

CME341 Laboratory Manual

Created by Eric Salt

Created June 30, 2012
Latest Version: September 27, 2023

Revised Nov. 30, 2012

Revised Dec. 13, 2012 (to end of Lab 2)

Revised Dec. 18, 2012 (to end of Lab 3)

Revised Jan. 13, 2013 (to end of Lab 4)

Revised Jan. 24, 2013 (to end of Lab 5)

Revised Feb. 1, 2013 (added .csv format for Quartus 12.1)

Revised Feb. 20, 2013 (added labs 6 to 10)

Revised April 23, 2013 (major revisions described below)

The April 23 revisions were made in response to the feedback provided by the EE431 and EE831 classes of 2012-2013

1. Many typographical errors were corrected and several explanations were expanded/improved.
2. Time estimates for the labs were changed to be the 75 percentile of the times reported by the 2012-2013 EE431 class.
3. Labs 6 and 10 were made optional to reduce the total in-Lab time to 36 hours.
4. Lab 5 was divided into parts A, B and C and treated as two separate Labs.
5. Labs 7 and 8 were merged into one Lab.
6. The labs were re-ordered to appear in the sequence Lab 1, Lab 2, Lab 3, Lab 4, Lab 7&8, Lab 9, Lab 5, Lab 6 (optional), Lab 10 (optional)
7. Labs 7&8 were combined and renumbered Labs 5A and 5B; Lab 9 was renumbered Lab 6; Lab 5 was broken into sections and renumbered Lab 7A, 7B and 7C; Lab 6 was renumbered Lab 8 and made optional, Finally Lab 10 was renumbered Lab 9 and made optional.

Revised May 7, 2013 (updated the table on page 4 that gives the time estimates for each of the labs)

Revised May 13, 2013 (added the learning outcomes to the objective section in each of the labs)

Revised Aug. 13, 2013 (minor changes suggested by TA Cinatti Loi)

Revised Sept. 14, 2013 (significant revisions described below)

1. Corrected Typographical and grammatical errors found by TA Andy Fontaine.
2. Removed a few of the extra labs to reduce the length of the document.
3. Incorporated instructions for the DE2-115 (Cyclone IV E FPGA). This was done to allow the students to use either the DE2 board (Cyclone II) or the DE2-115 board (Cyclone IV E).

Revised Sept. 16, 2013 (noted that configuration device for DE2-115 is EPCS64)

Revised Sept. 17, 2013 (updated warning 169174 to exclude the Cyclone IV E)

Revised Sept. 20, 2013 (added hex display driver Figure for lab 1 and removed a reference to Questa Intel FPGA)

Revised Sept. 25, 2013 (revised lab 2 to force words running off the page to appear on the next line.
Began making revisions for Lab 3 to remove all references to modelsim and add instructions and pin-outs for the DE2-115 board.)

Revised Sept. 27, 2013 (Updated Lab 3 to include pin outs for DE2-115 board)

Revised Oct. 12, 2013 (Added some clarification to Lab 3 and added extra tasks to Lab 4 as described below)

1. A exercise showing how to use ‘‘design entry’’ signal names when configuring the trigger table in SignalTap.
2. An exercise that shows how to recover from pin assignment errors.

Revised July 21, 2014 (Reworded parts of Lab 3 and added screen captures to help explain the procedure. Also corrected typos throughout the document.)

Revised Sept. 26, 2014 (changed trigger transition condition from ‘‘from EH to FH’’ to ‘‘from FH to OH’’ in the very last part of Lab 3.)

Revised Nov. 5, 2014 ()

Revised Jan. 5, 2016 (Removed the requirement for the students to complete an evaluation sheet for each of the labs.)

Revised Sept. 14, 2016 (Clarified how and where to connect the oscilloscope in Lab 1.)

Revised Sept. 15, 2016 (Extensive revision that reduced the document in size by over 10 pages.
References to the Cyclone II FPGA and DE2 board were largely removed as that FPGA and board are

- no longer used in our labs.)
- Revised Nov. 1, 2016 (Rewrote several paragraphs in Lab 7a) and 7b) to make it more understandable.)
- Revised Nov. 2, 2016 (Rewrote parts of Lab 7c) to make it more understandable.)
- Revised Sept. 22, 2017 (Due to Rory Gowen - Revised the part of Lab 3 that generates the SDC file to make the instructions consistent with Quartus Prime.)
- Revised Sept. 5, 2020 (By Brian Berscheid
- Minor updates to support SystemVerilog source files
 - Minor typographical fixes)
- Revised Sept 26, 2023 (By Eric Salt
- major revision to SignalTap part of Lab 3
 - updated all the screen captures to Quartus V22.1
 - removed all material that would not be on a lab exam
 - added an exercise where the students have to set up SignalTap and trigger on a waveform)

Contents

1	Evaluation Related Material	6
1.1	Time Schedule	6
2	Objectives and Learning Outcomes	7
3	LAB 1: Getting Started	11
3.1	Synthesizing a combinational logic circuit	11
3.1.1	Active Low Set Clear Latch	11
3.1.2	Verifying the Slide Switches	17
3.1.3	Active High Set Clear Latch	17
3.1.4	Programming the Flash Memory on the DE2-115 Board	18
3.1.5	Re-Programming the Flash Memory on the DE2-115 Board with the original test circuit	20
3.1.6	Hex Display Driver	20
4	LAB 2: Designing a Binary to Binary-Coded-Decimal Converter	22
5	LAB 3: Sequential Logic	28
5.1	Synthesizing a circuit that uses a clock	28
5.2	Loading the circuit together with SignalTap (SignalTap is an Embedded Logic Analyser provided by Intel) and then Using SignalTap	35
5.2.1	Using SignalTab to debug prototype counter	36
5.2.2	Controlled Triggering in Sequential Mode	47
5.3	Exercise in Using SignalTap	48
6	LAB 4: Designing a B-to-BCD Converter with Sequential Logic	51
7	LAB 5: Controller for Traffic Lights	59
7.1	LAB 5A: A Simple Traffic Light Controller	60
7.2	LAB 5B: Adding a Left Turn Arrow to the Traffic Lights	64
8	LAB 6: Controller for Traffic and Pedestrian Lights	66
9	LAB 7: Electronic Card Lock for Hotel Room Doors	70
9.1	PART A: Linear Feedback Shift Registers	72
9.2	PART B: Emulation of Card Reader	75
9.3	PART C: Electronic Card Lock	78
10	OPTIONAL LAB 8: Arithmetic in Verilog	83
10.1	Adders	83
10.2	Multipliers	84
10.3	Comparators	84
10.4	Circuit Sizes	85

<i>LIST OF FIGURES</i>	5
------------------------	---

11 OPTIONAL LAB 9: Using Phase Lock Loops and SignalProbe	86
11.1 PART A - Post Compile Routing of a Signal to a Pin	86
11.2 PART B: Using a PLL to create clocks	88
12 APPENDIX A - Cyclone IV E FPGA to DE2-115 board pin Mapping	93

List of Figures

1	The DE2-115 board	9
2	Basic Layout for the Altera DE2 board. The DE2-115 board has the same basic layout, except with the Cyclone IV E FPGA in place of the Cyclone II	10
3	The display format for a hexadecimal number	21
4	Block diagram for a Binary to Binary-Coded-Decimal Converter	23
5	The content of the synopsis design constraints file (i.e. the .sdc file) that is needed for the Quartus Project counter	33
6	A screen capture of the SignalTap window (Quartus Lite V22.1)	37
7	A screen capture of the node finder window before the fields are filled out (Quartus Lite V22.1).	38
8	A screen capture of the node finder window after it has been filled out. (Quartus Lite V22).	40
9	A screen capture of the SignalTap window showing the trigger table (Quartus Lite V22.1)	41
10	A screen capture of the the trigger table set to trigger when all four signals in cnt transition from 0 to 1 at the same time. Hopefully in later versions of Quartus the table will not display so that it doesn't appear to be vertically compressed (Quartus Lite V22.1).	42
11	A screen capture of the full SignalTap display (data tab selected in the SignalTap window) (Quartus Lite V22.1).	44
12	A screen capture of a zoomed SignalTap display (data tab selected in the SignalTap window) (Quartus Lite V22.1).	44
13	The Architecture for a Sequential B to BCD converter	52
14	The 7-segment displays that are to be used for the lights	61
15	Skeleton state diagram for a controller with walk lights	68
16	The 7-segment displays that are to be used for the traffic and pedestrian lights	69
17	Linear Feedback Shift Register with synchronous Load	72
18	A door with an electric lock mounted on it	76
19	Cyclone II PLL clock generator.	90
20	Four stage ring oscillator.	90
21	Schematic diagram for clock generation circuit.	92

Evaluation Related Material

1.1 Time Schedule

This manual contains 9 Laboratories 2 of which are optional. The expectation is that it will take the students a total of 36 hours to complete the compulsory laboratories and the two laboratory exams¹. The typical schedule, which includes the due dates and the time allotments, is given below:

Event	Start	Due	Time Allotment	
	Date	Date	Pre-Lab	In-Lab
LAB 1	Wk 2	Wk 3	0 hrs	4.5 hrs
LAB 2	Wk 3	Wk 4	1.5 hrs	3 hrs
LAB 3	Wk 4	Wk 5	1.0 hrs	3 hrs
LAB 4	Wk 5	Wk 7	0 hrs	6 hrs
MT Exam	Wk 7			1.5 hrs
LAB 5A&5B	Wk 7	Wk 8	3 hrs	4 hrs
LAB 6	Wk 8	Wk 9		3.5 hrs
LAB 7A&7B	Wk 9	Wk 10	1.5 hrs	3 hrs
Lab 7C	wk 10	wk 12		6 hrs
Final	Wk 12			1.5 hrs
Total			8.5 hrs	36 hrs

- * Week 1 starts on the Monday of the week that classes start unless the first day of classes is a Friday, in which case Week 1 starts on the Monday following the first day of classes.
- ** The due date is 11:59 PM on the Tuesday of the week listed.

Please refer to the course website for the most up-to-date schedule.

Grading

The laboratory reports are neither collected nor graded. However, the students are advised to keep a record of what is done in each laboratory, as such information can be very useful in the Lab exams.

The grading will be entirely based on lab exams, which will be one or two midterms and a final laboratory exam. The midterms and finals will likely be 1 to 1.5 hours in length, but could be a bit shorter or longer, depending on the year. The worth of laboratory exams are given in the course outline.

¹Please note that the laboratory exams will not be held in 2020-2021. Instead, students will be asked to submit specific files and/or other evidence they have successfully completed each laboratory. The specific submission requirements for each lab will be listed on the course webpage.

Important Note

There are numerous versions of the Verilog language. As in the course notes, the terms “Verilog” and “Verilog HDL” are used in a general sense in this document to refer to the entire family of Verilog releases, including the more recent versions, which are known as SystemVerilog

2 Objectives and Learning Outcomes

The overarching objective of the set of Laboratories in this manual is to provide the student with the ability to translate a concept into a digital circuit and implement that circuit on an FPGA.

This overall objective is achieved in a sequence of laboratories, with each lab having its own set of specific objectives. The main objectives of the first few labs are to familiarize the students with the DE2-115 board to a point where they can load the output of the synthesizer into the FPGA and use the features and circuits inside an FPGA. The objective of the next few labs is to augment the theoretical concepts learned in CME341 to include the practical knowledge necessary to design, implement and debug logic circuits that are housed in FPGAs. In these labs emphasis is placed on the practical aspects of debugging a circuit built in hardware (as opposed to a circuit running on a simulator) to get it operational. The objective of the last half of the labs is to provide the students with the ability to translate a concept described in writing into a digital circuit and implement that circuit on an FPGA without having the concept explained in the lectures.

To be more precise, the objectives for the set of laboratories in this manual are to have the students learn:

1. How to configure an FPGA (i.e. load a configuration file into the FPGA).
2. The internal workings of an FPGA.
3. The loading and operation of SignalTap, which is an internal logic analyser.
4. The type of problems that can be easily mapped to a finite state machine.
5. How to implement a finite state machine in an FPGA.
6. How to incorporate time activated events into a finite state machine.
7. The cost of Verilog HDL constructs in terms of the hardware generated. The laboratory exercises will augment the student’s command of Verilog HDL in a practical setting and increase the student’s appreciation of the relationship between Verilog HDL procedures and the hardware they generate as well as the cost of that hardware.

8. To debug a circuit they have designed based on feedback gained by observing/measuring how the circuit responds to a stimulus. This objective is achieved over a sequence of labs where the amount of support provided to help with the design and debugging decreases from one lab to the next.

When the set of labs in the sequence has been completed, the students will have learned a reasonably complex subject using only written material (provided in this laboratory manual) and the feedback from measurements made on the hardware that was built. This type of learning is viewed as independent learning as it is done without lecture style instruction or any other guidance from faculty or staff.

9. To trust their ability to translate a concept into a working circuit.

The overarching learning outcomes, which are the things the students should be able to do upon completion of the entire set of labs, are listed below. Upon completion of the entire set of labs the students should be able to

1. Load a configuration file into the FPGA on the DE2-115 board.
2. Load and operate the SignalTap logic analyser into the FPGA on the DE2-115 board.
3. Design and build a one-hot finite state machine.
4. Incorporate time activated events into a finite state machine.
5. Debug a circuit by first building it and then exciting it and observing/measuring its response.
6. Translate a concept into a working circuit.

BACKGROUND

This Laboratory has been written to accommodate students with different backgrounds and may contain superfluous information for students with a strong background. In the past, students from Electrical Engineering, Computer Engineering, Engineering Physics and Computer Science have taken the class. The laboratory experience of some of these students may not include the use of standard equipment like oscilloscopes and logic analysers. This lab has been written for the lowest common denominator.

LAB EQUIPMENT

1. Altera University Program Design Laboratory Package (See Figure 1 and 2)
 - DE2-115 Education Board
 - Power Supply for the DE2-115 board

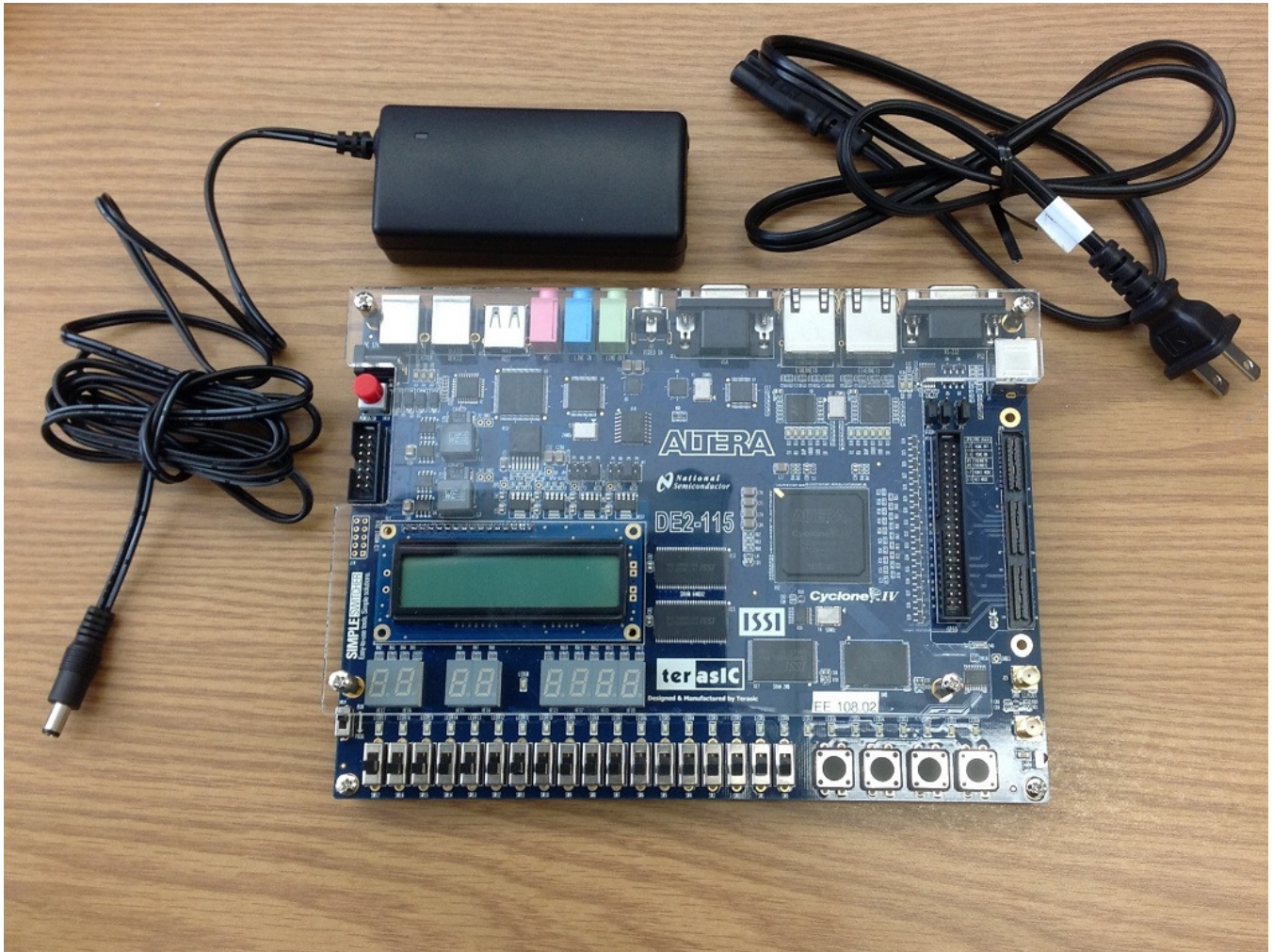


Figure 1: The DE2-115 board

- USB Blaster Cable
2. Windows or Linux Computer running Intel / Altera's Quartus software²
 3. Logic Analyzer
 4. Oscilloscope (Lab or your personal usb-connected oscilloscope)

²Altera was acquired by Intel in 2016, but not all of the relevant documentation and software has been upadated with the Intel name.

