs support non-synthesizable statements whose only purpose is to generate test benches, which are programs that run like software to drive inputs and check outputs of our hardware design. This is used to "verify" the hardware designs using simulators.

Verilog has become one of the most widely-used HDLs since its introduction in 1984. SystemVerilog, initially introduced as a Hardware Verification Language (HVL) has later been merged with Verilog to facilitate both hardware design and verification. While the synthesizable constructs of SystemVerilog are much similar to Verilog, testing and verification constructs support high-level programming language features such as Object-Oriented Programming.

6.1.4 Sequence Detectors

A *sequence detector* is used to detect a predefined binary sequence within a stream of incoming bits. Detecting flags (which usually indicate the beginning or the end of a data packet) in digital communication systems is a major application of sequence detectors. A sequence detector has a single-bit output which indicates whether the predefined sequence has arrived or not. Depending on the application, they are implemented in 2 different ways, namely, *overlapping* and *non-overlapping*. See figure 6.3 for a detailed illustration. A sequence detector can be considered as a Mealy machines since its output depends on both the current state and the current input.

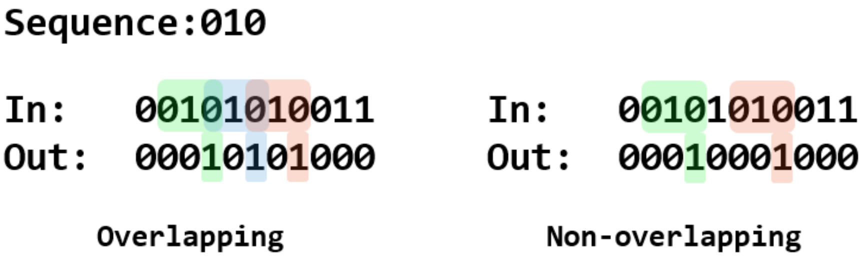


Figure 6.3: Types of sequence detectors

6.2 Pre-Lab

6.2.1 State Diagram and Truth Table

Task 1. *Consider the 3-bit binary sequence,* ***010****. Draw the state diagram for an* ***overlapping*** *sequence detector to detect this sequence. Name the states as follows;*

- $q_1 = 0, q_2 = 0$ - *none of the elements belonging to the sequence detected*
- $q_1 = 0, q_2 = 1$ - *'0' detected*
- $q_1 = 1, q_2 = 0$ - *'01' detected*
- $q_1 = 1, q_2 = 1$ - *'010' detected*

Task 2. *Complete the following truth table using the above state diagram. Assume that the states q_1 and q_2 will be stored in the first and the second JK flip-flops, respectively. Use 'X' for "don't care" terms.*

Current state		Input	Next state		Output	First flip-flop (q_1)		Second flip-flop (q_2)	
q_1	q_2	x	$\hat{q}_1$	$\hat{q}_2$	y	j_1	k_1	j_2	k_2

Table 6.1: Truth table for 3-bit sequence detector

Task 3. *Derive and simplify boolean expressions for y, j_1, k_1, j_2 and k_2, in terms of q_1, q_2 and x.*

Task 4. *Create a new project in Quartus and name it as* ***sequence_detector****. Create a new* ***SystemVerilog HDL*** *file (under* ***Design Files*** *category) named* ***jk_ff.sv****. Copy-paste into jk_ff.sv, and complete the following code snippet which implements a positive edge triggered JK flip-flop. Refer section 5.2.2 for more information.*
Hint: Refer the truth table of a positive edge triggered JK flip-flop.

```
module jk_ff(reset, clk, j, k, q);  // Module definition start

  input reset, clk, j, k; // Define inputs
                // The 'reset'signal is used to reset the internal state of
                // the flip–flop to logic zero.

  output q;   // Define outputs

  reg q;  // Allocate a register to hold the internal state.
        // Since both the output and the reg have the same name, they are
        // connected automatically.

  always @(posedge clk, posedge reset) begin  // Triggered at each positive edge
                          // of 'clk' or 'reset' signal.

    if (reset) begin
      q <= 0;    // If 'reset' is high, reset the internal state to zero.
    end
    else begin     // Otherwise, update q according to truth table.
      // ******************************
      // ****** your code goes here *****
      // ******************************
    end
  end

endmodule   // Module definition end
```

Listing 6.1: Positive edge triggered JK flip-flop

Task 5. *Set the* ***jk_ff*** *module as the top-level entity. Compile and simulate the module using the "Quartus Simulation Waveform Editor" and verify the functionality.*

6.3 Implementing the Sequence Detector

SystemVerilog facilitates *hierarchical* designing in order to allow design reused. Hence, we can use two instances of the previously constructed JK flip-flop to create the sequence detector. Example 3 demonstrates how to insert an instance of a previously implemented module.

