

ECE385

DIGITAL SYSTEMS LABORATORY

Introduction to Quartus Prime

A Simple Design in SystemVerilog Using Altera Quartus Prime 18.0 Lite Edition

Design a full-adder and use it to design a 2-bit adder:

You need to be on a machine with Linux or Windows 7, 8, or 10, either on your own machine or in the ECE Open Lab (3022 ECEB). A Linux version is also available online for your own Linux system. To perform this exercise on your own computer, you will need to install the Quartus Prime software (you may download the installer from the link below). Follow these steps to fully install the software:

- Install Quartus Prime Lite Edition v18.0, ModelSim-Intel FPGA Edition, and Cyclone IV device support from the Intel FPGA website (https://fpgasoftware.intel.com/18.0/?edition=lite&platform=windows&download_manager=dlm3). You do not have to install the three downloads separately. Rather, if the downloads are placed in the same directory, they will all be automatically selected for installation through the main installer, QuartusSetupWeb-XXX-XXX.exe.
- Once the installation has completed, check the box **Launch USB Blaster II driver installation** upon existing the Quartus installation, then follow the prompt to complete the USB Blaster driver installation.
- Alternatively, you can manually install the Altera USB Blaster driver through the following steps:
 1. Plug the USB-Blaster download cable into PC. The **Found New Hardware** dialog box will appear.
 2. Select **Locate and install driver software (recommended)**.
 3. Select **Don't search online**.
 4. When you are prompted to **Insert the disc that came with your USB-Blaster**, select **I don't have the disc. Show me other options**.
 5. Select **Browse my computer for driver software (advanced)** when you see the **Windows couldn't find driver software for your device**.
 6. Click **Browse** and browse to the **<Path to Quartus Prime installation>\drivers\<cable type>** directory.
 - For USB-Blaster cables, the driver is in the **<Path to Quartus Prime installation>\drivers\usb-blaster**
 - Note: Do not select the x32 or x64 directories.
 7. Click **OK**.
 8. Select the **Include subfolders** option and click **Next**.
 9. If you are prompted **Windows can't verify the publisher of this driver software**, select **Install this driver software anyway** in the **Window Security** dialog box.

10. The installation begins.

- Make sure the ModelSim-Altera directory is entered properly.
 1. Go to **Tools->Options** from the menu bar. Navigate to the **EDA Tool Options** page.
 2. Enter the corresponding directory next to **ModelSim-Altera**. The default path on Windows is < **Path to Quartus Prime installation**
> \18.0\modelsim_ase\win32aloem.
 3. Click **OK**.

You are now ready to begin using Quartus. Below is a sample 2-bit adder project.

Create a New Project:

- From the **File** menu select **New Project Wizard**. Click **Next** to pass the intro screen.
- The window in Figure 1 will appear. Fill in the fields from Figure 1 (make sure there are no spaces in any of your entries). The program will ask you if it should create the specified directory if it does not exist; choose **yes**.
- Select **Next** on page 2 without adding any files.
- On page 3, select the device family **Cyclone IV E**, make sure the second option under Target device is selected, and chose **EP4CE115F29C7** in the Available devices list. See Figure 2.
- Click **Next** on page 4. Select ModelSim-Altera as the simulation tool, and SystemVerilog HDL as the simulation format. See Figure 3. Click **Finish** on page 5.
- You should see an entry for the project in the **Project Navigator** window. It should appear as in Figure 4.

New Project Wizard

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

What is the working directory for this project?

C:\ece385\2Bit_Adder

What is the name of this project?

adder2

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

adder2

Use Existing Project Settings...

< Back Next > Finish Cancel Help

Figure 1

New Project Wizard

Family & Device Settings [page 3 of 5]

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 II software in which your target device is supported, refer to the [Device Support List](#) webpage.

Device family

Family: Cyclone IV E

Devices: All

Target device

☐ Auto device selected by the Fitter

☒ Specific device selected in 'Available devices' list

☐ Other: n/a

Show in 'Available devices' list

Package: Any

Pin count: Any

Core Speed grade: Any

Name filter:

☒ Show advanced devices

Available devices:

Name	Core Voltage	LEs	User I/Os	Memory Bits	Embedded multiplier 9-bit elements	PLL	Global Clocks
EP4CE115F23C9L	1.0V	114480	281	3981312	532	4	20
EP4CE115F23I7	1.2V	114480	281	3981312	532	4	20
EP4CE115F23I8L	1.0V	114480	281	3981312	532	4	20
EP4CE115F29C7	1.2V	114480	529	3981312	532	4	20
EP4CE115F29C8	1.2V	114480	529	3981312	532	4	20
EP4CE115F29C8L	1.0V	114480	529	3981312	532	4	20
EP4CE115F29C9L	1.0V	114480	529	3981312	532	4	20
EP4CE115F29I7	1.2V	114480	529	3981312	532	4	20
EP4CE115F29I8L	1.0V	114480	529	3981312	532	4	20

< Back Next > Finish Cancel Help

Figure 2

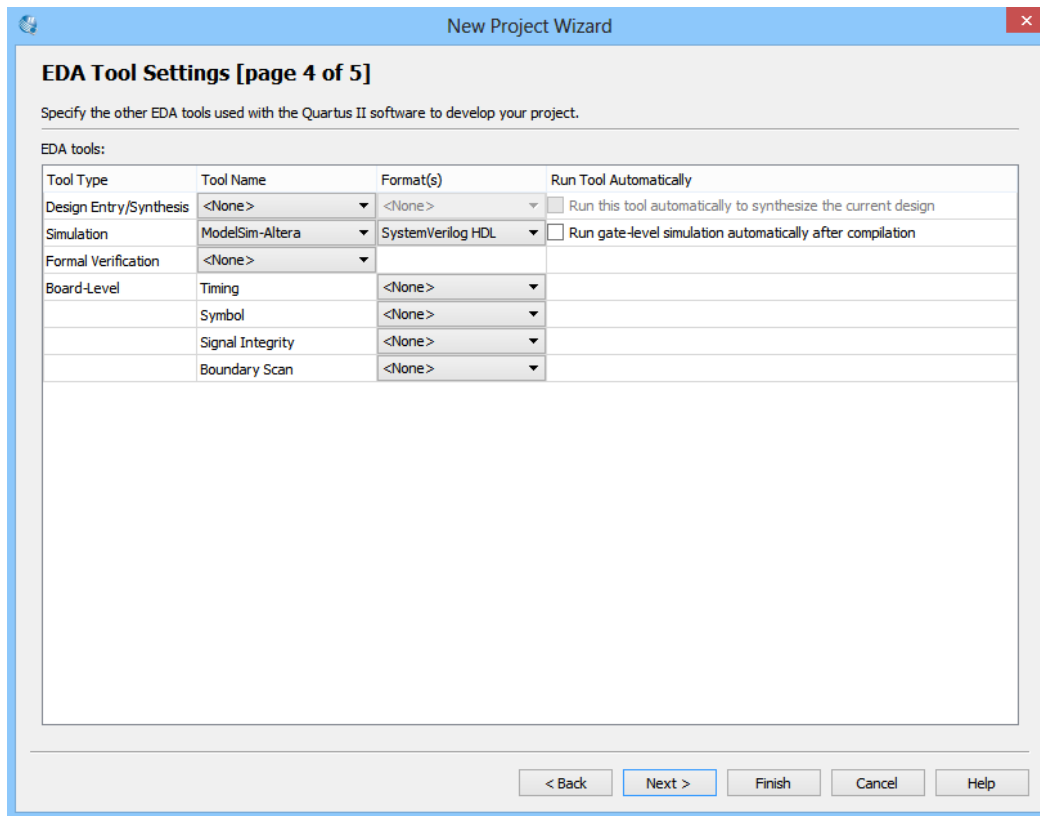


Figure 3

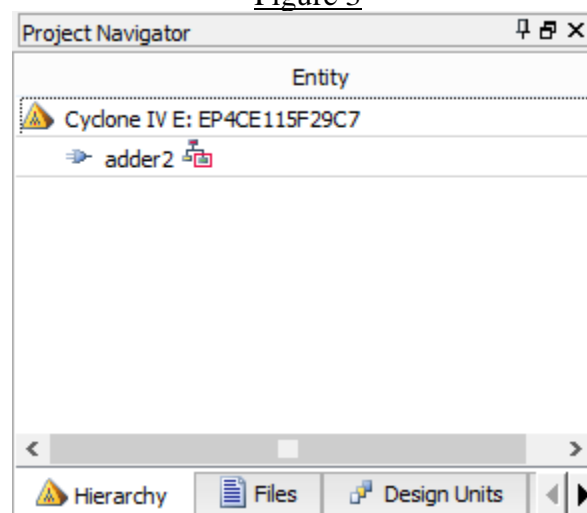


Figure 4

Create a Full-Adder Entity:

Now you can start entering your design. In the **File** menu, select **New....** Select **SystemVerilog HDL File** in the window that pops up as shown in Figure 5 and click **OK**. You will be presented with a blank SystemVerilog file. Enter the code below to define the full adder unit. Save the file as full_adder.sv. Figure 6 shows schematic representations of what the code defines; if you do not

understand what the code means, you should review the Introduction to SystemVerilog on pages IST.1-25.