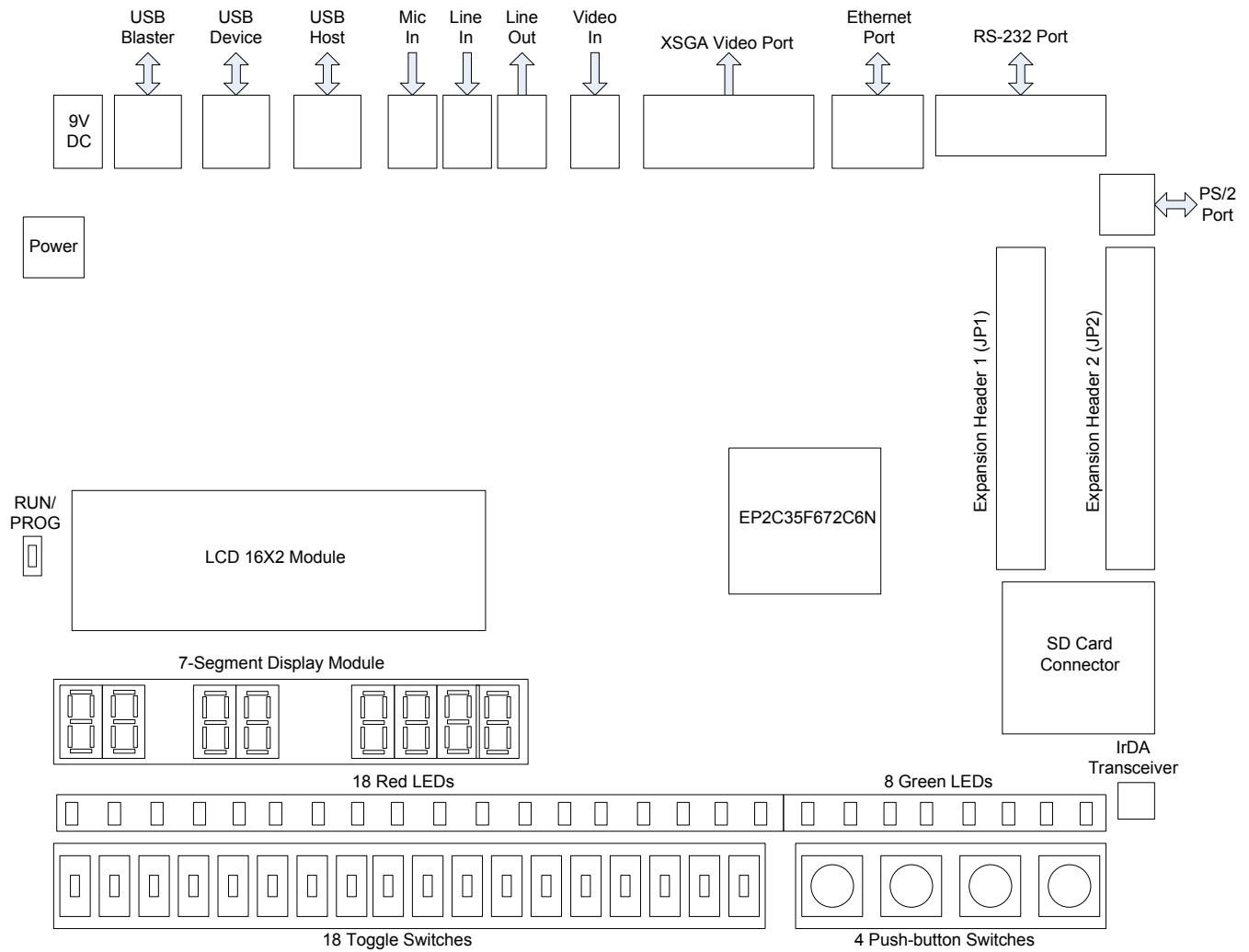


Figure 2: Basic Layout for the Altera DE2 board. The DE2-115 board has the same basic layout, except with the Cyclone IV E FPGA in place of the Cyclone II

3 LAB 1: Getting Started

Objectives and Learning Outcomes

The objectives of this lab are to have the students learn how to:

1. assign pins,
2. set all the unused pins on the FPGA to be tri-stated input pins with weak pull up resistors
3. load a configuration file into the FPGA and
4. program the flash memory on the DE2-115 board to automatically load a configuration file into the FPGA on power-up.

The learning outcomes for this lab are listed below. Upon completion of the Lab the students will be able to:

1. Assign signal to the pins on an FPGA,
2. Set all the unused pins on the FPGA to be tri-stated input pins with weak pull up resistors,
3. Load a configuration file into the FPGA and
4. Program the flash memory on the DE2-115 board to automatically load a configuration file into the FPGA on power-up.

Pre-laboratory assignment

Complete the assignment (on the lecture material) in which an active-high set-clear latch was constructed in Verilog HDL.

3.1 Synthesizing a combinational logic circuit

Make a directory on your hard drive called `CME341_Lab_1`. All projects and files created in Lab 1 should be saved in this directory or a subdirectory within it. These files, especially the .sv file, may be used in future Laboratories and/or Laboratory exams.

3.1.1 Active Low Set Clear Latch

Procedure

1. Create a subdirectory under `CME341_Lab_1` called `set_clear_latch_nand`.
2. Open Quartus and create a new project called `set_clear_latch_nand` in the directory `set_clear_latch_nand`. The project is created by selecting **File** → **New Project Wizard** The wizard will sequence through a series of “forms” that are

to be completed. After each form is completed, clicking the “next” button at the bottom of the form brings up the next form. Each form is described in order of appearance below:

Introduction: Read this page. There is nothing to fill out.

Directory, Name, Top Level Entity: Select directory `set_clear_latch_nand` as the working directory. Name both the project and the top-level entity `set_clear_latch_nand`.

Add Files: There are no files to be added to this project. Just click Next.

Family and Devices: The selections made on this page depend the specific FPGA on the DE2-115 board.

The FPGA on the **DE2-115 board** belongs to the **Cyclone IV E** as the family. It has part number **EP4CE115F29C7N** (look at the DE2-115 board, find the FPGA chip and confirm the part number). Select the part number from the list (the last letter of the part number stamped on the chip, in this case N, is not included in the numbers in the list).

If, later in the lab, you realize you made a mistake entering the part number, the device (i.e. the FPGA family and part number) can be changed using menu option **Assignments**→**Device**.

EDA Tools Setting: Four tool types are listed: Design Entry / Synthesis; Simulation; Formal Verification and Board Level.

Design Entry / Synthesis: All Laboratories in CME341 will use the native Quartus Design Entry / Synthesis tool. This tool is selected by selecting `<none>` for **Design Entry / Synthesis tool**.

Simulation: The simulator as well as the language used by the simulator must be specified. The laboratories in CME341 will likely not use the Questa Intel FPGA simulation tool. However, to be complete, select it by pulling down the **tool name** menu (first column in the simulation row) and selecting **Questa Intel FPGA**. In this class Verilog HDL is used so the simulation format must be Verilog HDL. This is done by selecting **Verilog HDL** in the simulation format box (i.e. third column in the simulation row). This selection will instruct Quartus to produce a Verilog HDL output file, which is basically the Verilog HDL input file enhanced to include all the propagation delays

Formal Verification: Set to `<none>`.

Board Level: All setting should be `<none>`.

Summary Page There is nothing to be filled in on this page. Read this page to ensure that the displayed information matches what you entered, then click on finish.

3. **OPTIONAL:** It is important that all pins on the FPGA that are not used in your design be programmed as input pins, specifically tri-stated input pins, with a weak pull up resistor connected to Vcc. This should be the default setting. To verify this setting select **Assignments** → **Device ...** → **Device and Pin Options** → **Unused**

Pins . Then the box next to “reserve all unused pins:” should read “as input tri-stated with weak pull up”.

4. Create a new SystemVerilog file by selecting **File** → **New** → **Design Files** → **SystemVerilog HDL File**. Enter the code given below and then save the file in the project directory as `set_clear_latch_nand.sv`

```
module set_clear_latch_nand(n_set, n_clear, Q, n_Q);
  input n_set, n_clear;
  output Q, n_Q;

  nand nand_1(Q, n_set, n_Q);      //instantiate nand_1
  nand nand_2(n_Q, Q, n_clear);    //instantiate nand_2

  endmodule // note endmodule is one word and the
           // line does not end in a semicolon
```

5. Draw the equivalent schematic diagram for `set_clear_latch_nand`.
6. Compile `set_clear_latch_nand.v` by selecting **Processing** → **Start Compilation** or by typing **control** + **el** where el = small L. To see just the errors, click on red error icon at the top left of the message window.

If there are any errors correct them and recompile.

7. A successful compile will still generate warnings. In this case there will be 7 warnings (“warnings” means ordinary warnings) and 3 critical warnings for a total of 10.

To see just the warnings, click on either the yellow or blue triangle at the top left of the message window. The yellow triangle will display the ordinary warnings and the blue triangle will display the critical warnings.

Some of these warning/critical warnings require action before the circuit can be downloaded to the FPGA in the DE2-115 board. The warnings and the associated actions that must be taken are described below:

Critical Warning (169085): This warns that the inputs and outputs of the set-clear latch circuit have not been assigned pins. The design is incomplete until this has been done. Pins are assigned as follows:

- (a) Create a comma separated file (a text file) by selecting **File** → **New** → **Other Files** → **text file**. Paste the following text strings into the file and then save the file in the project directory as `set_clear_latch_nand.csv`. Make sure to check mark the box that asks if the file is to be added to the project.

To,	Assignment Name, Value
n_set,	Location, PIN_M23
n_clear,	Location, PIN_M21
Q,	Location, PIN_E21
n_Q,	Location, PIN_E24

This will eventually assign FPGA pins to the inputs and outputs. This will happen on the first recompile after the .csv file is imported into Quartus. The Cyclone IV E FPGA pins PIN_M23 and PIN_M21 are wired to two momentary switches (push buttons on DE2-115 boards) labeled KEY0 and KEY1, respectively, on the DE2-115 boards. The Cyclone IV E FPGA pins PIN_E21 and PIN_E24 are wired to green LEDs labeled LEDG0 and LEDG3, respectively, on the DE2-115 board. The green LEDs are located just above the push button switches on the DE2-115 board.

NB: The momentary switches are active low (i.e. low when pressed) and the green LED are active high (lit when output is 1'b1).

- (b) After the file `set_clear_latch_nand.csv` has been created it must be first imported and then the project must be recompiled for it to take effect. Importing is done by selecting **Assignments** → **Import Assignments** and then selecting `set_clear_latch_and.csv` and clicking on open. This brings up a window that asks if the old .csv file should be backed up. Click OK. After importing, recompile the project.

Cyclone IV E – Warning(15714): The rise time and fall time of the voltage on an output pin depends on the transmission line properties of the printed circuit board track connected to it as well as the impedance of the termination load. For the timing analyser to accurately predict propagation delay on an output pin it must first accurately predict the voltage on an output pin. Since the pin is connected to a track on the printed circuit board and that track is terminated by an impedance of some sort, the timing analyser should use a transmission line model for the track and its load impedance when calculating the voltage on an output pin.

For the newer FPGAs, like the Cyclone IV E, the timing analyser uses a transmission line model for the PCB track. For these devices the user specifies the track length, the transmission line capacitance in farads/inch and inductance in henrys/inch as well as the termination impedance. Specifying the transmission line properties of the PCB tracks will eliminate the warning, but doing so is cumbersome and does not contribute to the objective of this class. If left unspecified the timing analyser makes an assumption and continues on. This warning does not prevent Quartus from generating the .vo and .sof files. Therefore, this warning for the Cyclone IV E will not be addressed. However, for those interested, the transmission line parameters are entered into a schematic diagram, which is obtained by pulling down **Assignments** → **Pin Planner** then clicking on an output pin to select it and finally selecting **View** → **Board Trace Model**.

8. Recompile and verify that two of the three warnings that were addressed above no longer appear in the warning printout.
9. Verify the pin assignments by selecting **Assignment** → **Pin Planner**. In the window showing the top view of the FPGA hover over a red pin to find out what signal has been assigned to it.

10. Configure the FPGA on the DE2-115 board via the JTAG port. Configuring the FPGA means loading (programming) the SRAM that controls all the transistor switches in the FPGA. **This is called JTAG mode programming.** JTAG mode programming is done as follows:
 - (a) Connect the 9V DC power supply to the DE2-115 board and push the red button to turn on the board.
 - (b) There is a slide switch next to the left of the LCD display with positions marked as “run” and “prog”. Make sure this switch is in the “run” position. Strange as it may seem, the switch must be in the “run” position to program the FPGA from the PC.
 - (c) Next connect the DE2-115 board to the PC with the USB cable. The DE2-115 has two receptacles for the USB cable. Connect the USB cable to the one closest to the power supply jack.
 - (d) If Quartus is not already open, open it and open the `set_clear_latch_nand` project.
 - (e) After the cable is connected, Quartus is open and the project of interest is open turn the power to the DE2-115 board off and on (red button) so that Quartus will recognize the DE2-115 board.
 - (f) Program the DE2-115 board by selecting **Tools** → **programmer**. This will open a window that controls the loading of the configuration file. Usually Quartus figures out the specifics of hardware that will be used to configure the FPGA and knows the .sof file (the extension .sof stands for SRAM Object File”, which means it is the configuration file) that is to be used to load the FPGA. If indeed Quartus did find all the necessary information/files the “Start” button will be active. In this case just click on the “Start” button to configure the FPGA.

Manual Set Up: If the “Start” button is not active (i.e. if it is grey instead of black) then Quartus must be informed of the hardware that will be used to configure (i.e. program the SRAM) the FPGA and/or the configuration file (i.e. the .sof file) that is to be loaded into the FPGA.

- i. The mode box, i.e. the box where the mode is to be entered, which is located near the top of the window a little to the right of center, should contain **JTAG**. If it does not, select **JTAG** from the pull down list.
- ii. The hardware that Quartus will use to load the configuration file is listed in the **Hardware Setup** box, which is a long entry box near the top of the window to the right of a button called that **Hardware Setup**. This box should contain **USB-Blaster**. If it does not:
 - A. Click on the **Hardware Setup** button. This will bring up another window called the **Hardware Setup Window**.
 - B. Pull down the menu for the **current selected hardware:** box and select **USB-Blaster [USB-0]**.

If **USB-Blaster [USB-0]** (or perhaps USB-Blaster [USB-3-2] or something like it that starts with USB-Blaster) is not available then the

USB cable is either not connected or connected to the wrong connector on the DE2-115 board. It should be connected to the connector closest to the 9 V DC power connector. Connect the USB cable properly and start over.