Example 3. *Consider the schematic illustrated in figure 6.4. The module named* ***top*** *has two instances of the module named* ***bottom****. Both instances share the same* ***aa*** *input which is also connected to the* ***top*** *module's input* ***xx****. In such cases, a separate wire (****aa_to_aa****) has to be defined. However, the input* ***bb*** *of the first* ***bottom*** *instance and the output* ***cc*** *of the second* ***bottom*** *instance are directly connected to the* ***top*** *module's input* ***yy*** *and output* ***zz*** *respectively. Hence, they don't need additional wires to be defined. Code snippets below show how to implement this schematic in SystemVerilog.*

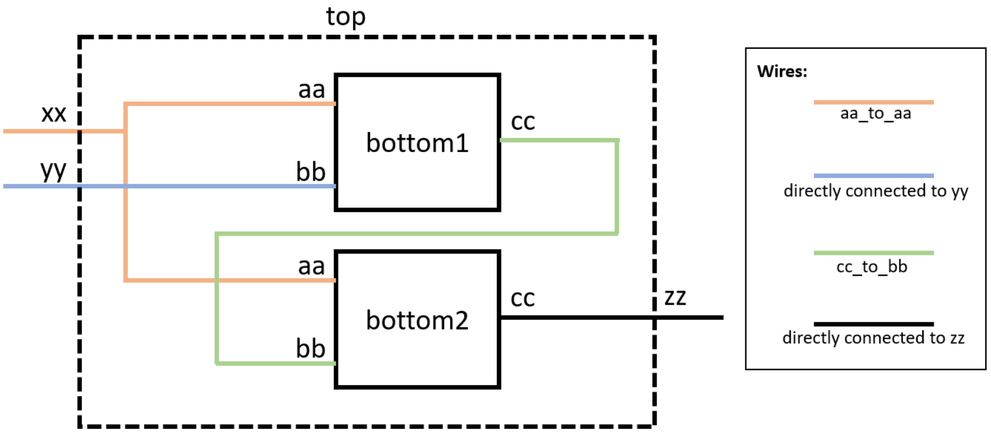


Figure 6.4: Example schematic of a hierarchical design. Wire names are also shown on the right-hand side

```
module bottom(aa, bb, cc); // Module definition start

  input aa, bb;   // Define inputs

  output cc;  // Define outputs

  reg cc;     // Allocate a register to hold output value.

  always @(posedge aa) begin      // Triggered at the positive edge of 'aa'.

    cc <= bb & cc;      // Some logic goes here.

  end

endmodule      //  Module definition end
```

Listing 6.2: Definition of **bottom** module

```
module top(xx, yy, zz); // Module definition start

  input xx, yy;   // Define inputs

  output zz;      // Define outputs

  wire aa_to_aa;  // Define a wire to connect 'aa's of the two instances.
```

```
   wire cc_to_bb;  // Define a wire to connect 'cc' and 'bb'.

   assign aa_to_aa = xx;   // Need to connect 'aa_to_aa' to 'xx' as well.

   // First instance of bottom module
   bottom bottom1(.aa(aa_to_aa),  // Connect 'aa' of the instance to 'aa_to_aa' wire
                  .bb(yy),        // Connect 'bb' of the instance directly to 'yy'
                  .cc(cc_to_bb)); // Connect 'cc' of the instance to 'cc_to_bb' wire

   // Second instance of bottom module
   bottom bottom2(.aa(aa_to_aa),   // Connect 'aa' of the instance to 'aa_to_aa' wire
                  .bb(cc_to_bb),   // Connect 'bb' of the instance to 'cc_to_bb' wire
                  .cc(zz));        // Connect 'cc' of the instance to 'zz'

endmodule          // Module definition end
```