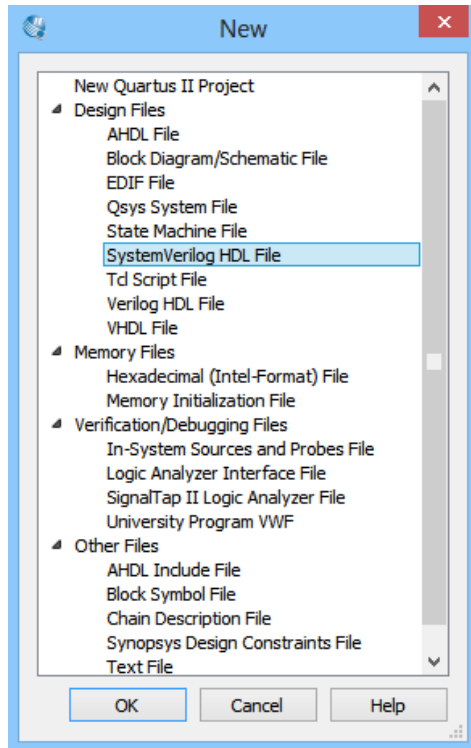


Figure 5

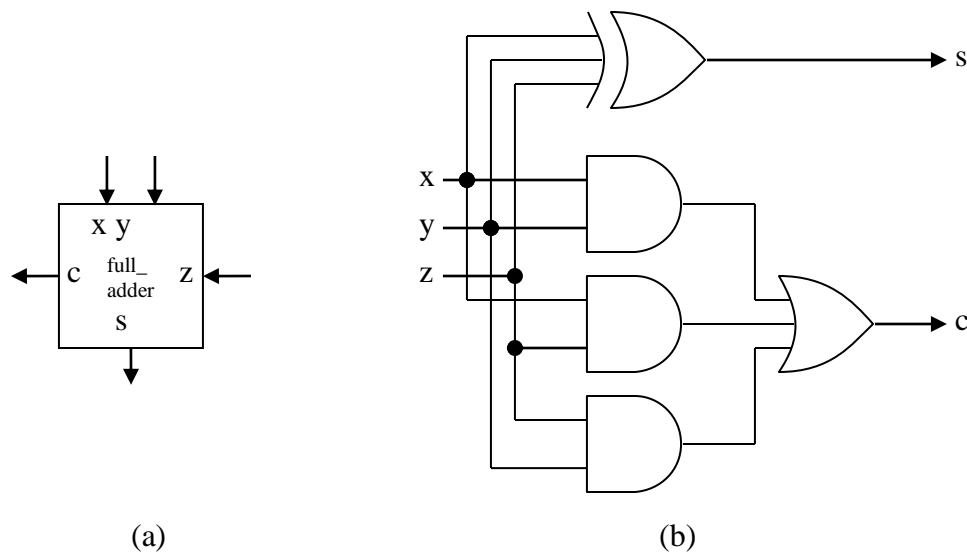


Figure 6: Schematic depiction of the meaning behind the Full Adder
(a) module block and (b) combinational logic.

```
module full_adder (input x, y, z,
                  output s, c);
```

```

assign s = x^y^z;
assign c = (x&y)|(y&z)|(x&z);

```

```

endmodule

```

Create an Entity for 2-Bit Adder:

Now, follow the same steps to create a new file that defines a 2-bit adder with inputs A (2-bits), B (2-bits) and Cin, and outputs Sum (2-bits) and Cout. The code is given below. Save the file as adder2.sv.

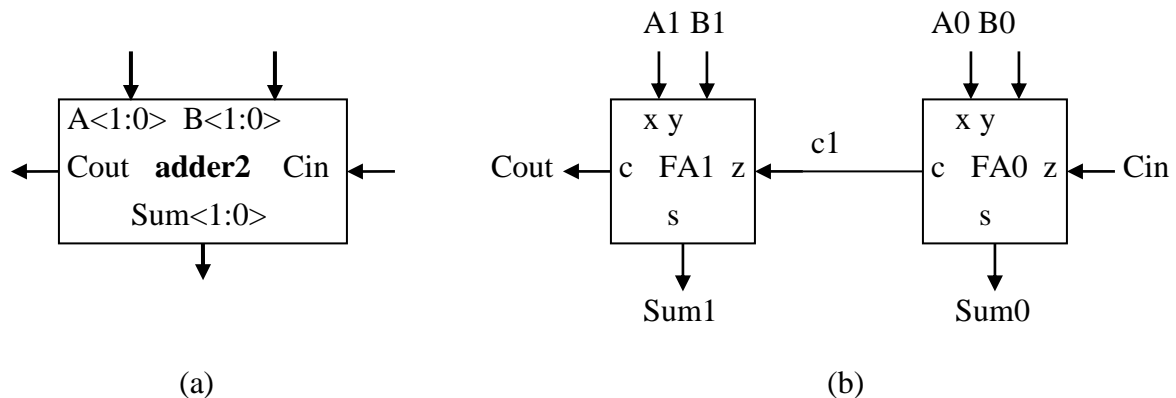


Figure 7: Module depiction of the meaning behind the 2-Bit Addder
(a) module block and (b) procedural block.

```

module adder2 (input [1:0] A, B,
               input  c_in,
               output [1:0] S,
               output  c_out);

```

```

// Internal carries in the 2-bit adder

```

```

logic  c1;

```

```

full_adder FA0 (.x (A[0]), .y (B[0]), .z (c_in), .s (S[0]), .c (c1));
full_adder FA1 (.x (A[1]), .y (B[1]), .z (c1), .s (S[1]), .c (c_out));


```

```

endmodule

```

Once you have entered and saved the code for adder2.sv, click the **Start Analysis & Synthesis** button (🔍) in the toolbar. Quartus Prime will check the code for correct syntax and generate errors or warnings if there is anything wrong or questionable about your code. A message pops up when the program has finished; you should get no errors or warnings if you have entered the code correctly. If you find errors in synthesis, correct the code, save the file, and run **Analysis & Synthesis** again.

Quartus Prime also builds the hierarchy in the **Project Navigator** window, which should look like figure 7. Next, click the **Start Compilation** button () in the toolbar. This time, you will get a few warnings about timing characteristics and load capacitances. You can ignore these. Now you have created a 2-bit adder and you can simulate the design.

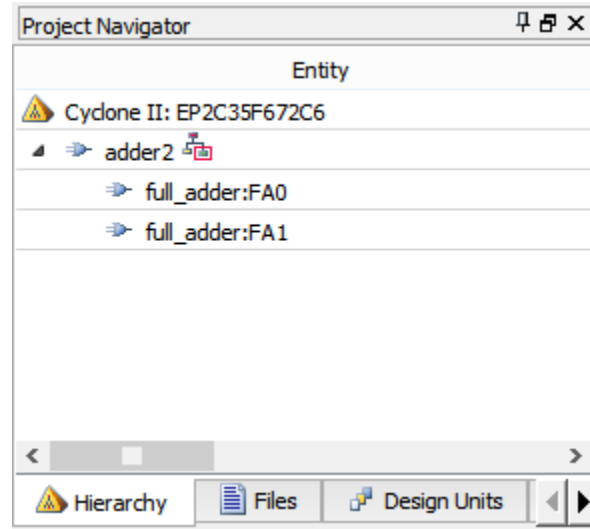




Figure 8

Using Simulator:

To create a testbench for your design and do simulations in ModelSim, please follow the instructions starting from IQT. 20.