After selecting **USB-Blaster [USB-0]** i.e. **USB-Blaster [something]** click on close. This will return control to the **programmer** window.

- iii. If the **start** button in the **programmer** window is still grey, then the programmer can not find the .sof file. The .sof file that Quartus will load into the FPGA must be listed in the window. If no file is listed then one must be selected. There are a couple of reasons why the .sof may not be listed.
 - A. It may not exist because it wasn't generated by Quartus due to compilation errors or it may have been erased.
 - B. The file may not be in the project directory. (For example Quartus versions 12.1 and higher place the .sof file in a subdirectory called **output_files**). To remedy this click on "Add File" and search for the .sof file. It should be named **set_clear_latch_nand.sof**.
 - C. The FPGA part number used in the compilation is different than the part number for the FPGA on the DE2-115 board. To remedy this select **Assignments** → **Device** and select the correct part number and then recompile.

Once the hardware is properly set up and the .sof file is listed in the **programmer** window, the "Start" button will be activated (the font will turn black). At that point just click "Start" to configure the FPGA.

- 11. After the board is programmed one of the two green LEDs that were assigned to **Q** and **n_Q** should be lit and one should be dark. All other LEDs, which includes both the red and green LEDs, should be glowing, but not lit. These LED are glowing because all unassigned pins are programmed to be tri-stated inputs with a weak pull-up resistor. The current that makes the LEDs glow is sourced through the weak pull up resistors. The HEX displays should not be glowing as the LEDs in the HEX displays are active low. The weak pull up resistor pulls these high and turns them off.
- 12. Push and release keys KEY0 and KEY1 to verify that the set-clear latch works properly.
- 13. Once the FPGA is configured, the USB cable can be removed from the computer without affecting the operation of the FPGA. Disconnect the USB cable and once again push and release keys KEY0 and KEY1 to verify that the set-clear latch works properly.
- 14. The wiring in the FPGA is determined by what has been programmed into the SRAM that is part of the FPGA. Since SRAM does not retain its values if power is lost, the FPGA loses its configuration (i.e. its circuit) if the power is removed. Turn the DE2-115 board off and then on again and verify that it has lost the functionality of the set/clear latch.

3.1.2 Verifying the Slide Switches

Objectives

1. Determine the positions at which the slides switches are opened (logical 1) and closed (logical 0).
2. Verify that all the slide switches are operating properly.

Background

The slide switches are part of a simple circuit. One side of the slide is connected to ground, while the other is connected to an FPGA pin as well as a pull-up resistor. When the slide switch is closed the FPGA pin is grounded and when the slide switch is open the pull-up resistor pulls the FPGA pin to Vcc.

Procedure

1. Create a new Quartus project called **slide_switch_verification** in a new directory named **slide_switch_verification**. Generate a Verilog HDL description that:
 - (a) determines the position at which the switch is open and the pull-up resistor pulls up the associated FPGA pin to a logical 1
 - (b) verifies the proper operation of all the slide switches on the DE2-115 board.

Do this by writing a Verilog HDL module with a port list that contains three vectors: input vector **SW[17:0]**, output vector **HEX1[6:0]** and output vector **LEDR[17:7]**. This module assigns switches **SW[6:0]** to hexadecimal display **HEX1[6:0]** and switches **SW[17:7]** to red LEDs **LEDR[17:7]**. The FPGA pin numbers for the switches, LEDs and HEX displays are given for both the DE2 and DE2-115 boards in Appendix A.

2. Make all unused pins tri-stated inputs with weak pull up resistors.
3. Assign the appropriate pins to the signals in the Verilog HDL.
4. Recompile.
5. Configure the FPGA on the DE2-115 board using JTAG mode.
6. Verify the operation of the slide switches.
7. Verify that the 7 LEDs (i.e. segments) in the HEX displays are active low.

3.1.3 Active High Set Clear Latch

Objectives

The objectives are:

1. To learn the voltage levels of FPGA I/O pins configured as output pins that correspond to a logic 0 and a logic 1.

Procedure

Configure the FPGA on the DE2-115 board with the nor-gate based set-clear latch that was designed and built as part of a CME341 assignment. Use slide switches SW[17] and SW[0] as the `set` and `clear` inputs, respectively, and assign the pins for red LEDs LEDR[17] and LEDR[0] to the `Q` and `n_Q` outputs, respectively. Configure the FPGA on the DE2-115 board and verify that the circuit works properly.

After establishing the proper operation of the circuit, modify the Verilog HDL description using the Verilog HDL “assign” statement, e.g. `assign Q_extra = Q;` to make a pair of outputs that duplicate `Q` and `n_Q`. Call the duplicates `Q_extra` and `n_Q_extra`. Connect outputs `Q_extra` and `n_Q_extra` to the FPGA pins that are wired with a PCB tracks to at GPIO_0 pins 1 and 2. See Table 8 on Page 106 to find the corresponding FPGA pin numbers. Note the pins for a Cyclone II FPGA are also given. Be sure to use the pins for a Cyclone IV E.

Recompile and configure the FPGA on the DE2/DE2-115 board.

Connect your own or a department oscilloscope to pins 1 and 2 of the GPIO_0 header to verify the voltages on these pins mimic the outputs `Q` and `n_Q`, respectively. The probes from the Digilent Discovery or equivalent oscilloscope (your own personal oscilloscope) will connect directly to the pins on the header, however the probes from a department oscilloscope will not. If you use a department oscilloscope you will need to remove the spring loaded grabber from the end of the probe. This uncovers a sharp pin which is now the scope probe. Touch the pin 1 or 2 on the GPIO_0 header with the exposed probe pin. Be very careful not to touch the ring on the probe located below the probe pin, which is grounded through the oscilloscope, on any of the neighbouring GPIO_0 pins.

Record (at least note and remember) the voltages that correspond to a logic 0 and a logic 1. These are the output levels of the FPGA I/O pins.

3.1.4 Programming the Flash Memory on the DE2-115 Board

Objectives

The objectives are

1. To learn to program the flash memory on the DE2-115 board.

Background

The DE2-115 board has flash memory, which is non-volatile memory (memory that maintains its integrity while powered down). The DE2-115 board has been designed to transfer the contents of this flash memory into the SRAM in the FPGA at power-up. The DE2-115 board has also been designed to program the flash memory when the prog/run switch is in the “prog” position.

At the time of purchase the flash memory is programmed with configuration information for a circuit that tests the lights on the DE2-115 board. When the board is powered-up the SRAM is loaded with the test circuit that is programmed into the flash memory.

Of course, as already has been done in this lab, the contents of the SRAM can be overwritten by “JTAG mode” programming.

The Quartus compiler produces a file that contains the data to be loaded into the flash memory. The flash programming file has an extension .pof (short for “Programmer Object File”). Programming the flash memory will over-write its contents. On power-up the contents of flash memory is transferred into the SRAM in the FPGA.

The flash memory is programmed in a different mode than the SRAM in the FPGA. The flash memory is programmed in **Active Serial Mode**.

Procedure

Program the flash memory on the DE2-115 board with the active-high set clear latch. Do this by following the procedure below:

1. Set the RUN/PROG switch on the DE2-115 board to the PROG position.
2. The DE2-115 board is designed with a EPCS64 configuration device. This device enables programming when the RUN/PROG switch is in the PROG position. The device is usually automatically found, but can be selected manually by: **Assignments → Device → Device & Pin Options** and then selecting “Configuration” from the list in the “category:” pane. Pull down the menu for the “configuration device box” and choose **EPCS64**. Then click OK to get back to the “Device” window and the click OK again.
3. Recompile your design.
4. Select **Tools → Programmer**. In the Mode box select **Active Serial Programming**. If previously the mode setting was JTAG, a pop-up box will appear, asking if you want to clear all devices. Click "Yes".
The “Hardware Setup” box must contain **USB-Blaster**. If it does not make it so.
5. If the .pof configuration file is not listed in the window, click “Add File” then find and select the .pof file for the project.
6. Check mark the **Program/Configure** check box in the programmer window. The window may have to be stretched horizontally to see the box. **The start button will not activate (i.e. turn black) until the program/configure box is check marked.**
7. Click start to program the flash memory. This could take a minute or two. The progress is shown in the “progress” box.
8. Wait until the progress box indicates the programming process is completed then slide the RUN/PROG switch back into the RUN position and reset the board by turning the power off and on (red button). The power-up will initiate the transfer of the configuration information from the flash memory to the SRAM in the FPGA.

Verify that the set/clear latch circuit works properly.

3.1.5 Re-Programming the Flash Memory on the DE2-115 Board with the original test circuit

Objectives

The objectives are:

1. To restore the DE2-115 board to its time-of-purchase state.

Background

The .pof file for the test circuit that was initially in flash memory is posted on the class web site along with the other lab files. It is called: 'DE2_115_default_test_circuit.pof'.

Procedure

Reprogram the flash memory with the default test circuit and verify that all the LED and HEX displays are operational.

NB: For subsequent Laboratories, unless otherwise specified, the FPGA is to be configured in JTAG mode with the prog/run switch in the "run" position.

3.1.6 Hex Display Driver

If you have reached this part of the lab and the "case" structure has not yet been covered in class, please defer this "Hex Display Driver" section to next lab.

Objectives

1. Learn to design, build and test a very simple combinational logic circuit.
2. Build a useful prototype that can and will be used in subsequent labs.

Description

Design a combinational logic circuit that has a 4 bit input and a 7 bit output. Each output drives one of the seven segments on a 7 segment display. The function of the circuit is to make a seven segment display, display the 4-bit binary input in hexadecimal format. The input output relationship is shown in Figure 3.

Procedure

Create a new Quartus project called `hex_display_driver` in a new directory named `hex_display_driver`. Make a module with a 4-bit input vector called `hex_digit` and a 7-bit output vector called `hex_segments`. The circuit is to use slide switches `SW[7:4]` for the input `hex_digit` (i.e. assign the pins for `SW[7:4]` to `hex_digit`)

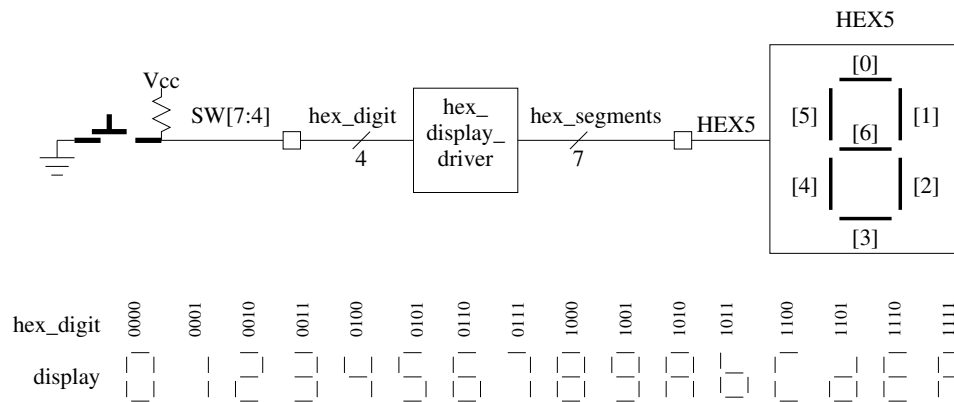


Figure 3: The display format for a hexadecimal number

and use HEX5 to display the binary number hexadecimal format (i.e. assign the pins for HEX5 to hex_segments).

Please construct the circuit using a single case statement inside an always @ * procedure.

NB: Make sure all unused pins have been set to “tri-stated inputs with weak pull up resistors” before configuring the FPGA.

Configure the DE2-115 board in JTAG mode and verify that the circuit works. Also read the compiler report to determine the number of Look-up-Tables (LUTs) used.

4 LAB 2: Designing a Binary to Binary-Coded-Decimal Converter

Objectives and Learning Outcomes

The objectives are:

1. To learn how to convert concepts into a Verilog HDL description.
2. To learn to design a circuit in a way that it can be broken into a few well defined blocks that can be debugged separately.
3. To learn that debugging a circuit one part at a time is effective (perhaps even necessary).
4. To get an appreciation for the amount of FPGA resources generated by comparators (i.e. the circuit generated by the “test condition” specified in an “if” statement).

The learning outcomes are to be able to

1. Convert a concept for a circuit into a Verilog HDL description.
2. Design a circuit in a way that it can be broken into a few well defined blocks that can be debugged separately.
3. Debug a circuit efficiently by selecting and debugging one part of the circuit at a time.
4. Determine from the compiler report the FPGA resources generated by the “test condition” specified in an “if” statement.

Pre-laboratory assignment

Time required estimated at 1.5 hrs.

Read the laboratory, set up all necessary directories and create the Quartus project. Also design the 10's digit circuit, but write the Verilog HDL for 10's digit circuit in a temporary file so that it can be copied into the top entity file at the appropriate point in the laboratory.

Background

Mathematical and relational operators use binary (i.e. base 2) operands and produce binary (base 2) results. Humans do not have a great appreciation for base 2 numbers so calculators and computers convert the binary result to binary coded decimal so that it can be displayed in human friendly base 10. Binary coded decimal represents a base 10 digit with a 4-bit binary vector. The binary coded decimal vectors are 4'b0000, 4'b0001, 4'b0010, ..., 4'b1001 and represent the decimal numbers from 0 to 9 in ascending order.

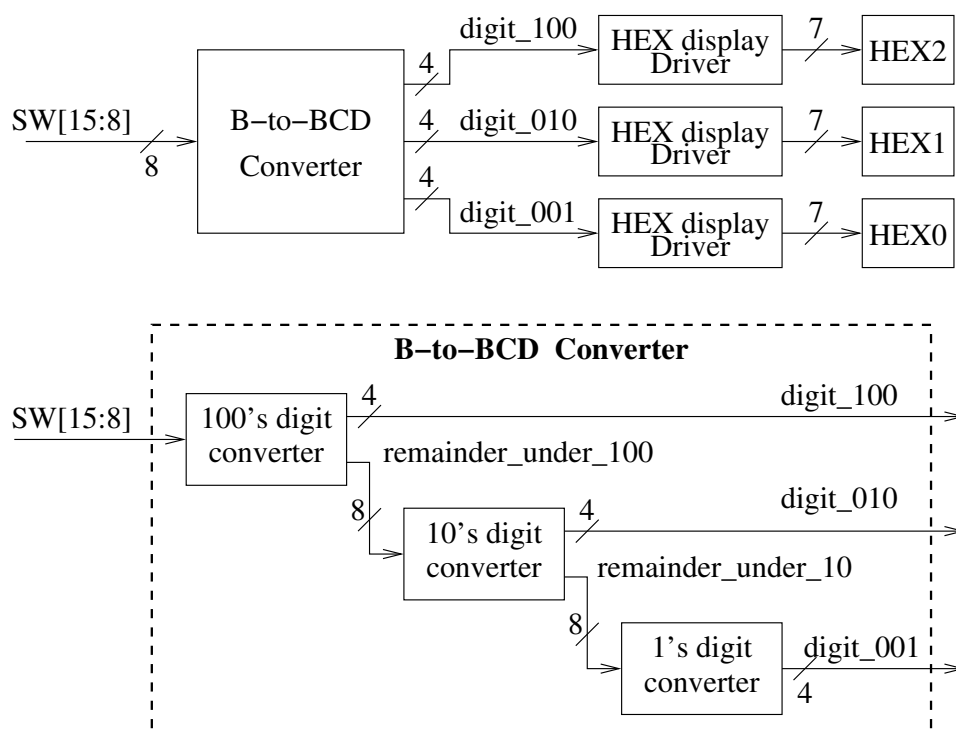


Figure 4: Block diagram for a Binary to Binary-Coded-Decimal Converter

For example, the 8-bit binary (base 2) unsigned integer `8'b1000001` expressed in base 10 is 129. The conversion circuit converts the 8-bit binary input into three 4-bit binary coded decimal outputs, which are referred to as `digit_100`, `digit_10` and `digit_1`. For example, if the input was `8'b1000001` the output would be `digit_100 = 4'b0001`, `digit_10 = 4'b0010`, and `digit_1 = 4'b1001`.

The problem

Design a binary to binary-coded-decimal converter for an 8-bit binary input. Use switches `SW[15:8]` as the input and display the three digit output on the DE2-115 board hexadecimal displays labelled `HEX2`, `HEX1` and `HEX0`.