Listing 6.3: Definition of **top** module

Task 6. *Create a new SystemVerilog file named* ***sequence_detector.sv*** *under the Quartus project that was created earlier (hereafter, referred to as "the project"). Copy and complete the code snippet given below in order to implement the sequence detector (to detect the sequence* ***'010'****) using the JK flip-flop module developed under Task 4. Refer Table 6.1, Task 3 and Example 3.*

```
module sequence_detector(reset, clk, x, y);

   input reset, clk, x; // Define inputs

   output y; // Define outputs

   wire j1, k1, q1, j2, k2, q2;    // Define wires

   jk_ff ff1(                  // First JK flip–flop to store state q1
         .reset(reset),
         .clk(clk),
         .j(j1),
         .k(k1),
         .q(q1));

   // Include the second JK flip–flop to store state q2
   // ********************************
   // ****** your code goes here *****
   // ********************************


   assign y = q1 & ~q2 & ~x;   // Boolean expression for y

   // Write 'assign' statements (boolean expressions) for flip–flop inputs
   // ********************************
   // ****** your code goes here *****
   // ********************************

endmodule
```

Listing 6.4: Sequence detector

Task 7. *Make* ***sequence_detector*** *the top-level entity and compile to see whether there are any syntax errors.*

In order to verify the implementation we can use the **RTL Viewer** to visualize the interconnections and the inferred logic blocks. It will display how each block of the design has been connected hierarchically. Figure 6.5 illustrates the **RTL Viewer** output of the **top** module in Example 3.

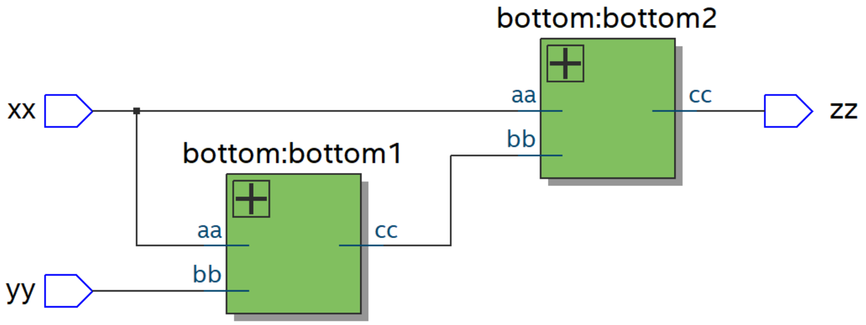


Figure 6.5: RTL Viewer output - **top** module in hierarchical design example

Task 8. *Goto* ***Tools > Netlist Viewers > RTL Viewer*** *from the menu bar to open the* ***RTL Viewer****. Interior of a block can be viewed by double-clicking on it. Verify the connections of* ***sequence_detector****.*

6.4 Using ModelSim for Simulation

In order to simulate the sequence detector, we will first create a sequence generator which generates a pseudo-random bit stream. Afterwards, we will create a test bench including both the sequence generator and the detector and simulate it using ModelSim.

6.4.1 Sequence Generator

The code snippet given below implements a sequence generator which is also known as a Linear Feedback Shift Register (LFSR). The schematic of the LFSR is illustrated in figure 6.6. The **init_seq** parameter in the snippet specifies the initial internal state of the LFSR.

```
module sequence_generator(reset, clk, seq); // Module definition start

  localparam init_seq = 4'b1000;  // Initial seed.
                                  // Can be any 4–bit sequence except 4'b0000.
                                  // 'localparam' specifies that this is just a
                                  // constant which will be resolved at compile time.


  input reset, clk;  // Define inputs


  output seq;        // Define outputs


  reg[$size(init_seq)–1:0] seq_reg;  // Linear Feedback Shift Register (LFSR). Just
                                     // an ordinary register which can store many
                                     // bits :)

                                     // '$size()' is a Verilog system function which
                                     // will be resolved at compile time. It
                                     // returns the length of its argument.


  assign seq = seq_reg[$size(init_seq)–1];  // Connect the output to the MSB
                                            // of the LFSR.
  always @(posedge clk, posedge reset) begin  // Triggered at each positive edge
                                              // of 'clk' or 'reset'

    if (reset) begin    // If 'reset' is high, reset LFSR to initial sequence.
      seq_reg <= init_seq;
    end

    else begin        // Else, generate the next sequence
      seq_reg <= {seq_reg[$size(init_seq)−2:0], seq_reg[$size(init_seq)−1] ^ seq_reg[$size(init_seq)−2]};
    end
  end

endmodule
```