Entering Pin Assignments:

Click the **Pin Planner** button () in the toolbar to open the Pin Planner. You will see the pin layout of the Cyclone IV chip on the top right of the window, as well as the pin assignment table at the bottom of the window. Enter the assignments from Table 1 in sequence. The planner should look like Figure 9. Save the assignment and recompile the design (). Rows with white background mean that those signals have not been assigned to any pins, and rows which have been assigned to pins will appear with color background.

Port Name	Location	Comments
Cin	PIN_AB28	On-board slider switch (SW0)
A[0]	PIN_AC28	On-board slider switch (SW1)
A[1]	PIN_AC27	On-board slider switch (SW2)
B[0]	PIN_AD27	On-board slider switch (SW3)
B[1]	PIN_AB27	On-board slider switch (SW4)
S[0]	PIN_G19	On-Board LED (LEDR0)
S[1]	PIN_F19	On-Board LED (LEDR1)
Cout	PIN_E19	On-Board LED (LEDR2)

Table 1: Pin Assignments

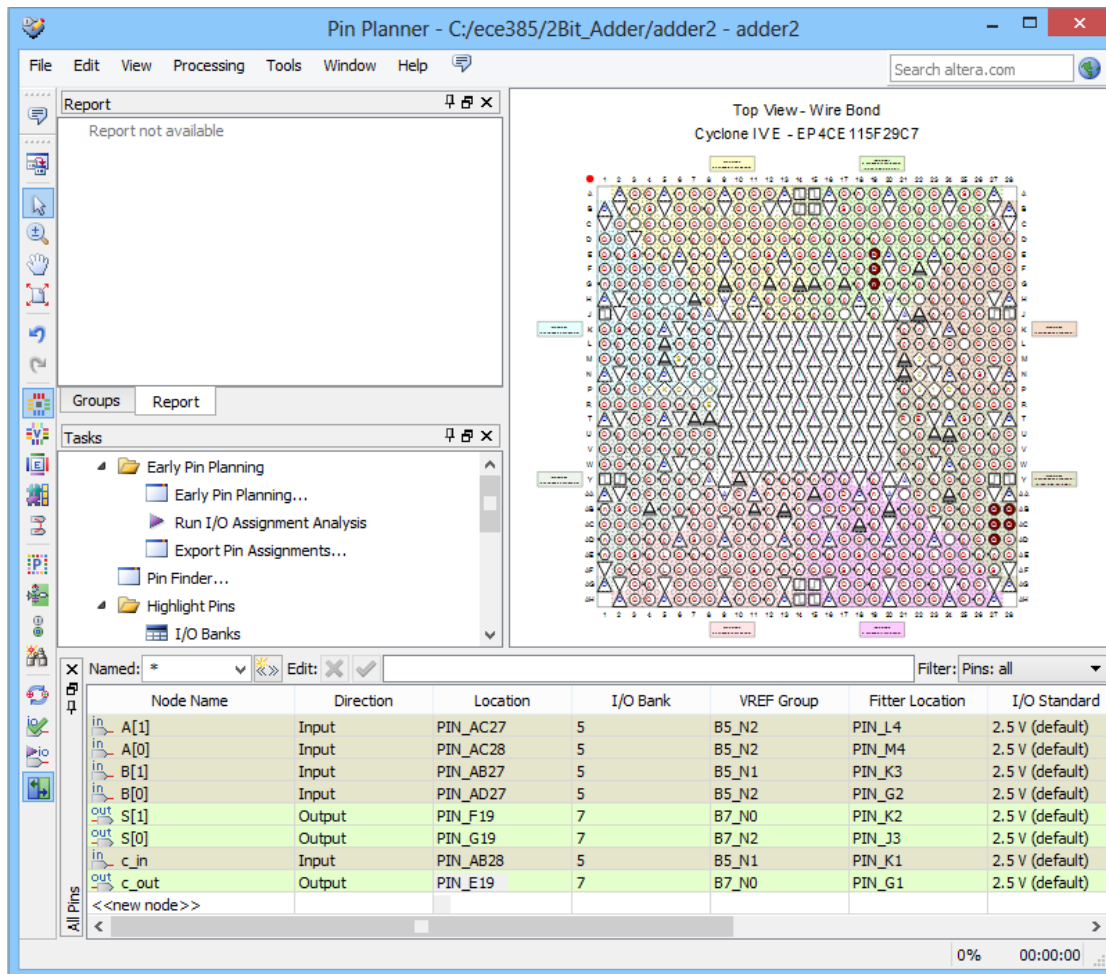


Figure 9

Programming the FPGA:

Plug in the FPGA power, and connect the FPGA Blaster port to the computer with the included USB cable. Click the **Programmer** button (🖱️) to open the programmer. The screen should list the file *adder2.sof*. Turn on the FPGA power, and click the **Hardware Setup...** button. Select **USB Blaster** from the **Currently selected hardware** dropdown list, and click **Close**. Back in the programmer, check the box next to *adder2.sof* under **Program/Configure**. Finally, click the **Start** button. The FPGA will be programmed with your design. Toggle the switches and verify that the circuit is performing correct 2-bit addition with carry-in. Note that there is a switch on the bottom left of the FPGA board marked with “Run” and “Program”. Please leave that switch at “Run” even when you are programming the FPGA.

Instructions for Performing Experiments Involving SystemVerilog in the Lab


Creating a Project

- Start Quartus Prime.
- From the **File** menu, select **New Project Wizard**.
- Enter a location, name, and top-level entity for the project. Create your project on the W: drive.
- Add any files that you have brought with you to the lab (you may need to copy them to the directory you specified above first).
- Select **Cyclone IV** for Device Family
- Select **EP4CE115F29C7** for Device
- Select **ModelSim-Altera** as the simulation tool name, and **SystemVerilog HDL** as the simulation format. Keep clicking **Next** until you get to **Finish**.
- You can see in the **Project Navigator** window at the left of the screen that a project has been created.

Adding Files to a Project

- If you didn't add your files in the step above, or if you wish to add additional files, right click on **Cyclone IV: EP4CE115F29C7** in the **Project Navigator** window on the left of the screen and click **Settings**. Select **Files** under **Category**. You can add files by entering their names directly or by finding them with the file browser; the **add all** button will add all vhd files in the project directory. Get the testbench files (*.vwf) from your TA and add to the project the same way as above. Additionally, in the **Settings** window, select **Simulator** in the **Category** box, and set the simulation **Time scale** to "1ns".


Analyzing, Synthesizing, and Implementing the Design

- Click the **Start Compilation** () button to begin the compilation process. If there are no errors, you will see each of the steps complete to 100%. If there are errors, you can view them on the **error** tab of the **Messages** window.
- You will need to fix any errors in your design before continuing. Additionally, you may receive warnings from the various steps. You should correct all warnings from the **Analysis & Synthesis** step. Warnings from the other steps may be acceptable; check with your TA.

Viewing the Synthesized Design



- Once the design is fully compiled, you can view the synthesized circuit in **Tools > Netlist Viewers > RTL Viewer** in an interactive GUI. If your design contains a state machine, and if your state machine is well structured, Quartus Prime should be able to extract the state machine for you as well. It can be viewed in **Tools > Netlist Viewers > State Machine Viewer**. This may help you generate the block diagrams required in lab reports.

Simulating the Design

- Once you are ready to run simulations on your design, click the **RTL Simulation** () button. This will bring up the **ModelSim-Altera** simulator, which will automatically generate the simulation netlist.


Setting pin mappings and programming the DE2 Board

Once your design simulates correctly, you can download the design bit-stream onto the DE2 board and test it.