Figure 4 shows a block diagram of the entire system as well as the architecture for the Binary to Binary-Coded-Decimal (B-to-BCD) converter. The system block diagram has only two types of blocks, however the “hex display driver” block is used three times. The HEX display driver has already been designed and debugged in a previous lab so only the B-to-BCD converter needs to be designed in this lab.

Figure 4 also shows an architecture for the B-to-BCD converter. The “100’s digit converter” block extracts the most significant base 10 digit, which is the 100’s digit, from the 8 bit binary number and outputs what is left over in the 8-bit vector `remainder_under_100`. This output will be a binary (base 2) number that is less than the 100 base 10. The “100’s digit converter” block also outputs

the most significant base 10 digit in BCD format.

(ASIDE: The output remainder_under_100 is sure to be less than 8'd100 which is 8'b0110_0100. Therefore remainder_under_100 could be a 7-bit vector, but it is more convenient to use 8-bits. Using 8-bits will not cost more as the compiler will optimize away the most significant bit.)

The “10’s digit converter” block extracts the second digit from remainder_under_100 and outputs what is left over in the 8-bit vector remainder_under_10. The “1’s digit converter” block acts similarly on remainder_under_10.

NB: For purposes of referencing and making modifications during exams use the signal names given in Figure 4.

Procedure

1. Create a directory on the hard drive called CME341_Lab_2 and a subdirectory within it called B_to_BCD_converter_with_hex_driver.
2. Copy the module named hex_display_driver.sv that you constructed in Laboratory 1 and place the copy in subdirectory B_to_BCD_converter_with_hex_driver.
3. Create a Quartus project called B_to_BCD_converter_with_hex_driver and make the project directory B_to_BCD_converter_with_hex_driver. Make the top entity the module called B_to_BCD_converter_with_hex_driver. When the “Add Files” window appears in the sequence of windows presented by the new project wizard find and then “add” the recently copied hex_display_driver.sv to the project. This can be done later using **Project** → **Add/Remove Files in Project ...** and then finding and hex_display_driver.sv and then clicking on **add**.
4. The first step in designing an 8-bit binary to 3-digit decimal converter given in Figure 4 is to design, build and debug the 100’s digit converter. A Verilog HDL design for this block is given below.

```
module B_to_BCD_converter_with_hex_driver(
    input [7:0] binary_input,
    output reg[7:0] remainder_under_100,
    output wire [6:0]seven_segments_100
);
    reg[3:0] digit_100;
    always @ *
    if (binary_input >= 8'd200) // note: since binary_input is 8 bits,
                                // its maximum value is 255
    begin
        digit_100 = 4'd2;
        remainder_under_100 = binary_input - 8'd200; // max value is 55
    end
endmodule
```



```

end
else if (binary_input >= 8'd100)
    begin
    digit_100 = 4'd1;
    remainder_under_100 = binary_input - 8'd100; // max value is 99
    end
else
    begin
    digit_100 = 4'd0;
    remainder_under_100 = binary_input - 8'd000 ; // max value is 99
    end

hex_display_driver hex_driver_100(
    .hex_digit(digit_100),
    .hex_segments(seven_segments_100)
);

endmodule

```

Study the Verilog HDL module above until you understand its operation and then cut it from this file and paste it into a file called B_to_BCD_converter_with_hex_driver.sv. Assign the input to switches SW[15:8] and outputs seven_segments_100 and remainder_under_100 to HEX2 and red leds LEDR[15:8], respectively. The pin assignments for the DE2-115 board are listed below.

```

# pin assignments for DE2-115 board
To, Assignment Name, Value
binary_input[0], Location, PIN_AC25
binary_input[1], Location, PIN_AB25
binary_input[2], Location, PIN_AC24
binary_input[3], Location, PIN_AB24
binary_input[4], Location, PIN_AB23
binary_input[5], Location, PIN_AA24
binary_input[6], Location, PIN_AA23
binary_input[7], Location, PIN_AA22
seven_segments_100[0], Location, PIN_AA25
seven_segments_100[1], Location, PIN_AA26
seven_segments_100[2], Location, PIN_Y25
seven_segments_100[3], Location, PIN_W26
seven_segments_100[4], Location, PIN_Y26
seven_segments_100[5], Location, PIN_W27
seven_segments_100[6], Location, PIN_W28
remainder_under_100[0], Location, PIN_J17
remainder_under_100[1], Location, PIN_G17

```

```

remainder_under_100[2], Location, PIN_J15
remainder_under_100[3], Location, PIN_H16
remainder_under_100[4], Location, PIN_J16
remainder_under_100[5], Location, PIN_H17
remainder_under_100[6], Location, PIN_F15
remainder_under_100[7], Location, PIN_G15

```

Make sure to assign all unused pins to “tri-stated input with weak pull up resistor”.

Then compile the project and configure the FPGA on the DE2-115 board. Check the operation. Make sure the most significant digit is correct and the remainder is also correct.

5. The second step is to design, build, debug and test a “10’s digit converter” block. The input to this module only need six bits because its can be at most 99. For purposes of debugging, make the input to the “10’s digit converter” the least significant 6 bits of binary_input instead of remainder_under_100. These 6 bits are set by the 6 switches SW[13:8].

Since binary_input is also the input to 100’s digit circuit, ignore the hex display for the 100’s digit while changing the least significant 6 switches to test the 10’s digit circuit.

Design the “10’s digit converter” block and place the Verilog HDL for it in

B_to_BCD_converter_with_hex_driver.sv (Do not delete the Verilog HDL for the “100’s digit converter”). In this step of the design binary_input is to be the used as the input for both the “10’s digit converter” block and the “100’s digit converter” block. At this point the “10’s digit converter” and “100’s digit converter” operate in parallel.

For this step the port list for module B_to_BCD_converter_with_hex_driver, in Verilog HDL 2001 format, will be

```

module B_to_BCD_converter_with_hex_driver(
input [7:0] binary_input,
output reg[7:0] remainder_under_100,remainder_under_10,
output wire[6:0] seven_segments_100, seven_segments_10
);

```

- (a) Please use the same system for naming for the signals, which means the signal names for the 10’s digit converter will have “10” instead of “100” in them. For example, the second instantiation of hex_display_driver should be named hex_driver_10.
- (b) Display the 10’s digit on HEX1 and output remainder_under_10 on red LEDs LEDR[7:0].

- (c) Update the .csv file to assign the pins for the two new outputs. Then recompile and configure the FPGA.
 - (d) Test the “10’s digit converter” circuit for proper operation.
6. The third step is to design, build, debug and test a “1’s digit converter” using the philosophy that was used to design, build, debug and test the “10’s digit converter”.
- As was done for the 10’s digit converter, place the Verilog HDL description in `B_to_BCD_converter_with_hex_driver.sv` and modify the port list.
7. Sometimes engineers get tunnel vision when designing a sequence of blocks that do similar things. In such cases the engineers will sometimes omit the input-output analysis that they usually perform to gain insight into the circuit. That could well be the case for the 1’s digit converter. This step leads the students through an input-output analysis for the 1’s digit converter.
- (a) To start the analysis look at the compiler report and record the number of LUTs used to build your prototype `B_to_BCD_converter_with_hex_driver`.
 - (b) The input to the 1’s digit converter is the 8-bit vector `remainder_under_10`. Make a table showing the 10 possible values that `remainder_under_10` can take on and the 10 corresponding 4-bit outputs for the 1’s digit converter.
 - (c) It should be clear at this point that the 1’s digit converter does not require any logic. The output of the ones digit converter is the least significant 4 bits of its input.
8. Modify `B_to_BCD_converter_with_hex_driver.sv` so that the 1’s, 10’s and 100’s digit converters are properly connected to each other to make an 8-bit binary to three decimal digit converter as shown in Figure 4.

Display the 1’s digit output on hex display HEX0. Debug the newly formed circuit and test it for proper operation.

9. Observe the compiler report to record the “total logic elements” (i.e. the total number of logic elements) required. Each logic element has a flip/flop (also called a register) and combinational logic in the form of a look-up-table. From the compiler report record what parts of the logic elements were used. That is record the “total combinational functions” (i.e. total number of look-up-tables) and the total number of register that were used in your design.

5 LAB 3: Sequential Logic

Objectives and Learning Outcomes

The objectives of this lab are:

1. To learn how to declare clock pins. That is, inform Quartus which pins on the FPGA are connected to the outputs of a circuits on the Printed Circuit Board (PCB) that generates square waves. Such circuits are often referred to as clock generators or simply clocks.
2. To learn how to generate a Synopsys Design Constraints (.sdc) file.
3. To learn how to fabricate a logic analyser (referred to as SignalTap) inside the FPGA.
4. To learn how to use SignalTap (i.e. the logic analyser that can be fabricated inside the FPGA).

The learning outcomes of this lab are to be able to

1. Declare clock pins.
2. Generate a Synopsys Design Constraints (.sdc) file.
3. Load a logic analyser circuit (SignalTap) into the FPGA.
4. Use the SignalTap logic analyser.

Pre-laboratory assignment

When you arrive at the lab be prepared to start with item 12 on page 34.

5.1 Synthesizing a circuit that uses a clock

Objectives

The objectives of this section are:

1. To learn how to declare clock pins, i.e. inform Quartus of the properties of the clocks connected to certain pins. For example, provide Quartus with the frequency of the clock.
2. To learn how to generate a Synopsys Design Constraints (.sdc) file.

Procedure

1. Create a new directory/folder called CME341_Lab_3 and inside it create another directory/folder called counter.
2. Open Quartus and create a new Quartus project:
 - (a) Name the project counter.
 - (b) Make the project directory CME341_Lab_3/counter, which is the one you created above.
 - (c) Make the top level entity counter.
 - (d) Do not add any files to the project. The source for module counter, i.e. the file counter.sv, will be added to the project when it is created.
 - (e) On the family and device setting page, select part number **Cyclone IV E EP4CE115F29C7**, which is the FPGA on the DE2-115 board.
 - (f) Set the EDA simulation tool to Questa Intel FPGA and specify the format is to be Verilog HDL.
3. Enter the Verilog description below into a new file called “counter.sv”, add it to the project and make it the top entity.

```
module counter(up_down, ena, clk, cnt, clk_out);
input up_down, ena, clk;
output clk_out;
output [3:0] cnt;

assign clk_out = clk;

reg [3:0] cnt;
always @ (posedge clk)
    if (ena == 1'b0)
        cnt = cnt;
    else if (up_down == 1'b1)
        cnt = cnt + 4'd1;
    else
        cnt = cnt - 4'd1;
endmodule
```

4. Verify that all unused I/O pins are set to be “tri-stated inputs with weak pull-up resistors”. To verify this setting, select **Assignments → Device ... → Device and Pin Options → Unused Pins** . Then verify that the box next to “reserve all unused pins:” reads “as input tri-stated with weak pull up”. If the box reads something else, pull down the options and select “as input tri-stated with weak pull up”.

Table 1: Explanation of routing from signals in module `counter` through pins on the Cyclone IV E FPGA to GPIO header and other circuits

Signal name	Location (Cyclone IV E pin #)	connects to
<code>clk</code>	<code>PIN_B14</code>	on board 27 MHz oscillator
<code>clk_out</code>	<code>PIN_AB22</code>	pin 1 on expansion header 1 (<code>GPIO_0</code>)
<code>cnt[3]</code>	<code>PIN_AF26</code>	pin 34 on expansion header 1 (<code>GPIO_0</code>)
<code>cnt[2]</code>	<code>PIN_AD22</code>	pin 24 on expansion header 1 (<code>GPIO_0</code>)
<code>cnt[1]</code>	<code>PIN_AF15</code>	pin 16 on expansion header 1 (<code>GPIO_0</code>)
<code>cnt[0]</code>	<code>PIN_AC15</code>	pin 2 on expansion header 1 (<code>GPIO_0</code>)
<code>ena</code>	<code>PIN_M23</code>	on board push button switch <code>KEY0</code>
<code>up_down</code>	<code>PIN_AB28</code>	on board toggle switch <code>SW0</code>

Table 2: The .csv format for Pin Assignments for module `counter`

To,	Assignment Name,	Value
<code>clk,</code>	<code>Location,</code>	<code>PIN_B14</code>
<code>clk_out,</code>	<code>Location,</code>	<code>PIN_AB22</code>
<code>cnt[3],</code>	<code>Location,</code>	<code>PIN_AF26</code>
<code>cnt[2],</code>	<code>Location,</code>	<code>PIN_AD22</code>
<code>cnt[1],</code>	<code>Location,</code>	<code>PIN_AF15</code>
<code>cnt[0],</code>	<code>Location,</code>	<code>PIN_AC15</code>
<code>ena,</code>	<code>Location,</code>	<code>PIN_M23</code>
<code>up_down,</code>	<code>Location,</code>	<code>PIN_AB28</code>

5. Create a comma separated variables file, i.e. a .csv file, to define the pin assignments for project `counter`. (By the way, comma separated variables files are text files so could have extension .txt, but .csv is a more meaningful extension.) Table 1 explains the connections from the signals in module `counter` through the pins on the Cyclone IV E FPGA and on to the `GPIO_0` header. However, Table 1 is not in a form that can be cut and pasted into the .csv or .txt file, but Table 2 is in the right form to cut and paste into a text editor to make the .csv file.

It should be pointed out that switch `SW[0]` on the DE2-115 board is wired to FPGA pin `PIN_AB28`. This means input `up_down` to circuit “`counter`” is `SW[0]`.

It should also be pointed out that push button switch `KEY[0]` on the DE2-115 board is wired to FPGA pin `PIN_M23`. This means input `ena` to circuit “`counter`” is `KEY[0]`. It is also important to know that `KEY[0]` is a momentary push button switch that is **active low**, meaning it is normally 1'b1 and only becomes 1'b0 when someone is pushing the button.

To do some of the future labs, knowledge of which pins on Cyclone IV E FPGA are wired to which specialty circuits on the DE2-115 board will be needed. Those connections are given in Appendix A from page 94 to page 105. The connections are specified in the format required for the .csv file so can be cut and pasted as needed.

In particular to mapping from the Cyclone IV E FPGA pins to the GPIO_0 header will be useful. To make this information more accessible it has been organized into a separate table, Table 8 on page 106. It is pointed out Table 8 also provides the mapping from the GPIO_0 header to the FPGA on an older version of the DE2-115 board, which was called the DE2 board. The older DE2 board, which is no longer used in our labs, used a Cyclone II FPGA (not the Cyclone IV E FPGA). The pin numbers in Table 8 under the columns labeled "Cyclone II Pins" are for the outdated DE2 boards. Please be careful in reading the table to get the pin numbers for the Cyclone IV E.

6. After creating the .csv file, import it into Quartus. This is done by selecting **Assignments** → **Import Assignment...** and then entering the name of .csv file or .txt file with the pin assignment information.

NB: NB: NB: After importing the .csv or .txt file be sure to read the comments in the message window (green font). The message window has two tabs at the bottom: one for messages from menu commands like the programmer, importing assignments, and SignalTap. The other tab is for messages from the compiler. The "Processing" tab is basically for messages from the compiler. The "System" tab is for messages from menu commands like "import assignments".

Look at the messages in the "System" tab. (the Systems tap is located at the bottom of the message window) They should say 8 pins were written, meaning 8 pins were assigned to signals. If for example, you had a spelling error in one of the fields at the top, it will not report an error. It will report in green print: Import completed: 0 assignments were written ...)

7. After importing the pin assignments, recompile project counter. The project must be recompiled for the pin assignments to take effect. Then look at the warnings and notice three things;
 - (a) There should be no warnings about any pin or pins having not been assigned. If there is such a warning then there was either a typo or omission in the .csv or .txt file that was imported.
 - (b) There should be a warning (all warnings are printed in blue font) that appears one or more times that reads something to the effect:
332012 Synopsys Design Constraints File file not found: 'counter.sdc'.
A Synopsys Design Constraints File is required by the Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design.

The wording of this warning may change from one version of Quartus to another.

- (c) There should also be a warning (blue font) that appears one or more times that reads something to the effect:

332148 Timing requirements not met.

This warning will appear if the Synopsys Design Constraints File (i.e. the .sdc file) is missing. If this warning appears and the .sdc file has been included in the project, then it indicates there is a timing problem. Specifically, it means the propagation delay through the logic that sits between the output of one flip/flop and input of another flip/flop is greater than one clock period.