Listing 6.5: Sequence generator

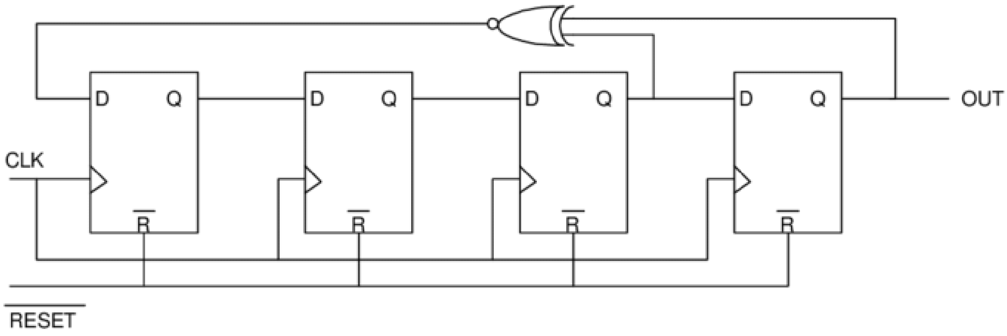


Figure 6.6: Schematic of a Linear Feedback Shift Register

Task 9. *Add a new SystemVerilog file named **sequence_generator.sv** to the project and copy-paste the above code snippet. Change the parameter **init_seq** to any 4-bit sequence of your preference other than **4'b0000** and save.*

6.4.2 Test Bench

A test bench is somewhat similar to an ordinary module, but has few additional properties such as;

- includes timing information for the simulator
- does not have any inputs or outputs
- includes un-synthesizable code (eg: delays - #1, utility functions - $display()) and hence, cannot be compiled

The test bench includes the synthesizable modules that are to be tested, and provides internally generated signals to the inputs of those modules. This way, we can observe the output of the synthesizable modules through a simulator such as ModelSim.

Implementation

Our test bench should;

- include one instance each of sequence generator and detector
- provide **clk** and **reset** signals to the above modules
- internally connect the output **seq** of the generator to the input **x** of the detector

- extract the output **y** of the detector

The **clk** and **reset** signals can be generated using the unsynthesizable delay operator (#*n* where # is the operator and *n* specifies the duration in number of timescales).

Task 10. *Add a new SystemVerilog file named **tb_sequence_detector.sv** under the project. Insert and complete the code given below, which implements a test bench for the sequence detector and the generator. Set the test bench as the top-level entity.*

```
‘timescale 10ns/1ns // Timing information for the simulator.
                // 10ns – means #1 corresponds to 10 nano seconds.
                // 1ns – means the accuracy should be upto 1 nano second.

module tb_sequence_detector();  // Module definition start.
                          // Just another ordinary module, but without
                          // any inputs/outputs.

   reg reset, clk; // Registers to hold the values for 'clk' and 'reset'

   wire x, y;      // Wires to connect inputs and outputs of the modules being tested.


   // Include an instance of sequence_generator
   // Note that wire 'x' should be connected to the 'seq' output of the generator
   // ********************************
   // ****** your code goes here *****
   // ********************************


   // Include an instance of sequence_detector
   // ********************************
   // ****** your code goes here *****
   // ********************************


   initial begin      // Triggered at the very beginning. This is not synthesizeable.
      clk = 0;       // Initialize 'clk' to zero
      reset = 0;      // Initialize 'reset' to zero

      // All the modules need to be reset to start from a definite internal state.
      // Hence, we need to generate a 'reset' posedge in the beginnig.
      #1 reset = 1;   // After 10ns, make 'reset' high
      #1 reset = 0;   // After another 10ns, make 'reset' low
   end

   always #1 clk <= ~clk;  // Triggered at each 10ns period.
                      // Each time, the state of 'clk' will be inverted, causing
                      // it to act as a clock signal. Note that the
                      // delay operator '#' is not synthesizable.

endmodule // Module definition end
```