- First you have to provide pin assignments for inputs and outputs. Click the ***Pin Planner*** button ()
- This will open the Pin Planner Editor. The list of circuit input and output pins should have been automatically listed at the bottom portion of the window. The list of assignments can be found in the lab manual entry for the current lab. Type in these assignments. (If you prefer, you can double-click the fields in the editor and select signals and pins from a dropdown menu.)
- Once you have entered the pin assignments for all ports, save the assignment editor view and recompile the design ()

Downloading the design to the DE2 Board

The Programmer sends the completed bitstream to the FPGA.

- Connect the programming cable to the computer's USB port and the board's ***USB Blaster*** connector.
- Power on the dev board.
- Click the ***Programmer*** button () to open the programmer. You should see a ***.sof*** file bearing the same name as the top-level entity in the window. If it does not appear automatically, you can add it from the "output files" folder.
- If the box next to the ***Hardware Setup...*** button reads ***No Hardware***, click the ***Hardware Setup...*** button and select the ***USB-Blaster*** option.
- If ***Select Device*** is prompt, select ***EP4CE115***.
- Check the box under ***Program/Configure*** to select the programming file.
- Make sure the slider switch next to the LCD display on the DE2 board is set to ***Run***, not ***Prog***.
- Click ***Start***. The FPGA will be programmed with your design.
- Once the programming process completes, test your design for functionality.
- If your design does not work or exhibits bugs, alter your design, recompile, and reprogram the FPGA. If you cannot easily determine how to fix your design, design a simulation to capture the incorrect behavior to better understand what is going wrong.
- **If the FPGA does not program correctly, ensure the RUN/PROG switch on the DE2-115 board is set to RUN**

I/O Pins available on the DE2-115 Development Board

Pins used in the lab exercises are listed in the manual description for each experiment. A full listing of the devices and associated pins on the DE2 development board can be found in the DE2 User Manual, which can be found on the DE2 System CD in the file `\DE2_115_user_manual\DE2-115_UserManual.pdf`. The contents of the DE2-115 System CD can be downloaded at <http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html> if you do not have the CD handy.

A spreadsheet containing all the pins listed here can also be found on the DE2 System CD, in the file `\DE2_115_lab_exercises\DE2_115_pin_assignments.csv`. You may use this file to easily copy and paste groups of assignments into the Assignment Editor.

Quartus Tutorial Exercise – Bit-Serial Processor in SystemVerilog

The 8-bit serial processor will be very similar to the design you have done for Experiment 3. The only difference is that you will now use two 8-bit shift registers to store the data inputs and the result. You will first load the registers with the values that you want to be used as operands. Then you will specify the operation that you want to perform on the data and specify the register that should hold the result. The result will be computed using bit-wise operators. Every cycle during computation, the least significant bits from each register will be shifted out, desired logic operation will be performed on the two bits, and the result bits will be shifted into the registers as the most significant bits. After repeating this procedure eight times, the specified register will contain the 8-bit result.

Task: Download the 4-bit module files from the course website (under Lab 4). Use it to build an 8-bit serial processor project using the provided modules by creating a project, adding the provided files to the project and test the circuit operation using the included testbench. Follow the SystemVerilog tutorial ISV and the instructions for testbench (**IQT. 23-30**) to complete the exercise. Include a copy of the schematic block diagram and the simulation waveform (with annotations) in your Lab 4 lab report.

Your circuit should have the following inputs and outputs (once it has been extended to 8-bits):

Inputs

Clk, Reset, Execute, LoadA, LoadB – logic
 Din – logic [7:0]
 F – logic [2:0]
 R – logic [1:0]

Descriptions of the inputs:

Clk: Clock signal for the processor
 Reset: Reset signal to initialize the controller to the start state.
 Execute: Signal that triggers the execution of a computation cycle.
 LoadA: Signal that loads Register A with the data specified by the input switches (Din<7:0>).
 LoadB: Signal that loads Register B with the data specified by the input switches (Din<7:0>).
 Din<7: 0>: Input switches for the input data
 F<2:0>: Function select
 R<1:0>: Router select

Outputs

Aval, Bval – logic [7:0]
 AhexU, AhexL, BhexU, BhexL – logic [6:0]
 LED – logic [3:0]

Descriptions of the outputs:

Aval<7:0>, Bval<7:0>: Binary values in registers A, B
 AhexU<6:0>, AhexL<6:0>: 7-segment display driver signals (Upper and Lower nibbles) for register A
 BhexU<6:0>, BhexL<6:0>: 7-segment display driver signals (Upper and Lower nibbles) for register B
 LED<3:0>: Concatenated values of Execute, LoadA, LoadB, & Reset signals, to be displayed on LEDs.

Note: For Experiments 4-9, you should use the same input/output port names as provided for each experiment in this manual.

The block diagram for the circuit is shown in Figure 14. The register unit contains two 8-bit registers, Register A and Register B. The computation unit performs the desired logical operation based on the function select ($F<2:0>$). The routing unit selects the input bits to Register A and Register B after the computation. The control unit provides control signals (e.g. load, shift) to the register unit.

Control Unit

The control unit will accept Clock, LoadA, LoadB, Execute, and Reset as inputs. It will output the control signals to the register unit to instruct the registers when to load the values from the switches and when to shift the register values. The Reset signal, when set to high, will put the controller in the reset/start state. A 0→1 transition on the Execute signal will trigger the execution of a computation cycle. Once the computation cycle has started, the controller should ignore the Execute signal until the appropriate logical operation is performed on all bits and the registers contain appropriate 8-bit values. That is, during execution, the Execute signal is a ‘Don’t Care’. The processor should halt at the end of a computation cycle until the Execute signal is set to low. Another computation cycle can then begin when the Execute signal again switches from low to high. Load A and Load B should only be allowed to take effect in the reset/start state and should be ignored during the execution of a computation cycle.

Register Unit

The register unit will contain two 8-bit registers, Register A and Register B. The control signals to parallel load the values of the data inputs ($D_{in} < 7 : 0 >$) to one or both of the registers and to shift in the result bits from the routing unit will come from the control unit. During a computation cycle each register will shift-out one bit at a time to the computation unit as the result bit from the routing unit shifts in. The 8-bit values of the registers will also be provided as outputs that the user can observe.

Computation Unit

The computation unit will perform a logical operation based on the function select ($F < 2 : 0 >$) on the operands provided by the register unit. The computation unit will output three one-bit values to the routing unit – A, B, and F(A, B). Table 1 provides the functions associated with the value of $F < 2 : 0 >$ and the values that should be transmitted to the register unit from the routing unit based on $R < 1 : 0 >$.

Routing Unit

The routing unit will accept A, B, and F(A, B) as inputs and will output A' (shift-in to Register A, newA) and B' (shift-in to Register B, newB) based on $R < 1 : 0 >$ as shown in Table 1.

HexDriver Units

Each HexDriver unit accepts one 4-bit hex digit as input and outputs the 7 bits necessary to correctly display this digit on a 7-segment display. You will instance 4 HexDriver units, because you need to display 16 bits of data.

Pin Assignment Table

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Execute	PIN_R24	On-Board Push Button (KEY3)
LoadA	PIN_N21	On-Board Push Button (KEY2)
LoadB	PIN_M21	On-Board Push Button (KEY1)
Reset	PIN_M23	On-Board Push Button (KEY0)
Din[0]	PIN_AB28	On-board slider switch (SW0)
Din[1]	PIN_AC28	On-board slider switch (SW1)
Din[2]	PIN_AC27	On-board slider switch (SW2)
Din[3]	PIN_AD27	On-board slider switch (SW3)