In this case this warning, i.e. warning 332148, is just a companion to warning 332012, which is the warning that indicates the .sdc file is missing.

8. To remedy warning “332012 Synopsys Design Constraints File file not found: 'counter.sdc'”, a Synopsys Design Constraints (.sdc) file must be created. The .sdc file is a text file that contains information on the period of the clock and the duty cycle of the clock. The timing analyzer needs this information to check for timing issues.

The .sdc file should have the same name as the Quartus project. While the .sdc is a “text” file, it should have extension .sdc. Therefore, for this project, which is titled counter, the .sdc file would be named counter.sdc.

The .sdc file contains Tcl commands, where Tcl stands for Tool command language. The clock can be specified with a single Tcl command. However, a Tcl command that specifies the units of time that are to be used throughout the script is also needed. The .sdc script needed for project counter is given in Figure 5 and explained below. Note that a comment in Tcl begins with #.

The first command, which is `set_time_format -unit ns -decimal_places 3`, is self explanatory. It sets the units of time to nanoseconds and sets the precision to three decimal places, which in this case is 0.001 ns.

The second command is a `create_clock` command. **Note that this command does not actually create a clock circuit.** It just specifies the properties of the clock generated by a circuit on the DE2-115 board. It uses three options: `-name`, `-period`, and `-waveform`. The `-name` field assigns a second name to `clk`. It can be any name as long as it is unique. The `create_clock` command in Figure 5 assigns “`clk`” the second name `sys_clk`.

The `-period` field is where the period of the clock is specified. Of course, verilog signal `clk` must be connected to the output of a clock circuit on the DE2-115 board. Since the clock circuit on the DE2-115 board is wired to pin `PIN_B14` and that pin has been assigned to verilog signal `clk`, the argument to the `-period` option must be the period of the clock generated on the DE2-115 board. That clock has a frequency of 27 MHz, which means


```

#####
#####
#    Start of .sdc script
#

#####
# Time Information --- set units to nanoseconds to an accuracy of 3 decimal places
#####

set_time_format -unit ns -decimal_places 3

#####
# Create Clock --- numbers are in nanoseconds
#####

create_clock -name {sys_clk} -period 37.000 -waveform { 0.000 18.500 } [get_ports {clk}]

#
#    End of .sdc script
#####
#####

```

Figure 5: The content of the synopsis design constraints file (i.e. the .sdc file) that is needed for the Quartus Project counter.

it has a period of 37 ns. Since the `set_time_format` command set the units of time to ns, the argument for the `-period` option must be 37.000. I.e. `-period 37.000` declares the period of Verilog signal `clk` to be 37 ns.

The `-waveform` field provides information about the duty cycle of the clock. The field contains two numbers, both of which are times in ns. The first number is the time of the rising edge and the second number is the time of the falling edge. The clock circuit on the DE2-115 board produces a square clock (i.e. a clock waveform with a 50% duty cycle). Therefore, the time of the rising edge is declared to be the beginning of the period and the time of the falling edge is declared to be mid period.

The `[get_ports clk]` Tcl command gets the pin number for the signal named `clk`. If the clock for module `counter` was named something else, say `clock_A`, then `clock_A` would replace `clk` in the argument of the `[get_ports clk]` Tcl command.

Cut and paste Figure 5 into a text editor and name the file `counter.sdc`. Save the file in the folder `counter` and add the file to the project.

9. Recompile the project. Warning 332012 should no longer be issued.
10. Once the clock period has been specified the Timing Analyzer will report things like the “worst case set-up slack” and “worst case hold slack” for both fast and slow models of the FPGA. However, while such measurements can be very helpful, we will not be using them in this lab.
11. Now changing gears a bit: The compiler places the output file, which it names `counter.vo` in a subdirectory of the project directory. Specifically, `counter.vo` is placed in subdirectory `simulation` → `questa` (In older versions of Quartus the subdirectory is `simulation` → `modelsim`). This `.vo` file is intended for use in the simulator. It is not used in the generation of the `.sof` configuration file.
12. Configure the FPGA on the DE2-115 board with the counter circuit. I.e. program the FPGA with `counter.sof`.

It is pointed out that configuring the PFPGA means programming the static ram inside the FPGA. Each bit in the static ram controls a transistor switch that can be either on or off. These transistor switches connect the track segments in the FPGA together to make the circuit specified by the prototype Verilog HDL. The `.sof` file contains the image for the the static ram so it is the input file to the programmer.

Other phases are used interchangeably with “programming the FPGA”. Some of them are: loading the `.sof` file, loading the FPGA, configuring the FPGA, loading the circuit into the FPGA and configuring the circuit.

5.2 Loading the circuit together with SignalTap (SignalTap is an Embedded Logic Analyser provided by Intel) and then Using SignalTap

Objectives

Background

Logic analyzers are instruments that display many binary, i.e. 0 or 1, waveforms on a screen. What makes logic analysers powerful is their ability to trigger as soon as several signals simultaneously have the values specified for the trigger condition.

A logic analyzer is similar to an oscilloscope in that it has probes that connect to voltage signals. However, its probes are logic probes. The probes contain a comparator with a preset threshold so that the output of the logic probe is either ground or V_{cc} .

The outputs of the logic probes are sampled on edges of a clock and stored in a buffer as they arrive. Once the buffer is full and the sampling process has stopped, the logical values in the buffer are displayed on the PC as a waveform.

The sampling process is stopped after two things have happened: First, the specified “trigger” pattern must have been encountered and second an additional fixed number of samples must have been collected and stored in the buffer. Once the sampling process has been stopped, the content of the buffer is displayed on the PC screen graphically as binary wave forms (one waveform for each logic probe) stacked one above the other. A vertical dashed line through the the waveforms is used to mark the time the trigger condition was met. The samples are numbered relative to the time the trigger condition is met. The sample where the trigger condition is met is assigned sample number 0. The samples before the trigger condition are numbered with negative numbers and samples the trigger condition are numbered with positive numbers.

In summary, the PC displays binary valued signals that span a time frame that starts before the trigger condition is met and ends after the trigger condition is met. The point in time when the trigger condition occurred is marked with a vertical dashed line. Also the sample numbers at the top of the display are with respect to the trigger. That is to say sample number 0 is the sample that meets the trigger condition and sample number -1 is the sample prior to the sample that meets the trigger condition etcetera.

Stand-alone external logic analyzers like the ones the ECE department owns or like the one that is part of the Analog Discovery module are powerful instruments, but have limited value when it comes to debugging FPGAs. The reason that the many of pins for FPGAs are not accessible to us. Pins on the bottom of the chip (called ball-grid arrays) can connect to a pin on the bottom of another chip with with a trace in the middle of a multilayered printed circuit board.

FPGA manufacturers fully understand the value of logic analyzers in the debugging process so they provide a hardware description in Verilog HDL for a logic analyzer

that can be embedded in the FPGA. The Verilog HDL of the embedded logic analyser is implicitly instantiated in the top level module. Intel calls the prototype for their Verilog HDL logic analyzer SignalTap. The names it uses for the instances (which are also called the instantiations) are SignalTap with a number appended, for example: SignalTap_0.

SignalTap is used like a regular logic analyzer, except the probes can be connected to any signal in the design, not just an input or output of the top level module. Once compiled, the logic analyzer communicates with the PC through the JTAG port, which is the same serial port used to program the FPGA with the .sof file. The JTAG port is used to transfer the trigger conditions (which are entered by the user via a gui on the PC) from the PC to the instantiated SignalTap circuit in the FPGA. The JTAG port is also used to transfer the samples collected by the SignalTap circuit to the PC so they can be displayed.

This part of the lab demonstrates how SignalTap is included in a project and how it is used.

5.2.1 Using SignalTab to debug prototype counter

In this subsection we are going to get Quartus to display 128 consecutive values of the output of prototype counter.sv. The displayed waveforms will look like those in Figures 11 and 12 on page 44. We start by defining the logic analyser, i.e. SignalTap.

1. In the Quartus II project “counter” (i.e. the project you created in this lab) create a new SignalTap file (.stp) by selecting **Tools→SignalTap Logic Analyzer**. This will bring up the SignalTap Logic Analyser window shown in Figure 6.
2. The first step is to inform SignalTap which nodes (i.e. signals) in top module are to be probed. I.e. which signals in counter.sv are to be connected to the inputs (i.e. logic probes) of SignalTap. Probes are assigned to nodes (i.e. signals) via the “trigger Control panel”, which is in the window pane that has the phrase “double click to add nodes” written inside it.
 - (a) Double click on the phrase “double click to add nodes”. This will bring up another window called “Node Finder” which is shown in Figure 7.
 - (b) The node finder will generate a list of the “nodes”, i.e. a list of signals, whose waveforms can be displayed and triggered upon. The nodes that it lists will depend the filter that is chosen as well as a couple of selections. A filter is necessary for two reasons:
 - i. There are usually a very large number of nodes in a circuit and it is helpful to limit the length of the list that has to be searched to find the signal of interest.
 - ii. The optimization process rearranges the circuit creating many new nodes (signals) and removing many of the nodes entered by the designer.

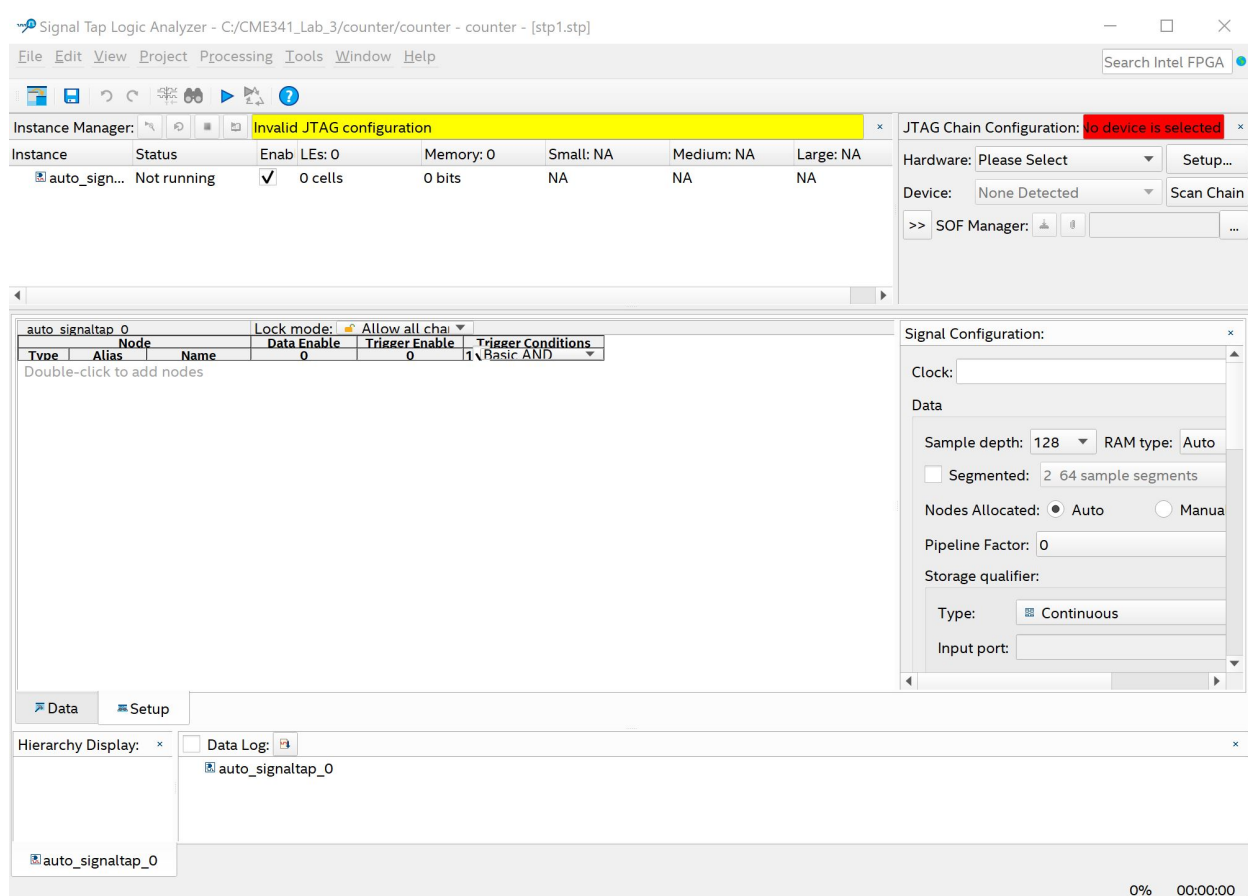


Figure 6: A screen capture of the SignalTap window (Quartus Lite V22.1)

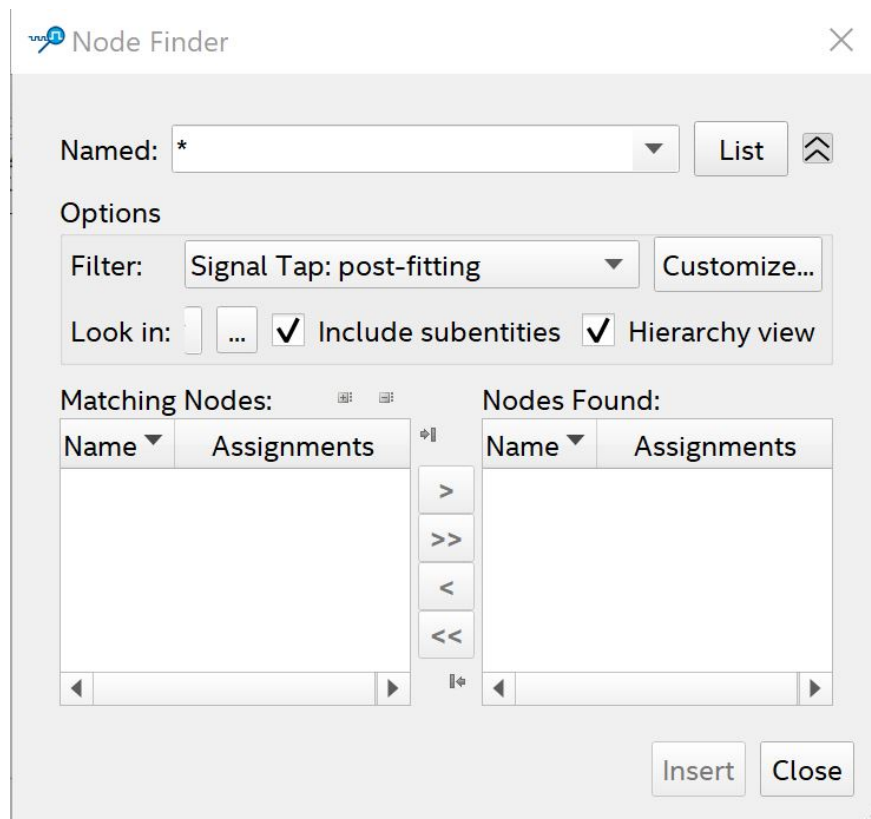


Figure 7: A screen capture of the node finder window before the fields are filled out (Quartus Lite V22.1).

The nodes created by the compiler are sometimes given cryptic names, which means after optimization it is normally difficult for the designer to recognize the signal names.

The filter is set up from a panel within the node finder, but that panel may be hidden when the node finder window first opens. The filter pannel is visible in Figure 7. The filter panel can be toggled between visible and hidden by the clicking on the button to the right of the “list” button. Once the filter panel is visible, make the filter “SignalTap: pre-synthesis”. Make sure the “Named” field has a “*” in it so that it will list all the pre-synthesis signals used in the design.

Also make sure the Hierarchy view box is not checked. Otherwise, you not be able to see the signal names.

- (c) Then click list.

This puts a two-column list of signals and pin names in the pane on the left, which is called the “Matching Nodes ” pane. It will list all the “pre-synthesis” nodes and, if they are connected to a pin, the pin number. Stretch the window both vertically and horizontally so you can see the two columns in the "matching nodes" pane. If you want to verify what your are seeing in the "Matching nodes" pane, look ahead to Figure 8 on page 40 to see what should be listed. Notice some signals were created by the compiler. For example cnt[0]~reg0.