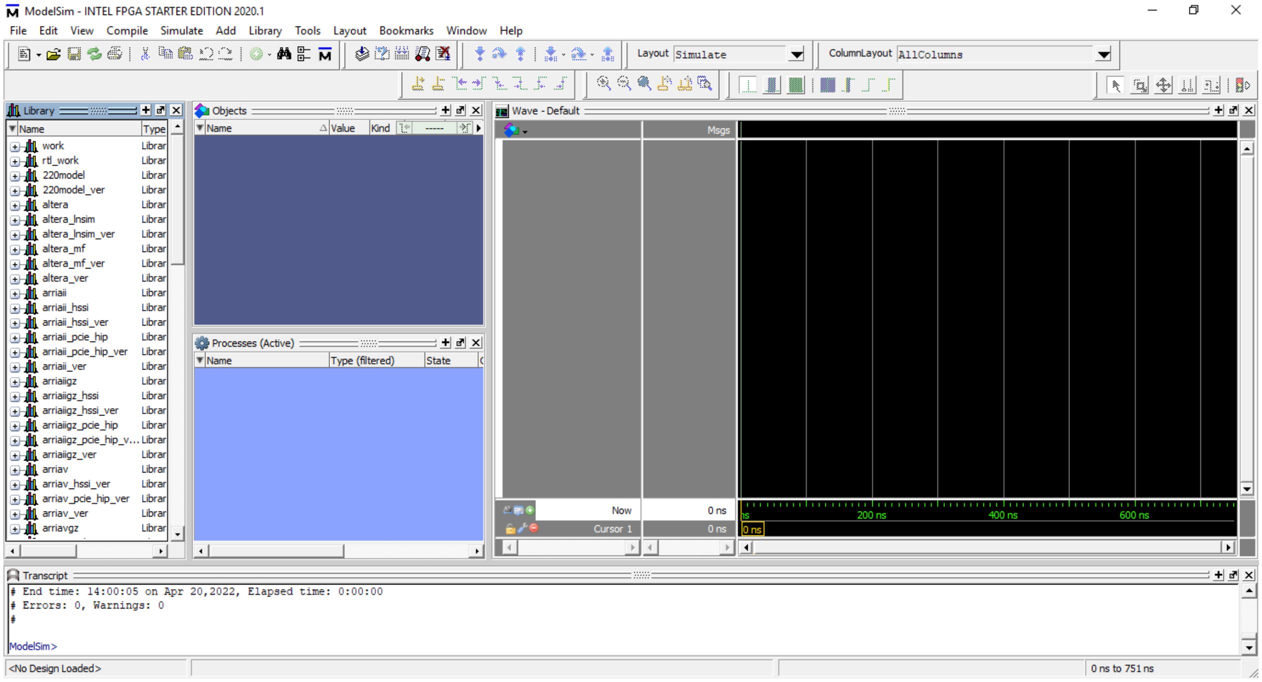
Listing 6.6: Test bench

Since the test bench includes un-synthesizable code, it cannot be compiled completely. However, we can perform the **Analysis** and **Elaboration** steps of the compilation workflow, which are sufficient for simulation.

Task 11. *After setting the test bench as the top-level entity, goto **Processing > Start > Start Analysis & Elaboration** from the menu bar. Monitor the **Messages Pane** for any error messages, and debug if any.*

Simulation

To start the simulation, goto **Tools > Run Simulation Tool > RTL Simulation** from menu bar. This will open up a ModelSim window. See figure 6.7 (note that the locations of the panes may differ depending on your initial settings).



```
# End time: 14:00:05 on Apr 20,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
#

ModelSim>
```

Figure 6.7: ModelSim main window

Task 12. *Go through the following instructions to set-up and run the simulation.*

1. *Locate the **Library*** Library *pane.*

2. *Expand the **work*** ⊞ work *node.*

3. *Locate **tb_sequence_detector**. Right click on it and select **Simulate**. This will open up **Sim*** sim *tab which lists all the module instances in the design, in a hierarchical structure. In addition, signals of the currently selected instance will be displayed under the **Objects*** Objects *pane.*

4. *To view a signal on the **Wave*** Wave *pane, right-click on the interested signal in **Objects** pane and select **Add Wave**. Do this for all the waves **clk, reset, x** and **y**.*

5. *Set the simulation duration at the **Run Length** text box* 100 ns *to **100 ns**.*

6. *Press **Run*** *button to run the simulation. See figure 6.8.*

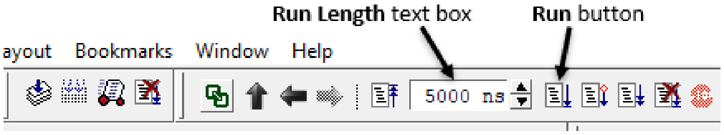


Figure 6.8: **Run Length** text box and **Run** button

6.5 Programming the FPGA

Under this section, we will program **DE0 Nano** development board with **Intel Altera Cyclone IV** FPGA and observe the waveforms through an oscilloscope.

6.5.1 Implementing the Top-level Module

As the first step, we will create a new top module which includes instances of the **sequence_generator** and the **sequence_detector**.