Din[4]	PIN_AB27	On-board slider switch (SW4)
Din[5]	PIN_AC26	On-board slider switch (SW5)
Din[6]	PIN_AD26	On-board slider switch (SW6)
Din[7]	PIN_AB26	On-board slider switch (SW7)
R[0]	PIN_AA24	On-board slider switch (SW13)
R[1]	PIN_AA23	On-board slider switch (SW14)
F[0]	PIN_AA22	On-board slider switch (SW15)
F[1]	PIN_Y24	On-board slider switch (SW16)
F[2]	PIN_Y23	On-board slider switch (SW17)
Aval[0]	PIN_J15	On-Board LED (LEDR10)
Aval[1]	PIN_H16	On-Board LED (LEDR11)
Aval[2]	PIN_J16	On-Board LED (LEDR12)
Aval[3]	PIN_H17	On-Board LED (LEDR13)
Aval[4]	PIN_F15	On-Board LED (LEDR14)
Aval[5]	PIN_G15	On-Board LED (LEDR15)
Aval[6]	PIN_G16	On-Board LED (LEDR16)
Aval[7]	PIN_H15	On-Board LED (LEDR17)
Bval[0]	PIN_G19	On-Board LED (LEDR0)
Bval[1]	PIN_F19	On-Board LED (LEDR1)
Bval[2]	PIN_E19	On-Board LED (LEDR2)
Bval[3]	PIN_F21	On-Board LED (LEDR3)
Bval[4]	PIN_F18	On-Board LED (LEDR4)
Bval[5]	PIN_E18	On-Board LED (LEDR5)
Bval[6]	PIN_J19	On-Board LED (LEDR6)
Bval[7]	PIN_H19	On-Board LED (LEDR7)
LED[0]	PIN_E21	On-Board LED (LEDG0)
LED[1]	PIN_E22	On-Board LED (LEDG1)
LED[2]	PIN_E25	On-Board LED (LEDG2)
LED[3]	PIN_E24	On-Board LED (LEDG3)
AhexL[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
AhexL[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
AhexL[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
AhexL[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
AhexL[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
AhexL[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
AhexL[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
AhexU[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
AhexU[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
AhexU[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
AhexU[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
AhexU[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
AhexU[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
AhexU[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
BhexL[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
BhexL[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
BhexL[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
BhexL[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
BhexL[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
BhexL[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
BhexL[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
BhexU[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
BhexU[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
BhexU[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])

BhexU[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
BhexU[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
BhexU[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
BhexU[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])

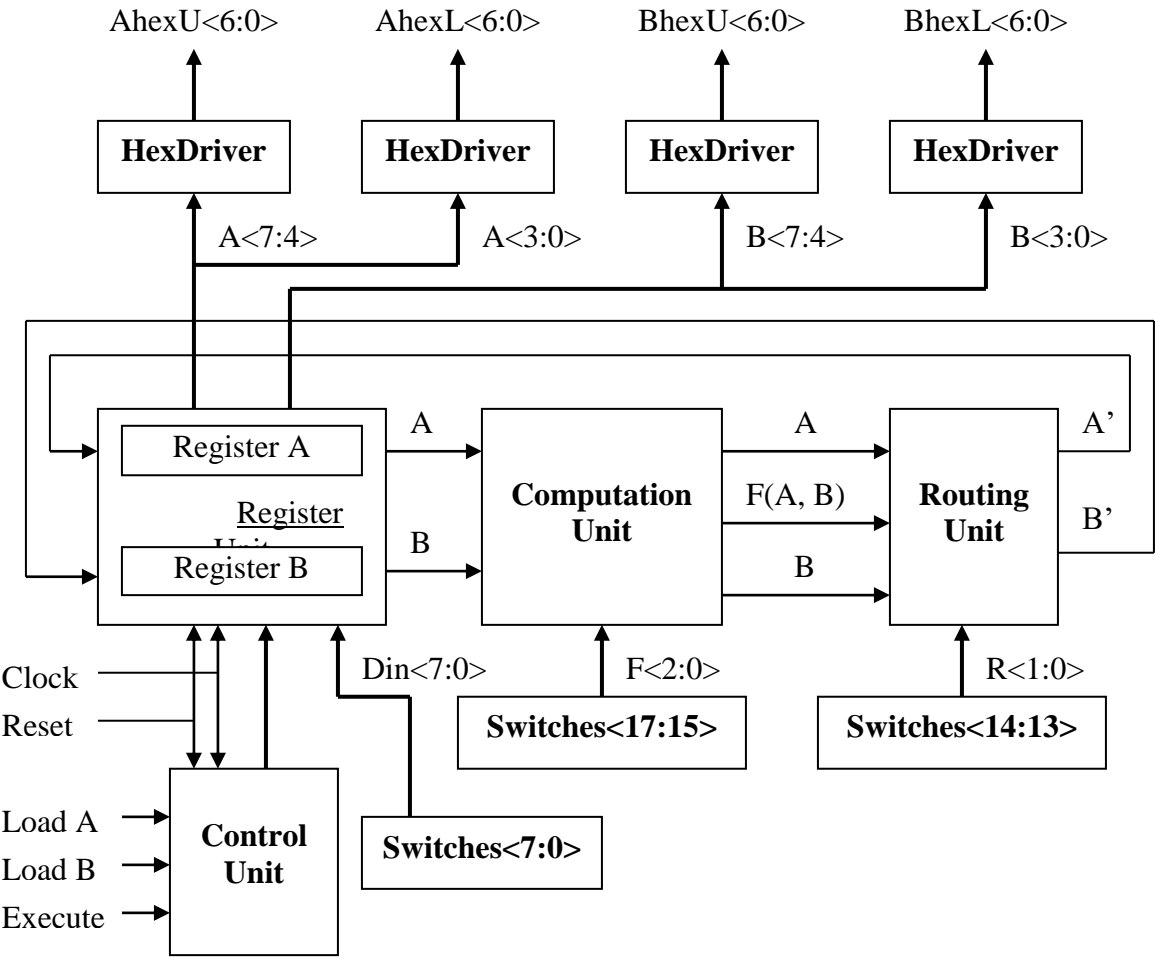


Figure 14: Block Diagram

Table 2. Function Table

Function Selection Inputs			Computation Unit Output	Routing Selection		Router Output	
F<2>	F<1>	F<0>	f(A, B)	R<1>	R<0>	A'	B'
0	0	0	A AND B	0	0	A	B
0	0	1	A OR B	0	1	A	F
0	1	0	A XOR B	1	0	F	B
0	1	1	1111	1	1	B	A
1	0	0	A NAND B				
1	0	1	A NOR B				
1	1	0	A XNOR B				
1	1	1	0000				

Design Resources and Statistics

Altera FPGAs use the term Logic Element (LE) to describe a functional block that contains 1 look-up table (LUT), 1 register (FF), and some additional circuitry. In FPGAs, all the combinational logic is implemented in LUTs. To find out the usage of LUTs after compilation, go to the **Compilation Report** tab. On the left, go to **Fitter > Resource Section > Resource Usage Summary**. The summary should look like what is shown in Figure 1.

Fitter Resource Usage Summary		
	Resource	Usage
1	▲ Total logic elements	2,661 / 33,216 (8 %)
1	-- Combinational with no register	898
2	-- Register only	432
3	-- Combinational with a register	1331
2		
3	▲ Logic element usage by number of LUT inputs	
1	-- 4 input functions	1128
2	-- 3 input functions	709
3	-- <=2 input functions	392
4	-- Register only	432
4		
5	▲ Logic elements by mode	
1	-- normal mode	2069
2	-- arithmetic mode	160
6		
7	▲ Total registers*	1,831 / 34,593 (5 %)
1	-- Dedicated logic registers	1,763 / 33,216 (5 %)
2	-- I/O registers	68 / 1,377 (5 %)
8		
9	Total LABs: partially or completely used	198 / 2,076 (10 %)
10	Virtual pins	0
11	▲ I/O pins	47 / 475 (10 %)
1	-- Clock pins	1 / 8 (13 %)
12		
13	Global signals	8
14	M4Ks	14 / 105 (13 %)
15	Total block memory bits	44,032 / 483,840 (9 %)
16	Total block memory implementation bits	64,512 / 483,840 (13 %)
17	Embedded Multiplier 9-bit elements	0 / 70 (0 %)
18	PLLs	1 / 4 (25 %)
19	Global clocks	8 / 16 (50 %)
20	JTAGs	1 / 1 (100 %)
21	ASMI blocks	0 / 1 (0 %)

Figure 1

Here, the total number of LEs is reported. LEs are divided into 3 categories: combinational with no register, register only, and combination with register. Add the numbers of *combinational with no register* and *combinational with register* to get the total number of LUTs used. Next, the numbers of LEs categorized by the number of LUT inputs are reported. Sum up the numbers of *4 input function*, *3 input funtions*, and *<= 2 input functions*. Note that this number is equal to the number of LUTs we just computed. Further below, we can find a report of total number of registers (flip-flips). Note that the total number of dedicated logic registers is equal to the numbers of LEs that are *combinational with register* and *registers only*. Further below, we can find a report of total block memory implementation bits. There is an on-chip RAM block on the Cyclone II FPGA, sizing 483,840 bits, which we call block memory (BRAM). BRAM is not used in most of the

SystemVerilog labs, but in the SoC lab, it is required for to store the software for the Nios II processor. The reported number gives you the usage of BRAM in the design.