- (d) Select (i.e. click on) cnt, which is vector for the counter output, ena, which is the count enable, and up_down. Then transfer them to the pane on the right called the “nodes found” pane. This is done by clicking on the “>” button or double clicking on the signal. **Do not select “clk”** as it will be used as the clock that “samples” the probed signals, i.e. the clock that transfers the samples into a shift register, and “clk” cannot sample itself.

If you have selected “clk” by mistake it can be removed by clicking on it in the “nodes found” pane and then clicking the “<” button.

At this point the node finder window should look like Figure 8

Once the signals have been transferred to the “nodes found” pane, click the button labeled “insert” in Figure 8.

- (e) The “nodes found” are transferred to a table displayed inside the SignalTap window. This table is referred to as the trigger control panel or trigger table for short. See Figure 9 on page 41.

Note: The signals in the first column of the trigger table, the column titled "nodes", are connected, i.e. hardwired, to SignalTap, specifically to SignalTap’s logic probe inputs.

3. Next, the “Signal Configuration” panel in the SignalTap window will be filled out. This panel is located to the right of the trigger table.

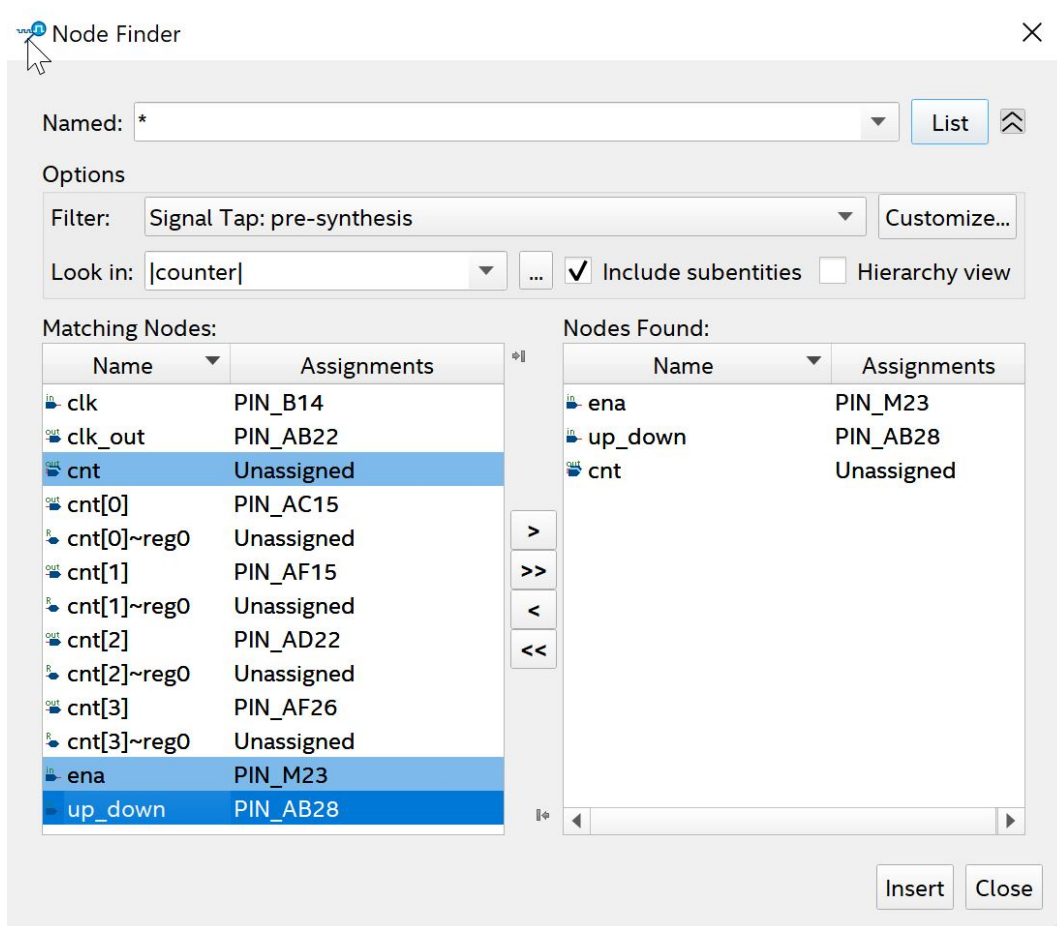


Figure 8: A screen capture of the node finder window after it has been filled out. (Quartus Lite V22).

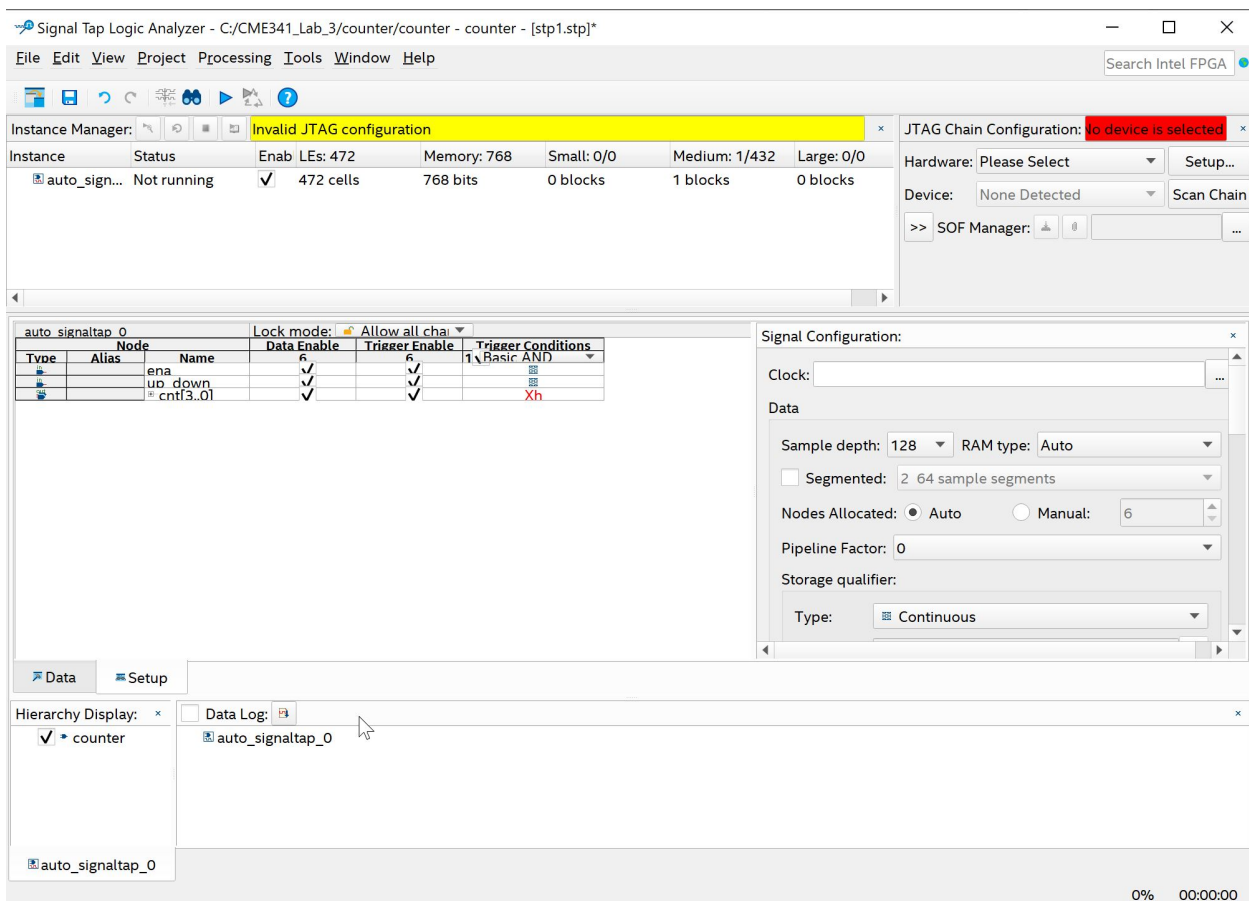


Figure 9: A screen capture of the SignalTap window showing the trigger table (Quartus Lite V22.1)










Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	6	6	1\Basic AND ▾
		ena	✓	✓	
		up down	✓	✓	
		cnt[3..0]	✓	✓	RRRRb
		cnt[3]	✓	✓	✓
		cnt[2]	✓	✓	✓
		cnt[1]	✓	✓	✓
		cnt[0]	✓	✓	✓

Figure 10: A screen capture of the the trigger table set to trigger when all four signals in cnt transition from 0 to 1 at the same time. Hopefully in later versions of Quartus the table will not display so that it doesn't appear to be vertically compressed (Quartus Lite V22.1).

- SignalTap needs a clock to clock the data samples into a shift register. The signal that will be used for that clock is assigned in the Signal Configuration panel. We want the signal clk in the verilog prototype to be the clock in SignalTap, so type clk into the box beside clock:.
 - Also, make the "Type:" box read "continuous".
 - Set the sample depth to 128. This tells SignalTap to collect 128 samples from each signal in the trigger table. Since each binary signal is collected by clocking it into a shift register, the sample depth defines the length of the shift registers.
Just to be perfectly clear on what "sample" and "sample depth" mean: A sample is the value of the signal at the time of the rising edge of clk. In essence, this sampling circuit is a 128-bit shift register for each signal in the trigger table. The sample depth is the length of the shift register.
 - Set the RAM type to "Auto". This gives SignalTap the freedom use the whatever RAM on the FPGA it wants to build the shift registers.
 - All other setting in the Signal Configuration panel should be the default settings.
4. Next, the trigger conditions in the trigger table will be set up. They do not have to be set up now. The can be, and usually are, set up after SignalTap has been compiled into the project and the .sof file has been programmed into the FPGA. We will set the trigger conditions to those shown in Figure 10.

Note: The trigger conditions can be changed after the project is compiled, loaded into the FPGA and running on the FPGA without having to recompile.

Before setting the trigger conditions, expand the vector cnt in the trigger table into its four binary signals by clicking on the + sign in front of it. Then set the trigger condition for signals cnt[0], cnt[1], cnt[2] and cnt[3] to rising edge. This is done by right clicking on grey-blue square symbols in the column titled "trigger conditions" that correspond to cnt[0],

cnt[1], cnt[2] and cnt[3] and then selecting “rising edge” for each of them. When you are done, the trigger table should look like Figure 10. This set of trigger conditions will cause SignalTap to trigger when, and only when, all four signals change from 0 to 1 on the same positive edge of clk.

Alternatively, the same trigger condition is set by writing RRRR (R means rising edge) in the “trigger conditions” corresponding to cnt.

NB: This trigger condition will only occur if the counter is counting down. There is never a transition from 0H to FH when the counter is counting up.

5. Save the .stp file then Click “yes” when prompted to enable SignalTap for the current project.

The prompt only appears the first time the .stp file is saved. However, SignalTap can be enabled or disabled anytime by selecting **assignments→settings...** . This will bring up the setting page for the project. Select **SignalTap Logic Analyzer** and then check or uncheck the enable box.

6. Recompile. (There will likely be one or more clock warnings. Ignore them for now.)
7. Configure the FPGA on the DE2-115 board. I.e. program the FPGA using file counter.sof.
8. Open the SignalTap GUI (**Tools→Signal Tap Logic Analyzer**). Under the JTAG Chain Configuration panel in the hardware box select USB-Blaster (USB-0). Click “setup” and select USB-Blaster. The Instance Manager should display Ready to acquire.
9. The command that starts the collection process is “**Run Analysis**”. Select **Processing → Run Analysis**. There is also a “**Run Analysis**” icon located in the instance manager bar.

This command instructs SignalTap to start sampling the “probed signals” with the rising edge of the specified clock (in this case “clk”). As explained earlier, SignalTap will clock the probed signals into shift registers. It will continue to clock in the probed signals, even if the shift register is full, until the trigger condition is encountered. It will not disable the shift register immediately upon finding the trigger condition. It will clock in another pre-specified number of samples to get post-trigger data and then it will disable the shift register, i.e. stop collecting data.

While SignalTap is searching for the trigger, it displays a message at the top of the instance manager window saying something to the effect “Acquisition in Progress”, which means “waiting for trigger”.

If the message “Acquisition in Progress”, (i.e. “waiting for trigger”), is displayed for more than a few seconds, then the transition from 4’H0 to



Figure 11: A screen capture of the full SignalTap display (data tab selected in the SignalTap window) (Quatus Lite V22.1).

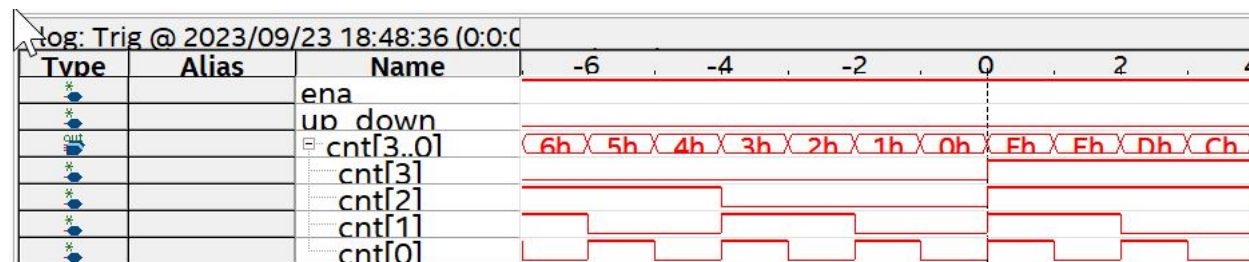


Figure 12: A screen capture of a zoomed SignalTap display (data tab selected in the SignalTap window) (Quatus Lite V22.1).

4'HF never occurs. The transition from 4'H0 to 4'HF will happen if the counter is counting down, but will not happen if the counter is counting up. If the message "Acquisition in Progress" persists, the counter must be counting up so the up-down input to circuit "counter" is high and it should be low. Since the up-down input to circuit "counter" is switch SW[0], slide SW[0] to the off position, which is toward the outside of the DE2-115 board, so the counter will count down.

10. Once you get SignalTap to trigger, look at waveforms it generates and verify the circuit is counting down. The counting sequence should be FH, EH, ..., 1H, 0H, FH, EH, The waveforms should look like Figure 11. A zoomed in view of the waveforms is provided in Figure 12.
11. The numbered 'ticks' in ruler at the top of the waveforms in Figures 11 and 12 mark the positive edges of clk. The numbering on the rules indicate the sample time (in clk periods) with respect to the vertical dashed line. The first sample is clocked into the shift register at the time indicated by the tick mark numbered -16 and the last sample clocked in at the tick mark numbered 112. In other words, the time axis is the elapsed time measured in clock periods from the vertical dashed line.
This is easily verified by looking at Figure 12 because we know the counter is counting down.
12. The vertical dashed line indicates the time of the rising edge of clk where the trigger condition was met. This is easily verified from Figure 12 where

it is clear the vertical dashed line marks the transition from 0H to FH, which is the trigger condition that was written into the trigger table.

13. The dashed vertical line coincides with tick mark 0 on the ruler at the top. Therefore, numbers on the tick marks indicate the time since the trigger condition was found.

14. **Note:** SignalTap can be re-triggered while the circuit is operational in the FPGA. Not only that, the trigger conditions can be changed. To refill the shift registers and get a new display, just re-run the analysis.

15. Slide the up_down switch to the count-up position. I.e. slide SW[0] toward the inside of the DE2-115 board to make up_down = 1'b1.

Then re-run the analysis.

SignalTap should not trigger as there should not be a transition from 4'H0 to 4'HF while the counter is counting up. The message "Acquisition in Progress" will be permanently displayed.

Slide the up_down switch to 1'b0, which is the count-down position. SignalTap should now trigger.

16. Now we are going to change the trigger table. Notice there are two tabs below the waveform display. One is called "data" the other is called "setup". These two tabs flip between the waveform display and the trigger table.

Select the "setup" tab to flip to the trigger table.

17. This time we are going to trigger on a value as opposed to a transition. We will trigger when cnt = 4'b1111 or, equivalently cnt = 4'HF. In this case, unlike triggering on rising edges, it doesn't matter how cnt got to be 4'HF. It could have transitioned from anything, for example it could have transitioned from 4'HE to 4'HF.