Task 13. *Add a new SystemVerilog file named **sequence_detector_top.sv** and insert the code given in the following snippet. Make this module the top-level entity of the project. Compile the project to see whether there are any syntax errors. Open the **RTL Viewer** and verify the connections.*

```
module sequence_detector_top(reset_inv, clk, x, y, blink);

  localparam clk_dev_coef = 10;  // This is a constant which is resolved at compile
                                 // time. Since the frequency of the FPGA clock
                                 // (runs at 50 MHz) is too high to observe clearly,
                                 // it is divided by 2^(clk_dev_coef+1).

  input reset_inv, clk;   // Define inputs. Note that pushbuttons are considered to
                          // be logic–high when not pressed. Hence, to use as the
                          // ’reset’signal for the rest of the modules, it should
                          // be inverted.

  output x, y, blink;     // Define outputs. ’blink’ will be connected to a LED
                          // which blinks in an observable frequency.

  wire clk_dev, reset;    // Define wires for divided clock and inverted reset
                          // pushbutton signals.

  reg[24:0] clk_div_reg;  // Register to divide the clock


  sequence_detector sd(                // Instance of sequence_detector
               .reset(reset),
               .clk(clk_dev),
               .x(x),
               .y(y));

  sequence_generator sg(               // Instance of sequence_generator
               .reset(reset),
               .clk(clk_dev),
               .seq(x));


  assign clk_dev = clk_div_reg[clk_dev_coef]; // Assign divided clock
  assign blink = clk_div_reg[24];             // Assign ’blink’
  assign reset = ~reset_inv;                  // Invert reset pushbutton signal


  always@(posedge clk, posedge reset) begin   // Incrementing the clock–div register

    if (reset) clk_div_reg <= 0;

    else clk_div_reg <= clk_div_reg + 1;

  end
```

```
endmodule
```

Listing 6.7: Top module for FPGA

6.5.2 Pin Assignment

Everything upto now did not have any dependency on the hardware to which the design is to be uploaded. However, the next step, which is called **Pin Assignment**, depends on the FPGA model that is being used. Under this step, we will assign physical pins of the FPGA to the I/O nodes of the module defined in SystemVerilog (i.e., **reset_inv, clk, x, y** and **blink**). There are two levels of pin mapping that we have to consider, which can be stated as follows;

- SystemVerilog design to the FPGA (e.g.: **clk** should be connected to FPGA pin **PIN_R8**)
- FPGA to the development board (e.g.: **KEY[0]** of the board which is a pushbutton is connected to the FPGA pin **PIN_J15**)

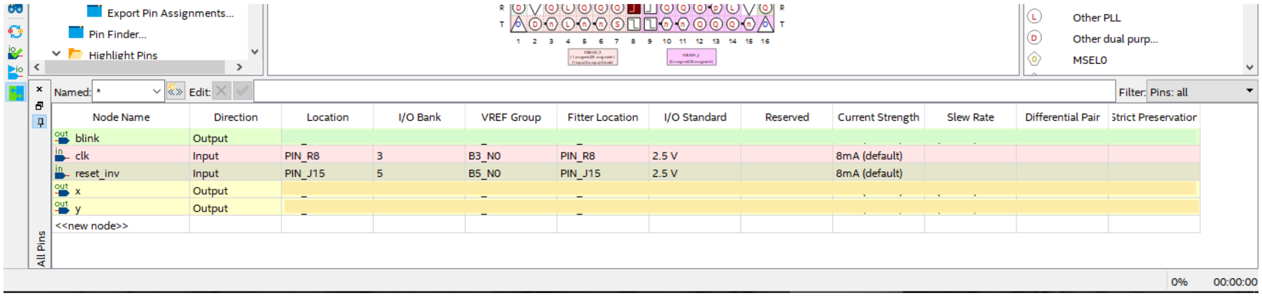
The first mapping is done through Quartus **Pin Planner** tool. The second mapping is specified in the user manual of the development board.

Task 14. *Complete the following pin assignment table by referring the DE0 Nano user manual.*

I/O node	DE0 Nano signal	FPGA pin
reset_inv	KEY[0]	PIN_J15
clk	CLOCK50	PIN_R8
x	GPIO_00	
y	GPIO_01	
blink	LED[1]	

Table 6.2: Pin assignment table

Task 15. *After compiling the top-level entity **sequence_detector_top**, click **Assignments > Pin Planner** from the menu bar to open the pin planner. Based on Table 6.2, assign the pins by specifying the pin name under the **Location** column of the table displayed at the bottom of the window (see Figure 6.9). Once the assignment is complete, close the pin planner and compile the module once again.*



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservatior
out blink	Output										
in clk	Input	PIN_R8	3	B3_N0	PIN_R8	2.5 V		8mA (default)			
in reset_inv	Input	PIN_J15	5	B5_N0	PIN_J15	2.5 V		8mA (default)			
out x	Output										
out y	Output										
<<new node>>											

Figure 6.9: Pin Planner

6.5.3 Installing Drivers

To install the usb driver for the development board, follow the instructions given below.