If the design utilizes Altera's built-in digital signal processing (DSP) blocks, a separate report can be found at **Fitter -> Resource Section -> DSP Block Usage Summary Report**. If no such report is there, it means that no DSP blocks are used.

For the timing analysis, the TimeQuest Timing Analyzer is used in Quartus Prime. To perform a detailed analysis on the operating frequency, input and output timing constraints of the circuit needs to be provided using the Synopsys Design Constraints (.sdc) format. These constraints usually arise when the circuit designed for the FPGA is not a standalone circuit, but rather it is merely a part of a larger system. The other parts of the system often have their own operating constraints, which eventually become the input and output constraints for the FPGA circuit. However, since the FPGA designs in our labs are standalone systems with manual inputs, it is not easy to come up with a SDC file with detailed timing constraints. However, simple timing analysis can still be performed by simply defining the system clock in the SDC file. This is because that the maximum possible operating frequency of the design can still be deduced from the design itself disregard of any information from the inputs and outputs. To define the system clock in the SDC file, simply save the following script in a .sdc file for your project:

```
#####
# Create Clock (where 'Clk' is the user-defined system clock name)
#####
create_clock -name {Clk} -period 20ns -waveform {0.000 5.000} [get_ports {Clk}]

# Constrain the input I/O path
set_input_delay -clock {Clk} -max 3 [all_inputs]
set_input_delay -clock {Clk} -min 2 [all_inputs]

# Constrain the output I/O path
set_output_delay -clock {Clk} 2 [all_outputs]
```

The maximum allowed frequency for the system clock can be found under **TimeQuest Timing Analyzer -> Fmax Summary**.

Note: It does not matter what name you give to the .sdc file, as long as it's declared as a .sdc file instead of a regular .sv file. With the inclusion of the .sdc file, the 'TimeQuest Timing Analyzer' section in the 'Compilation Report' after the full compilation will show constrained result for 'Slow 1200mV 85C Model', 'Slow 1200mV 0C Model', and 'Fast 1200mV 0C Model'. You can pick the

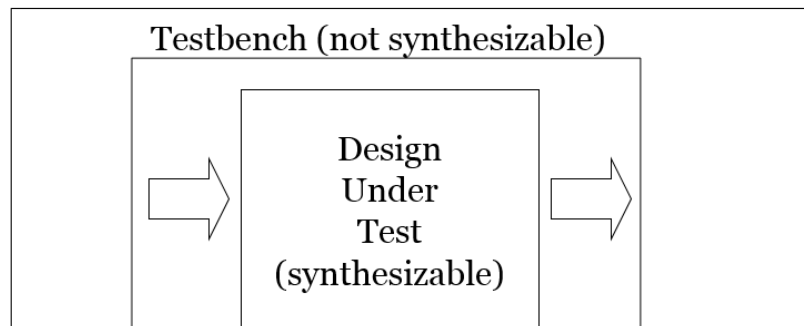
frequency from the 'Slow 1200mV 85C Model', since high temperature makes the chips slower, and 85C is the upper operating temperature limit for commercial-grade electrical devices. If you do not include the .sdc file in your project, TimeQuest Timing Analyzer will still give you some unconstrained frequencies in the same report (the three subsections will be shown in red instead of black). Notice that the frequency now is much higher. This is because in the .sdc constraint file we use the 50MHz clock, but without the file Quartus defaults the clock to be 1000MHz for all inputs. You can see the clock information under 'Clocks' subsection in the 'TimeQuest Timing Analyzer' report.

For the Power consumption analysis, the PowerPlay Power Analyzer is used in Quartus Prime. To activate the PowerPlay Power Analyzer during project compilation, go to **Assignments -> Settings**, and find the **PowerPlay Power Analyzer Settings** under **Category**. Check the box labeled **Run PowerPlay Power Analyzer during compilation**. The power analysis report can be found in **PowerPlay Power Analyzer -> Summary** in the **Compilation Report** tab.

Instructions for Testbench for the Bit-Serial Logic Processor

These instructions are a tutorial for creating a testbench that generates simulation waveforms and results for the bit-serial processor in the Quartus Prime environment, with the use of the ModelSim-Altera simulator. It is **required in the demo**.

A testbench envelopes the design under test (DUT), provides it with test stimuli, collects the results, and analyzes it, as illustrated below. In the bit-serial processor, the DUT is the processor module. We will use the testbench to provide some inputs for testing. The results can be viewed as waveforms in ModelSim-Altera.



Note that we have so far emphasized on the synthesizability of SystemVerilog coding because the code in concern describes the design itself. On the other hand, a testbench is used in simulations and is not required to be synthesizable on hardware. Therefore, some syntax used in the testbench is not synthesizable. Pay attention to the differences and avoid using syntax that are not synthesizable in the designs!

Preliminary Work:

Build the bit-serial processor project and include all the provided code. For the “Processor (Main Module)” part, either code it or use the schematic editor to complete it as taught in the previous section. Make sure the top-level “Processor” module uses all the port names given in the lab manual because they will be used in the testbench.

Writing the Testbench:

The following is the contents of *testbench_8.sv* in the provided code:

```

module testbench();

timeunit 10ns;    // Half clock cycle at 50 MHz
                  // This is the amount of time represented by #1
timeprecision 1ns;

// These signals are internal because the processor will be
// instantiated as a submodule in testbench.
logic Clk = 0;
logic Reset, LoadA, LoadB, Execute;
logic [7:0] Din;

```

IQT.22

```
logic [2:0] F;
logic [1:0] R;
logic [3:0] LED;
logic [7:0] Aval,
            Bval;
logic [6:0] AhexL,
            AhexU,
            BhexL,
            BhexU;

// To store expected results
logic [7:0] ans_1a, ans_2b;

// A counter to count the instances where simulation results
// do no match with expected results
integer ErrorCnt = 0;

// Instantiating the DUT
// Make sure the module and signal names match with those in your design
processor processor0(.);

// Toggle the clock
// #1 means wait for a delay of 1 timeunit
always begin : CLOCK_GENERATION
    #1 Clk = ~Clk;
end

initial begin: CLOCK_INITIALIZATION
    Clk = 0;
end

// Testing begins here
// The initial block is not synthesizable
// Everything happens sequentially inside an initial block
// as in a software program
initial begin: TEST_VECTORS
    Reset = 0;          // Toggle Rest
    LoadA = 1;
    LoadB = 1;
    Execute = 1;
    Din = 8'h33;        // Specify Din, F, and R
    F = 3'b010;
    R = 2'b10;

    #2 Reset = 1;

    #2 LoadA = 0;      // Toggle LoadA
    #2 LoadA = 1;

    #2 LoadB = 0;      // Toggle LoadB
    Din = 8'h55;        // Change Din
    #2 LoadB = 1;
```

```

Din = 8'h00;    // Change Din again

#2 Execute = 0; // Toggle Execute

#22 Execute = 1;
ans_1a = (8'h33 ^ 8'h55); // Expected result of 1st cycle
// Aval is expected to be 8'h33 XOR 8'h55
// Bval is expected to be the original 8'h55
if (Aval != ans_1a)
    ErrorCnt++;
if (Bval != 8'h55)
    ErrorCnt++;
F = 3'b110;    // Change F and R
R = 2'b01;

#2 Execute = 0; // Toggle Execute
#2 Execute = 1;

#22 Execute = 0;
// Aval is expected to stay the same
// Bval is expected to be the answer of 1st cycle XNOR 8'h55
if (Aval != ans_1a)
    ErrorCnt++;
ans_2b = ~(ans_1a ^ 8'h55); // Expected result of 2nd cycle
if (Bval != ans_2b)
    ErrorCnt++;
R = 2'b11;
#2 Execute = 1;

// Aval and Bval are expected to swap
#22 if (Aval != ans_2b)
    ErrorCnt++;
    if (Bval != ans_1a)
        ErrorCnt++;

if (ErrorCnt == 0)
    $display("Success!"); // Command line output in ModelSim
else
    $display("%d error(s) detected. Try again!", ErrorCnt);

end

endmodule

```

Configurations:

Go to the menu *Assignments > Settings*. Click on *EDA Tool Settings* on the left. Make sure the simulation tool is set to *ModelSim-Altera*, and the format is *SystemVerilog HDL*. Figure 1 shows the dialog box. If the project was created according to the lab manual, these settings should already be there.