This trigger condition can be set by writing a 1 into the trigger condition for each of cnt[0], cnt[1], cnt[2] and cnt[3] or by writing "1111" or "F" in the trigger condition for cnt. Surprisingly, writing this particular trigger condition into cnt is a bit tricky. When the letter "F" is entered into the trigger table it can be interpreted as two different things. When used for a binary signal it means "Falling edge". When used with a vector it could mean either the hexadecimal number 4'HF (i.e. Fh) or a Falling edge. If simply "F" is entered for a vector, the GUI will append an "h" and change the entry to "Fh", which means SignalTap interprets the entry to be 4'b1111.

If "FFFF" was entered (note it is just FFFF without a "b" or "h" appended) SignalTap appends a "b" and enters "FFFFb". This would trigger when all four binary signals in cnt have falling edges.

Note: If the trigger condition for a vector is a value, for example Fh or, equivalently, 1111b, the trigger condition does not depend on the previous state. It only depends on what the value of the vector will be after the a positive edge of the clock.

After setting the trigger condition for cnt to 1111b, run the analysis twice: once with the counter counting up and once with the counter counting down and verify that SignalTap now triggers when counting up.

18. **Explaining the Autorun analysis command:** There is command that will repeatedly run the “run analysis” command. That command is “Autorun Analysis”. The “Autorun Analysis” command is the same as the “run analysis” command except that after the trigger is found and the waveform displayed, it waits for a moment and then re-runs the “run analysis” command. It does this until the square stop button on the instance manager bar is clicked upon.

The “Autorun Analysis” command is executed by selecting **Processing**→**Autorun Analysis**. There is also an icon (counter clock wise arrow) for this command located in the instance manager bar.

The **Autorun Analysis** command is terminated by selecting **Processing** → **stop analysis**. The square “stop” button icon located in the instance manager bar also terminates the **Autorun Analysis** command.

19. Make sure the analysis is stopped by selecting **Processing** → **stop analysis** or clicking on the square button on the instance manager bar.

Then select the “Setup tab” and change the trigger condition so all four binary signals in cnt are don’t cares. I.e. write xxxx for the trigger condition of cnt.

Next select **Autorun Analysis**. Doing this should cause Quartus to switch to the “Data” tab and display the waveforms. If it doesn’t, select the “Data” tab.

Notice that the display is continuously updating. Since the trigger condition is don’t care, the display triggers at different times in the waveforms and so the waveforms do not appear stationary.

Finally, press and hold KEY[0] to disable the counter (KEY[0] is the enable input and is active low). The display should will now show the signals as constant values (i.e. not changing), since the counter is disabled.

20. Now we will change the size of the shift register.

It is worth repeating that if anything is changed in the “Signal configuration” panel or if a node (i.e. signal) is added to the trigger table, then the Quartus project has to be recompiled and the FPGA has to be reprogrammed.

Now change the size of the shift register by doing the following:

- (a) Stop the analysis (i.e. click on the square stop button).

- (b) Select the “Setup” tab at the bottom of the signal display / trigger table pane. To be clear there are two rectangular “setup” buttons in the SignalTap window. One is a tab in the signal display / trigger table display pane. That is the rectangular button of interest here. The other rectangular button in the JTAG Chain Configuration pane, which is not the one of interest here.
- (c) Change the number of data samples to be collected to 1k (which changes the size of the shift register to be 1k) by changing the **Sample Depth** box to 1k. The **Sample Depth** box is in the signal configuration panel. Notice that when the number of samples in the **Sample Depth** box is changed, the information box in the instance manager bar turns red and says “Start Rapid Recompile to continue”.
It is pointed out that a complete recompile also works and has to be used in the case of Quartus Lite V22.1, because Quartus Lite V22.1 does not provide/allow a rapid recompile.
- (d) Save the SignalTap settings (File → save), re-compile and then reconfigure the FPGA on the DE2-115 board.

There is a bug in Quatus that has been around for years. After you recompile, the message part of the instance manage bar may turn red with a message that reads something to the effect "program the device before you continue". This is a legitimate message, but sometimes this message will not be removed after you re-programmed the DE2-115 board.

Sometimes you can ignore the message and proceed.

Sometimes Quartus won't let you proceed. In this case, click on the rectangular "Setup" box in the "JTAG Chain Configuration" panel and then double click on "USB-Blaster" and then close the window.

Sometimes after the red message "program the device before you continue" is fixed, Quatus will still not activate the run analysis icon on the instance manager bar. Should that happen close the SignalTap window and reopen by selecting “tools → SignalTap logic analyser”.

- 21. Run the analysis and verify that 1024 data points are displayed.
- 22. Play with the cursors. Start by right clicking on the bar above the signal display and selecting “insert time bar”. Set any time for the position of the bar - say 512.
- 23. Change the display format of vector cnt to unsigned decimal (Right click on cnt and select “Bus Display Format”). This makes it easier to verify the proper operation of the counter.

5.2.2 Controlled Triggering in Sequential Mode

SignalTap is a full featured embedded logic analyzer and as such has the ability to trigger in a variety of ways. The mode used in CME341 is Sequential Trigger

Mode with only a single trigger condition in the sequence. It is possible to set as series of trigger conditions, each of which has to be met in sequence before SignalTap will trigger.

None of the labs in the DSP stream require triggering with 2 or more sequential trigger conditions, so it is not necessary to learn how to set up a sequence of trigger conditions. For those that are interested, the instructions for setting up 2 or more sequential trigger conditions are given below.

1. First the signal configuration panel will be set up.
 - (a) Make sure the “Storage Qualifier: Type” box contains “continuous”.
 - (b) Scroll down in the signal configuration window until the “trigger” section is reached.

Make sure that the “trigger flow control” box is set to sequential.

Set the “Trigger Position” to “Center Trigger Position”. This setting determines how many samples are collected after the trigger is found. In this case 1/2 the buffer is filled after the trigger is found.

- (c) Set the “trigger conditions” box to 2. Doing this will generate another trigger column in the trigger configuration table.

When 2 trigger conditions are specified, SignalTap will first wait for the condition set up in column 1. After that condition is encountered, it will wait until the trigger condition in column 2 is met and then it will trigger.

2. Save the .stp file, re-compile and configure the FPGA on the DE2-115 board.
3. Now, perform a sequential trigger on cnt. Set up the trigger control panel to trigger on the first rising edge of cnt[2] after cnt makes the transition from 4’HF to 4’H0. Do this by setting of the first trigger condition to detect falling edges on all four signals, i.e. a trigger condition of FFFF for cnt, and the second trigger condition to detect a rising edge of cnt[2].

Run the analysis and observe the waveforms in the data tab to verify that you have set the trigger conditions correctly.

5.3 Exercise in Using SignalTap

It is highly recommended that you do this exercise.

The intent of this exercise is to review setting up and running SignalTap. First, a Quartus project has to setup. To minimize the work is setting up the project, the contents of all the files needed are provided.

1. Inside folder CME341_Lab_3, create a folder named counter_2.
2. Inside folder counter_2 create a new Quartus project named counter_2
3. Name the top module counter_2.sv.
4. The module counter_2.sv should have the following Verilog HDL.

```
module counter_2(  
    input clk,  
    output reg[9:0] cnt_2  
);  
  
always @ (posedge clk)  
    if (cnt_2 == 10'd8)  
        cnt_2 = 10'd0;  
    else  
        cnt_2 = cnt_2 + 10'd11;  
  
endmodule
```

5. Copy the Synopsys Design Constrains file for project counter, i.e. counter.sdc, into folder counter_2 and rename it counter_2.sdc. Then add it to Quartus project counter_2.
6. Make a file counter_2.csv with pin assignments as specified below.

To,	Assignment Name,	Value
clk,	Location,	PIN_B14
cnt_2[0],	Location,	PIN_AB22
cnt_2[1],	Location,	PIN_AC21
cnt_2[2],	Location,	PIN_AD11
cnt_2[3],	Location,	PIN_AD15
cnt_2[4],	Location,	PIN_AC15
cnt_2[5],	Location,	PIN_AC19
cnt_2[6],	Location,	PIN_AD19
cnt_2[7],	Location,	PIN_AF24
cnt_2[8],	Location,	PIN_AF25
cnt_2[9],	Location,	PIN_AF22

7. At the bottom of the Quartus message window select the "System" tab. If there are any messages in that tab, right click in the message window and clear the messages.

Then import the pin assignments from .csv file counter_2.csv.

Then observe the message printed in the message window. A segment of it should read: ... 11 assignments were written (out of 11 read) ...

NB: You always have to check to see if all of your signals were assigned pins. Errors in pin assignments are not flagged - they are just ignored.

NB: Importing a .csv file is only the first step in assigning pins. The pins are not assigned until the next time the project is compiled.

8. Select the “Processing” tab and then Compile the project.
9. Double check the pin assignments by opening the Pin Planner (assignments → **Pin Planner**) and checking the table at the bottom of the page to see that all 11 output signals have been assigned pins.
10. Program the FPGA on the DE2-115 board.
11. Set-up SignalTap to trigger on vector cnt_2.
12. Recompile the Quartus project.
13. Now the exercise questions: Use the triggering feature in SignalTap to answer the following four questions:
 - (a) Does the sequence of values in vector cnt_2 include 10’d1023? (answer: triggers so yes)
 - (b) Does the sequence of values in vector cnt_2 include 10’d1019? (answer: doesn’t trigger so no)
 - (c) Does the sequence of values in vector cnt_2 include 10’d1010? (answer: triggers so yes)
 - (d) Does the sequence of values in vector cnt_2 include 10’d1005? (answer: doesn’t trigger so no)

6 LAB 4: Designing a B-to-BCD Converter with Sequential Logic

Objectives and Learning Outcomes

The objectives of this lab are:

1. To learn how to translate a concept or an idea that involves an ordered sequence of actions into a sequential logic circuit.
2. To learn how to use the SignalTap logic analyser.
3. To learn how the SignalTap logic analyser can be used to help debug a sequential circuit.
4. To re-enforce that divide-and-conquer is the best approach to use in debugging. In other words a complicated circuit is best debugged by breaking it into well defined functions and debugging each function separately.

The learning outcomes of this lab are to be able to

1. Translate a concept or idea that involves an ordered sequence of actions into a sequential logic circuit.
2. Use the SignalTap logic analyser to debug a circuit.
3. Debug a sequential circuit using SignalTap.
4. Debug a circuit using the divide-and-conquer approach.

Pre-Laboratory Preparation

Preparation is not required.

The Design Problem

Design a sequential circuit that converts an 8-bit binary integer into a 3-digit decimal number, where the decimal number is represented in binary coded decimal (The binary coded decimal numbers are 4'b0000, 4'b0001, ..., 4'b1001.)

The architecture for the circuit you are to design is given in the top part of Figure 13. The critical block is the Binary-to-Binary-Coded-Decimal (B-to-BCD) converter, the architecture for which is shown in the bottom part of Figure 13. The B-to-BCD circuit consists of four counters. One of which is labeled **binary counter**. The other three are labeled **basic_BCD_counter**. The counter labeled **binary counter** is an 8-bit binary counter. The counters labeled **basic_BCD_counter** are single-digit Binary-Coded-Decimal (BCD) counters, which means they have a 4-bit output. The three BCD counters are wired together to make a 3-digit binary-coded decimal counter. The top **basic_BCD_counter** is the least significant digit of the 3-digit BCD counter.

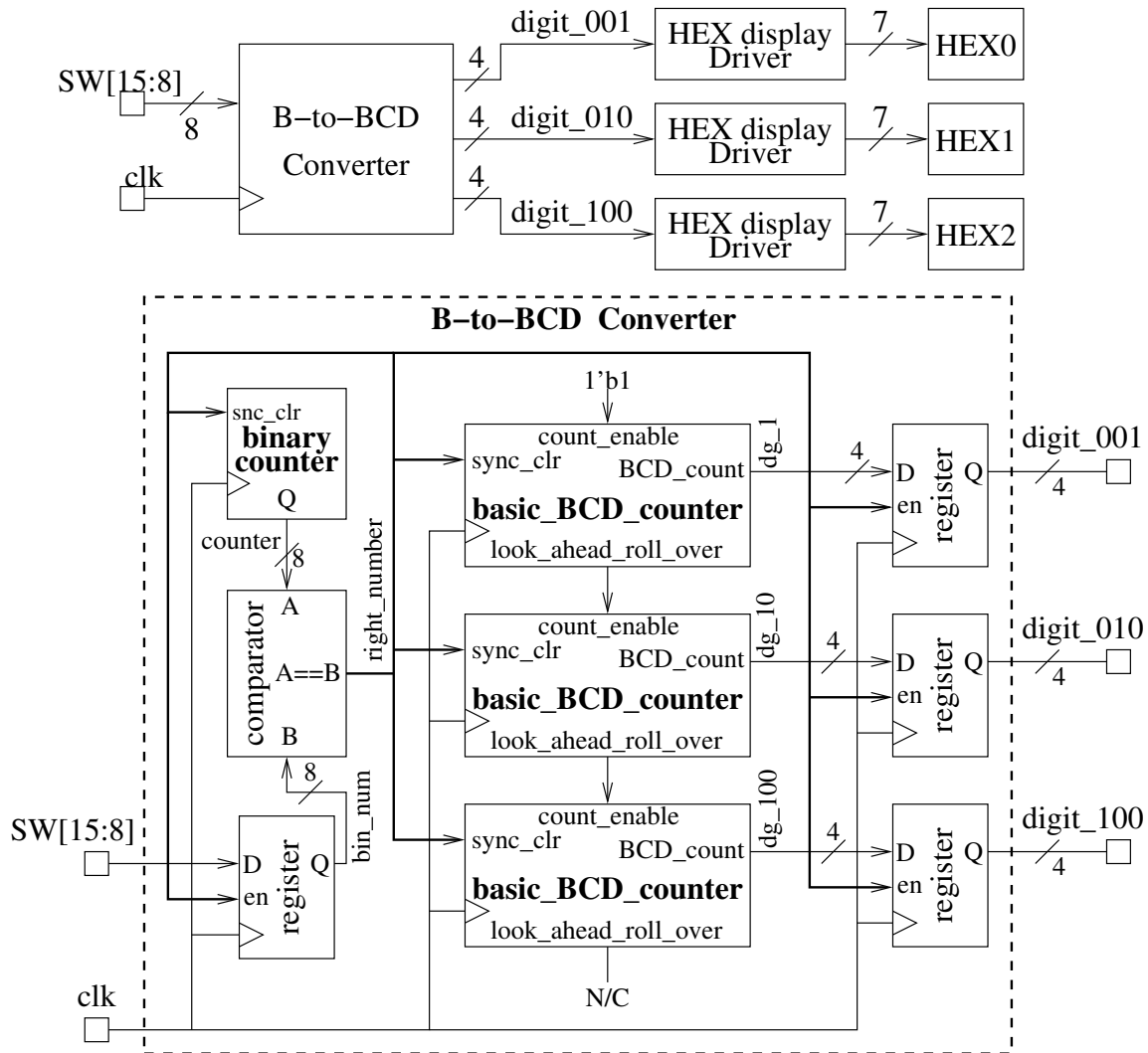


Figure 13: The Architecture for a Sequential B to BCD converter

The principal of operation is quite straight forward. Both the binary counter and the 3-digit BCD counter are reset at the same time and each is incremented by one in unison on a clock edge. This has the output of the 3-digit BCD counter being the BCD equivalent of the output of the binary counter. A comparator is used to indicate when the binary counter equals the binary number that is being converted. The clock edge that occurs while the comparator indicates equality is used to transfer the output of the BCD counter, which is the BCD equivalent of the binary number of interest, to three 4-bit registers. On that same clock edge both the binary counter and the 3 basic BCD counters are reset. Upon reset the conversion starts again.

The sequential conversion circuit is best debugged in stages. The procedure below leads the students through a reasonable design process that has a reasonable debugging strategy.

Procedure

1. Create a folder called CME341_Lab_4 and in that folder create a Quartus project called sequential_B_to_BCD_converter. Make the top entity for the project basic_BCD_counter
2. Design build and test a one digit BCD counter. Call the module basic_BCD_counter and place it file basic_BCD_counter.sv.

The function of a BCD counter can be described as follows. A BCD counter progresses by one on each positive going clock egde in the sequence 4'b0000, 4'b0001, ..., 4'b1001, 4'b0000, When the counter sits at 9, which is 4'b1001 in BCD, the next count is 0, which is 4'b0000 in BCD. The inputs and output of the BCD counter are described below.