1. Connect the DE0 Nano board to the PC using the provided cable.

2. Open **Device Manager**.

3. Locate **USB Blaster** listed under **Other devices**.

4. Right-click on to of **USB Blaster** and select **Update driver**.

5. From the opened dialog, select **Browse my computer for driver software**

6. Press the **Browse** button and navigate to **<Quartus installation folder>/quartus** and select **drivers** folder. E.g.: navigate to "C:/intelFPGA/textunderscore lite/20.1/quartus" and select "drivers"

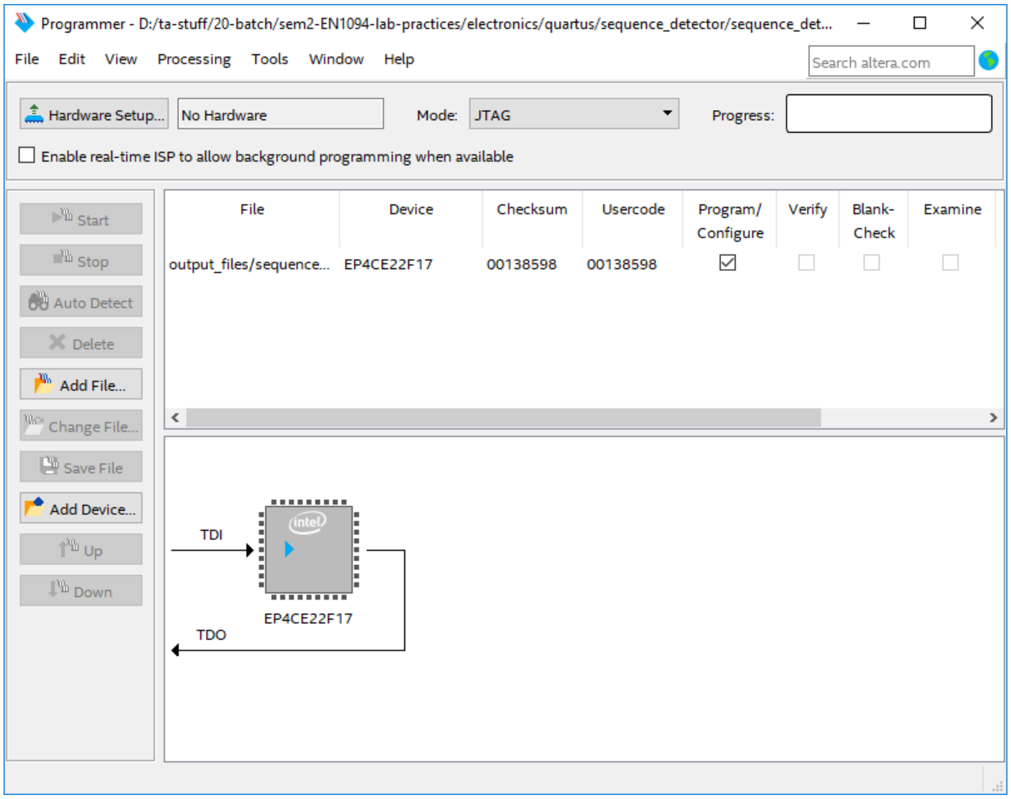
7. If the installation completes successfully, you are good to go.

6.5.4 Uploading the Design

To program the FPGA with the design, complete the following steps.

Task 16. *Make sure that you compiled the design after all the changes have been made. While the development board is connected to the PC;*

1. *Goto* ***Tools > Programmer*** *from the menu bar. It will open the* ***Programmer*** *(see Figure 6.10).*



Programmer - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/sequence_detector/sequence_det...