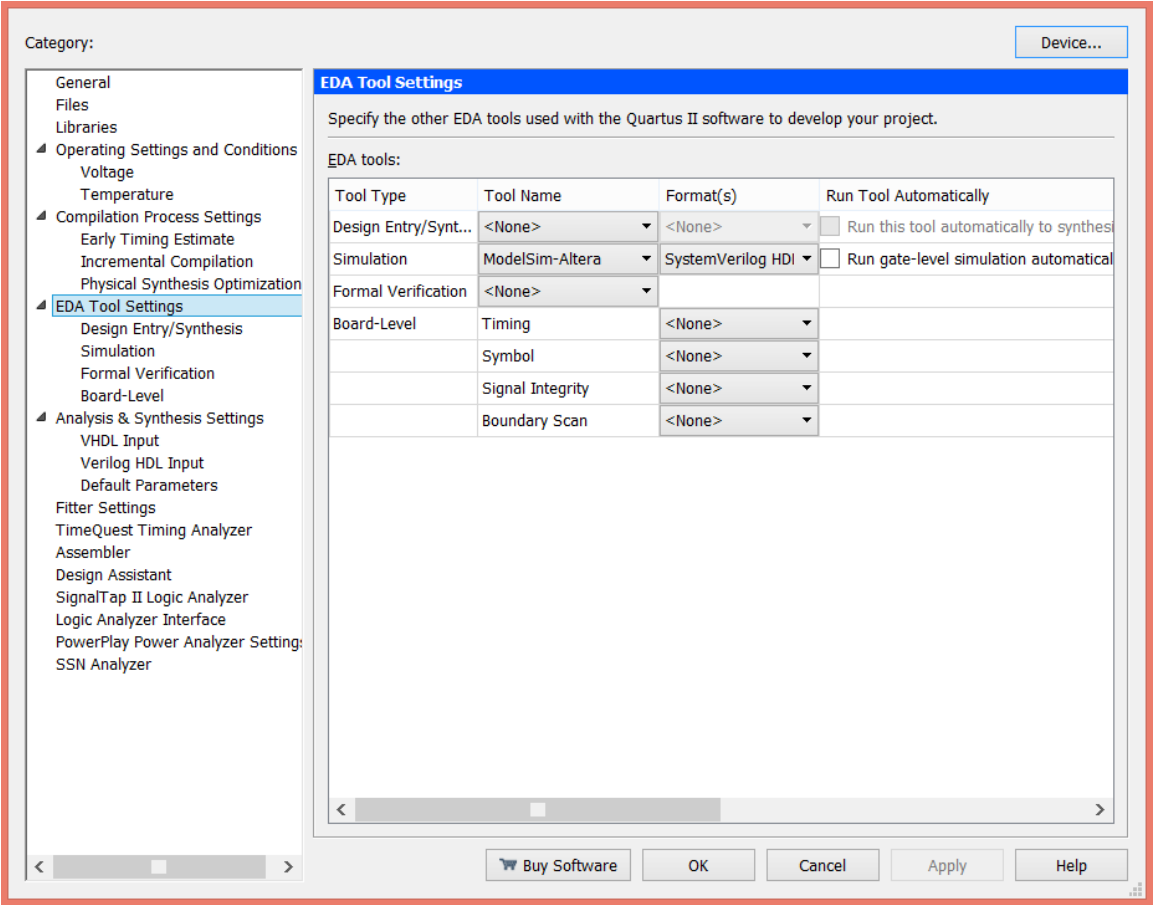


Figure 1

Next, click on **Simulation** under **EDA Tool Settings**. Select *ModelSim-Altera* for *Tool name*, *SystemVerilog HDL* for *Format of output netlist*, and “*simulation/modelsim*” for *Output directory*. Figure 2 shows the dialog box.

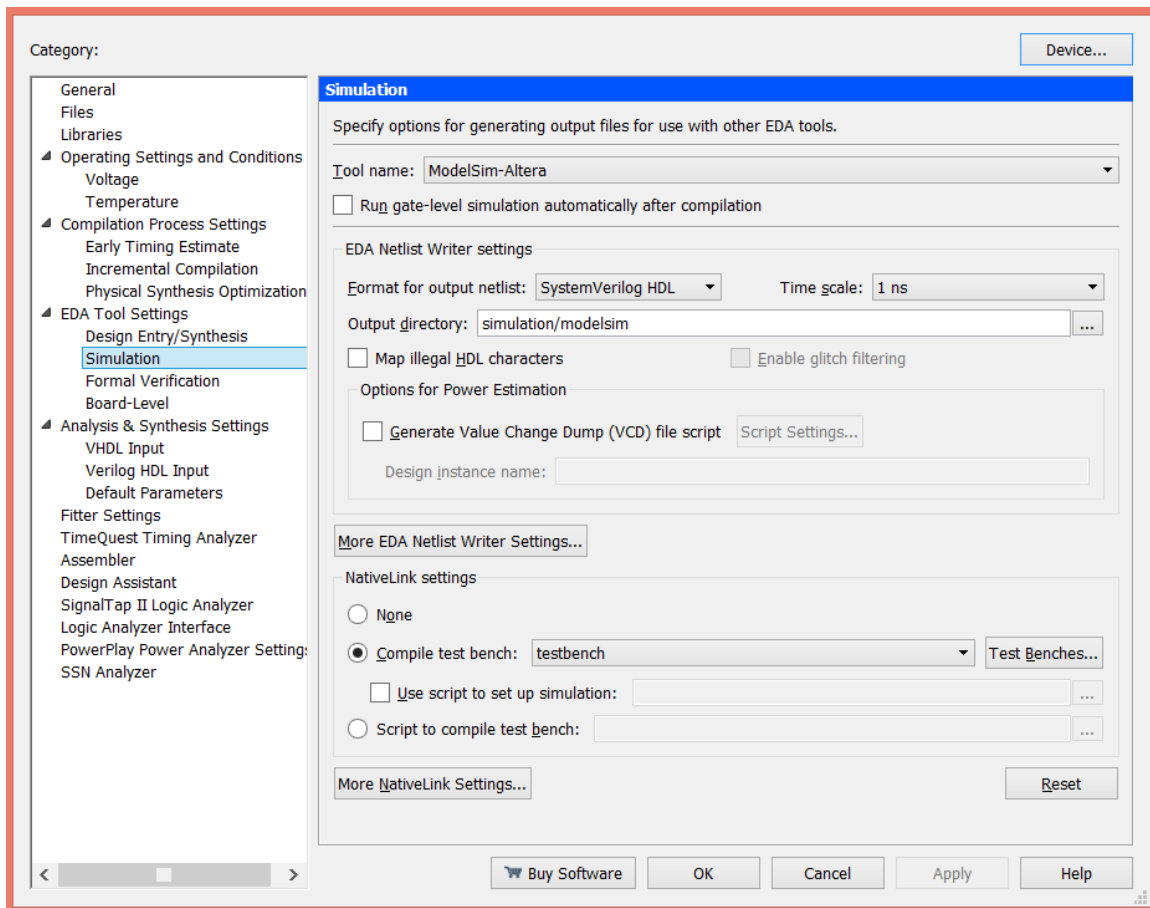


Figure 2

Next, click on the **Test Benches...** button in **NativeLink settings**, on the bottom of the dialog box shown in Figure 2. A window should pop up. Select **New...**, and a window shown in Figure 3 should pop up. Enter *testbench* for *Test bench name* and *Top level module in test bench*, *1000 ns* for *simulation end time*, and then click on the ... button and add *testbench.sv* into *test bench and simulation files*.