Hardware Setup... No Hardware Mode: JTAG Progress:

□ Enable real-time ISP to allow background programming when available

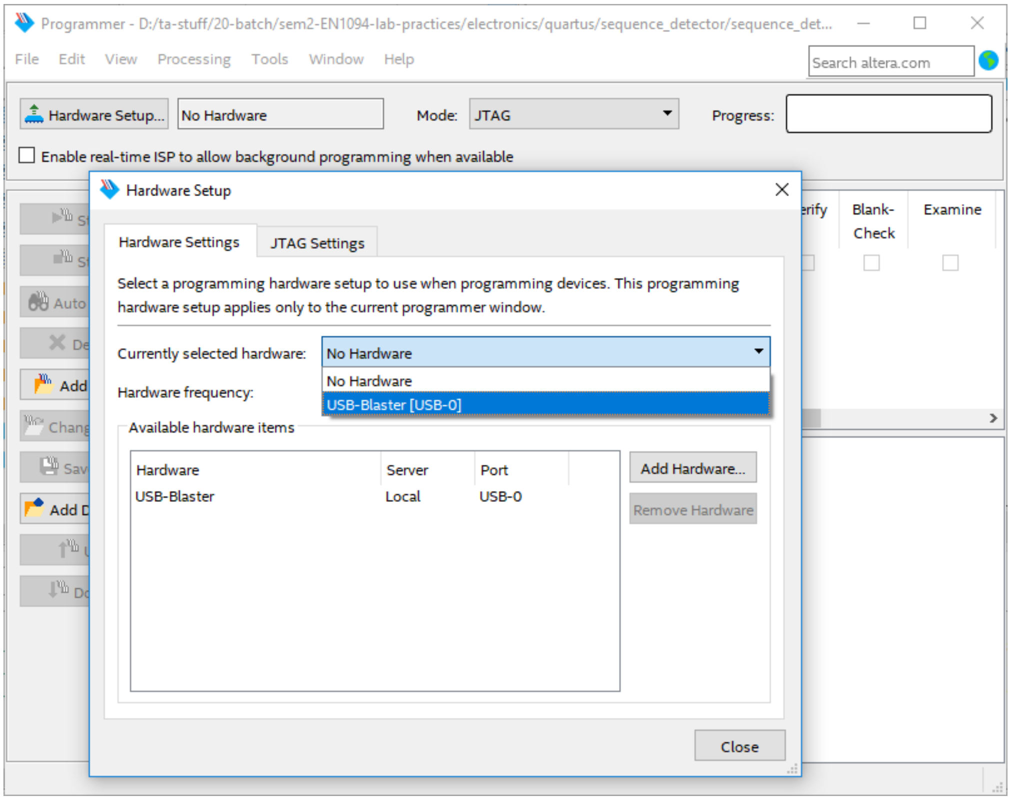
File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
output_files/sequence...	EP4CE22F17	00138598	00138598	☑	□	□	□

Figure 6.10: Programmer

2. *In the* ***Programmer****, select* ***JTAG*** *as the mode.*

3. *If the box next to* ***Hardware Setup...*** *buttons displays* ***No Hardware****, press* ***Hardware Setup...*** *button. It will open the* ***Hardware Setup*** *window.*

4. *In the* ***Hardware Settings*** *tab, set* ***USB-Blaster*** *as the* ***Currently selected hardware*** *and press* ***Close*** *(see Figure 6.11).*



Programmer - D:/ta-stuff/20-batch/sem2-EN1094-lab-practices/electronics/quartus/sequence_detector/sequence_det...

No Hardware

Mode: JTAG

□ Enable real-time ISP to allow background programming when available

Hardware Setup

Hardware Settings

JTAG Settings

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

Currently selected hardware: No Hardware

No Hardware

USB-Blaster [USB-0]

Hardware frequency:

Available hardware items

Hardware	Server	Port
USB-Blaster	Local	USB-0

Figure 6.11: Hardware Setup

5. *Press the **Start** button on the **Programmer** window to upload the design to the FPGA.*
6. *If the design has been uploaded successfully, LED[1] should start blinking.*

6.6 Observing the Waveforms

Due to the very high operational frequency of the sequence detector, we cannot verify the functionality with the naked eye (say for instance, by using a logic probe). Hence, we will use the oscilloscope to observe the waveforms.

Task 17. *Connect 3 male-to-female jumper wires to **GPIO_00**, **GPIO_01** and the **Ground** pin. Refer the DE0 Nano user manual to identify the pins. Connect oscilloscope ground probe to the FPGA **Ground**. Observe and draw the waveforms at **GPIO_00** and **GPIO_01**. Measure the frequency of the divided clock.*

♣ The End ♣