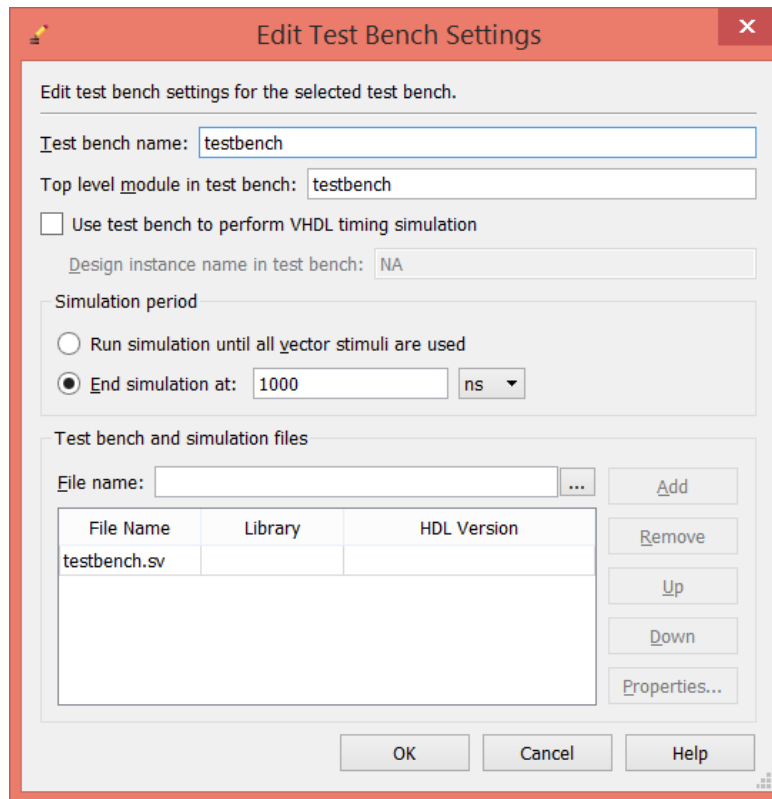





Figure 3

Now we're ready to run the testbench. Go back to the main Quartus Prime window, compile the project by clicking **Start Analysis & Synthesis**, and then select **Tools > Run Simulation Tool > RTL simulation** or simply click on the **RTL Simulation button** (). The results will be shown in ModelSim-Altera as in Figure 4. You may need to right click on the waveform and zoom to a proper scale for better reading. Press 'f' to see the entire waveform. Use Ctrl+Scroll Wheel to zoom in/out the waveform.

Note: By default the buses are displayed in binary value. To change the display format, right click on the bus name and select **Radix -> <format>**. For example, if you wish to see the decimal value, you should select **Radix -> Decimal**.

To run a functional simulation of your design, click the **Run-All button** () in the toolbar. If you wish to modify any portion of the waveform and re-simulate, you should first click the **Restart button** () to erase the existing outputs.

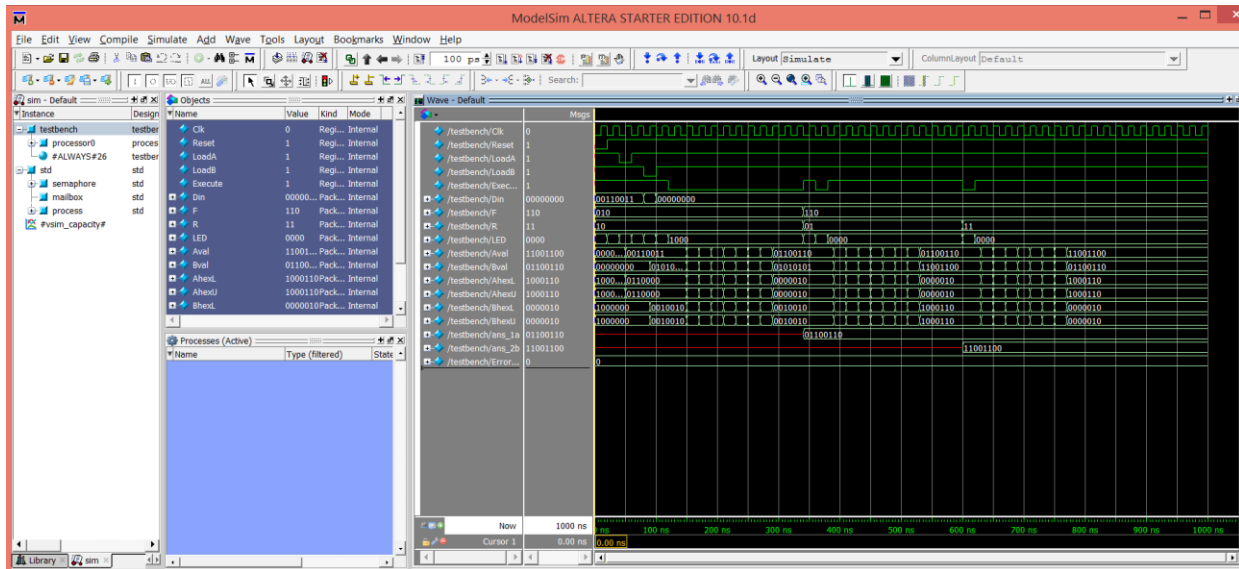


Figure 4

Note that the red lines indicating that ans_1a and ans_2b are unknown values in the beginning are expected because these two variables would not be computed until the middle of the TEST_VECTORS initial block.

Monitoring Internal Signals:

In ModelSim, we also have the ability to monitor internal signals in the hierarchical design in addition to the ones specified in the testbench. To do this, navigate the design hierarchy on the left, select the signal to be observed, and then right click and choose **Add Wave**. The selected signal will be added into the simulation waveform window. Figure 5 shows an example of adding the state variable in the controller.

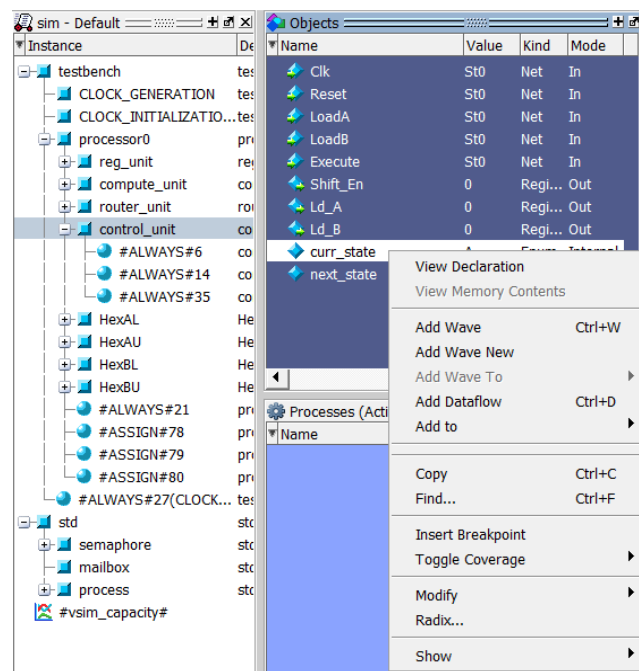


Figure 5

After all signals of your choice are added, go to the command line below and type in:

```
restart -f
```

This will reset the waveform window such that the testbench starts from the beginning again.

```
log -r *
```

This will tell ModelSim to record all signals in the circuit recursively.

```
run 1000ns
```

This will run the testbench for 1000ns. You may change the simulation time if needed. The waveforms of the selected internal signals should be available for view now.

Troubleshooting:

If a certain part of your design is not showing up in the design hierarchy window, it's usually because the design file fails to compile successfully in ModelSim. Look into the error messages in the *Transcript* to find out what went wrong.

ModelSim has its own internal compiler that's different from Quartus Prime's. The accepted syntax and the ways they handle certain statements are slightly different. Therefore, the compiled circuits may be slightly different as well, especially if a good coding style is not employed. Also, ideal circuits with no delays are assumed in simulations, whereas delays do exist in real world. If timing is not handled safely, the values read in a specific clock cycle may be one cycle off. These two reasons may lead to different results in simulation from those on-board.

If you have output signals computed in *always_ff* blocks, declare them as *output logic* instead of *output*. The ModelSim compiler tends to complain about this.