- (a) It has a 4-bit output vector called BCD_count which is the binary coded decimal value of the base 10 digit.
- (b) It has a clock input called clk.
- (c) It has a synchronous clear input called sync_clr. If sync_clr is 1'b1 at the time of a rising edge of the clock, the counter is to be reset to 4'b0000 by that rising edge. If sync_clr is 1'b0 it has no effect on the counter.
- (d) It has an enable input called count_enable. If count_enable is 1'b1 at the time of a rising edge of clk then the counter is to increment by 1 on that rising edge unless the counter sits at 9, in which case it rolls over to 0 on that rising edge. If count_enable is 1'b0 at the time of a rising edge of clk then the output of the counter does not change.
- (e) It has a 1-bit output called look_ahead_roll_over. This indicates that the counter will roll over to zero on the next rising clock edge. That is to say, look_ahead_roll_over will be 1'b1 if and only if BCD_count is 4'b1001 and count_enable is 1'b1.

Clock the BCD counter with the on board 27 MHz clock that is connected to an FPGA pin (see pinout list in appendix for the pin number). Connect slide switch SW[9] to count_enable and slide switch SW[8] to sync_clr. Connect the outputs to any pins you prefer on the GPIO_0 extension board.

NB: Use SignalTap to debug the circuit and also to verify that it is operating correctly. Load all the signals except clk into the trigger table. Make clk the clock i.e. enter clk into the “clock” box in the “Signal Configuration” pane.

In the previous lab it was stated something to the effect

To generate a list of signal names in the node finder make sure the name field at the top of the window contains a “*” and the filter field contains “SignalTap: pre-synthesis” and then click the “list” button. This will generate a list of signals in the pane on the left called the “nodes found” pane. The signals in this list are the signals in the prototype Verilog HDL. Some of these signals may have been optimized out of the design by the compiler.

In the B-to-BCD design some of the user-entered signals that need to be observed will likely be optimized out. For example, the compiler’s optimization process will most likely remove the signal look_ahead_roll_over and implement the signal counter in a way its output is not available.

If one of the signals selected for the trigger table has been optimized out of the design it will still be entered into the trigger table, but it will be colored red.

The compiler’s optimizer can be forced to leave a signal in the design by using one of the Verilog synthesis commands (Synthesis commands were added in Verilog 2001.). The three synthesis commands of interest are: keep, noprun and preserve. Synthesis commands are enclosed in “ (* ” and “ *) ”. Of course, if these synthesis commands are used the circuits generated by the compiler will not be fully optimized. For that reason these commands are often removed after the circuit is debugged, but before it is tested.

To keep a wire, like look_ahead_roll_over, in your design place “ (* keep *) ” at the beginning of the declaration line. The declaration line is the line that declares the signal type. keep is also used for type reg, but only if the signal is the combinational logic i.e. the signal is made in an “always @ *” procedure.

To keep a register, i.e. a signal made with an “always @ (posedge clk)” procedure either (* noprun *) or (* preserve *) must be placed at the beginning of the declaration line.

The declaration line may be inside or outside the port list. For example, if look_ahead_roll_over and counter were declared in the port list, then

to stop the optimizer from removing them the declaration lines would be

```
(* keep *) output reg look_ahead_roll_over,
(* preserve *) reg[7:0] counter,
```

Of course the comma would not be at the end of the second line if it was the last entry in the port list.

Most of the time `(* preserve *)` and `(* noprune *)` do the same thing. The difference between them comes into play when the output of a register is not in any way connected to or in no way influences a pin on the FPGA. In this case the `(* preserve *)` command will not stop the compiler from optimizing the signal away, but the `(* noprune *)` command will. For example, if a counter is added to your design for purposes of debugging, say you want to display it in SignalTap to measure time, and it's output is left dangling, i.e. not connected to FPGA pins, then the `(* noprune *)` command must be used to keep the register in the design.

NB: Remember to assign all unused pins to “tri-stated inputs with weak pull-up resistors” and re-compiling before down loading the configuration file to the FPGA. Forgetting to do this can damage the either the FPGA and/or other chips on the DE2-115 Board.

3. Design and debug a 3-digit BCD counter. Put your Verilog HDL for the three digit counter in a module called `BCD_counter` and put that inside a file called `BCD_counter.sv`. You can create a new project or you can put `BCD_counter.sv` in the same project directory as `basic_BCD_counter.sv`, i.e. put it in the folder `CME341_Lab_4`. If you do the latter you must add `BCD_counter.sv` to the project and change the top entity to `BCD_counter.sv` before you compile so that `BCD_counter.sv` will be compiled. A Verilog HDL file can be made the top entity after it has been added to the project by opening the file in Quartus, pulling down the **Projects** menu and selecting the line that in effect says “set as top level entity”.

It is suggested that you design the 3-digit BCD counter by instantiating `basic_BCD_counter` three times, once for each digit. Remember that when a module is instantiated, the `.sv` file containing the module must be added to the project.

Call the outputs of the three basic BCD counters `dg_1`, `dg_10`, `dg_100`, `look_ahead_roll_over_1` and `look_ahead_roll_over_2`. Connect `look_ahead_roll_over_1` and `look_ahead_roll_over_2` to the enables of the next counter as shown in Figure 13, which is page 52. The enable for the basic BCD counter that is used for the least significant digit must be tied high as shown in Figure 13. Output `look_ahead_roll_over` in the instantiation

for the 100's digit is not used and therefore no signal should be named in the “()”. That is, the association in the connection list of the instantiation should be “.look_ahead_roll_over(), ”. Connect the outputs to any I/O pins on the GPIO_0 extension board.

NB: Use SignalTap to debug the circuit and also to verify that it is operating correctly. If you modified the existing project then the signals in SignalTap will have changed which means the signals in the trigger table will no longer exist. Delete all the signals in the trigger table and then rebuild the trigger table before compiling.

4. It is very important to be comfortable using SignalTap. To help with that please do the following:
 - (a) Remove SignalTap from your project. Open SignalTap and delete the instance from the instance manager pane. Save SignalTap. Then remove the .stp file from the project (project → add/remove files in project Recompile and verify that SignalTap has been removed.
 - (b) Regenerate SignalTap. Recompile and verify that it is again operational.
5. At times the pin assignments as well as other assignments get corrupted. This is more likely to happen when SignalTap is being used. The exercise below is the recovery process.
 - (a) Select **Assignments** → **remove assignments**. Then check the “all” box and click OK. This will remove all assignments, which include the device part number and the assignment of unused pins as “tristated with weak pull ups.”
 - (b) Reassign the part number.
 - (c) Reassign unused pins as “tristated with weak pull ups.”
 - (d) Reimport the .csv with the pin assignments.
 - (e) Recompile and reload your design into the DE2-115 board.
 - (f) Verify correct operation.
6. Construct the 3-digit binary to binary-coded-decimal converter shown at the top of Figure 13, i.e. display the BCD output on the hex displays. Call the module sequential_B_to_BCD_converter and place it in a .sv file named sequential_B_to_BCD_converter.sv. Put the file in the same project directory and make it the top entity.

Signals entered into SignalTap that are later removed either by the engineer or by the optimizing compiler need to be deleted from SignalTap.

6 LAB 4: DESIGNING A B-TO-BCD CONVERTER WITH SEQUENTIAL LOGIC 57

Please use the shell given below for the basis of your design. The Lab exams will be phrased in terms of the signal names used in this shell. Note that the instantiations for the hex display drivers have been included in the shell to save keyboard entry. Be sure to remember to copy file `hex_display_driver.sv` to your project folder and add it to your project.

```

module sequential_B_to_BCD_converter (
input[15:8] sw, // switches SW[15:8], binary num to be converted
input clk,
output wire[6:0] hex0, hex1, hex2 // segment drivers
);

reg[7:0] counter; // the binary counter
reg[7:0] bin_num; // the register holding the
                //binary number being converted
reg[3:0] digit_001, digit_010, digit_100;
                // registers holding the BCD digits
(* keep *) reg right_number; // useful for debugging
                // so don't let the compiler optimize it out
// declaring it a reg means the comparator
// must be built with the ‘‘always @ * ’’
wire[3:0] dg_1, dg_10, dg_100; // outputs of ‘‘basic_BCD_counter’’
                // instantiations

// comparator

// binary counter

// bin_num register

// registers digit_001, digit_010 and digit_100

// make the 3 digit BCD by the
// instantiation of the basic BCD counter 3 times

// instantiation of the hex display drivers
hex_display_driver hex_display_0 (
    .hex_digit(digit_001),
    .hex_segments(hex0) );
hex_display_driver hex_display_1 (
    .hex_digit(digit_010),
    .hex_segments(hex1) );
hex_display_driver hex_display_2 (
    .hex_digit(digit_100),
    .hex_segments(hex2) );

```

```
endmodule
```

7. After the 3-digit binary to binary-coded-decimal converter is debugged and working, read the compiler report and record the number of logic cells and flip/flops that were used. Compare the number of logic cells used in the sequential circuit to the number used in the combination circuit that was designed in a previous Lab.

7 LAB 5: Controller for Traffic Lights

Learning Outcomes

Upon completion of this lab the students should be able to:

1. Generate state diagrams for one-hot finite state machines.
2. Design a simple one-hot finite state machine.
3. Design a timer that can be synchronized by (i.e. controlled by) entrances to and exits from states in the finite state machines.
4. Set up and trigger SignalTap to gather information useful in debugging a one-hot finite state machine.
5. Design a simple traffic light controller.

Pre-Laboratory Preparation

estimated time required 1.5 hours

Read Part A of this lab and generate a state diagram for the one-hot finite state machine that implements the traffic light controller described. Also generate a description of your machine in Verilog HDL. Do not use the Questa Intel FPGA simulator to debug the Verilog HDL description prior to the lab. Part of the laboratory experience is to debug a live circuit. That is to say debug a configured FPGA. At the beginning of the Lab you are to hand in the state diagram and a print out of your Verilog HDL description.

Perspective and Organization

This is the first of a two lab sequence that builds a traffic light controller for an intersection very much like the intersection of College Drive and Cumberland Avenue. This lab has two parts. Part A constructs a controller for a simple set of traffic lights. The second part adds an advanced left left turn light and left turn sensor. The next lab builds on this one by adding walk lights.

This lab constructs a controller for the traffic lights at a simple intersection of two roads crossing each other at 90°. One road runs East-West the other runs North-South. The controller controls four sets of red, amber and green lights with one set having a advanced left turn light. One set of lights controls north-bound traffic, one controls east-bound traffic, one controls south-bound traffic and one controls west-bound traffic. The set of lights that controls the southbound traffic also has an advanced left turn arrow that is controlled by a buried magnetic sensor.

This lab makes no provision for pedestrians so it does not include walk/don't-walk lights. The walk/don't walk lights are added in the next lab.

7.1 LAB 5A: A Simple Traffic Light Controller

Procedure

Design a controller for the lights at a level crossing intersection where there are four sets of lights, one for each direction. The four sets of lights control northbound, southbound, eastbound and westbound traffic. Each set has three lights and each light in the set has a different color. The colors are red (R), green (G) and amber (A). One and only one light in each set is lit at all times. The controller has 12 outputs which may well be named:

northbound_green, northbound_amber, northbound_red,
southbound_green, southbound_amber, southbound_red,
eastbound_green, eastbound_amber, eastbound_red,
westbound_green, westbound_amber, westbound_red,

NOTE: There is no left turn arrow in this part of the lab.

Table 3 shows which lights in a set are lit as a function of time. The table was constructed under the assumption the controller will be implemented as a finite state machine. Table 3 indicates there are six states. For this simple intersection the progression is from state 1 to state 6 in sequence and then back to state 1. The time spent in each state is given in the table. For example, the machine stays in state 1 for 60 seconds and then leaves state 1 and enters state 2. It stays in state 2 for 6 seconds and then enters state 3 etcetera.

The table indicates the 12 outputs while the controller is in state 1 are: red lights for northbound and southbound traffic active (lit), green lights for eastbound and westbound traffic active, all other lights inactive (dark).

Table 3: The timing sequence for the four sets of red, green, and amber lights at a simple intersection

Duration in seconds	60	6	2	60	6	2
State	1	2	3	4	5	6
North Bound	R	R	R	G	A	R
South Bound	R	R	R	G	A	R
East Bound	G	A	R	R	R	R
West Bound	G	A	R	R	R	R

One 7-segment display will be used to display a set of lights. The three horizontal segments in the display will be used for the three lights. The top segment will represent the red light, the middle segment will represent the amber light and the bottom segment will represent the green light. The four 7-segment displays that will be used are shown in Figure 14.

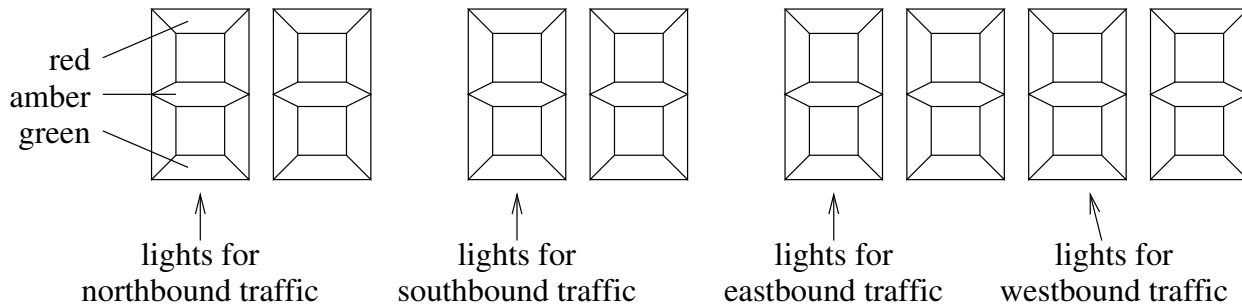


Figure 14: The 7-segment displays that are to be used for the lights

Use the right-most push button on the DE2-115 board as a reset. The controller must enter state 1 on reset.

Please use a 6-bit counter called “timer” to determine the length of time spent in each state. Use the value of “timer” as the time in seconds (i.e., holds values from 0 to 63 seconds). The clock that drives “timer” is to have a one second period so that timer can only change at one second intervals. The counter called “timer” is used to determine when to leave one state and enter another. For example, when entering state 1 the “timer” is set to 60. It is decremented by 1 at one second intervals until it reaches 1. One second after “timer” reaches 1, state 1 is exited and state 2 entered. When state 2 is entered “timer” is set to 6 and then decremented at one second intervals until it reaches 1, after reaching 1, state 2 is exited and state 3 entered, etcetera.

There are two on-board clocks that are connected to the FPGA. In this project it is probably best to use the 27 MHz clock. This clock will be referred to as “clk_27”.

From this clock you are to make a 1 Hz clock called “clk”. Clock “clk” is to drive the finite state machine. The 1 Hz clock can be constructed from a flip/flop that is clocked with “clk_27”. The “clk” flip/flop is controlled with logic to make it toggle every $27 \times 10^6 / 2$ cycles of “clk_27”.

The counter called “timer” could then be built with a Verilog description something like

```
always @ (posedge clk or posedge reset)
if (reset == 1'b1)
    timer <= 6'd60; // time for state 1
else if (entering_state_1 == 1'b1)
    timer <= 6'd60; // time for state 1
else if (entering_state_2 == 1'b1)
    timer <= 6'd6;  // time for state 2
else if ...
```

```

else if (timer == 6'd1)
    timer <= timer; // never decrement below 1
else
    timer <= timer - 6'd1;

```

The cycle time for the lights is 136 seconds, which is quite long. Such a long cycle time can drag out the debugging time. It is desirable to have a debugging mode in which the frequency of “clk” is increased by a factor of 10 to become 10 Hz. Please use slide switch 9 on the DE2-115 board to choose either normal operating mode or debug mode.

The “state” for this finite state machine can be held in a 3-bit register as there are only 6 states and a 3-bit register hold 8 different numbers. The value in the register would indicate the state. Alternatively, 6 flip/flops could be used, one for each state. When one flip/flop per state is used, only one flip/flop is active at any point in time as the controller can only be in one state at any point in time. Because only one flip/flop is active at one time, such machines are called one-hot coded machines. One-hot coded machines have the advantage that each state can be easily defined in a separate always statement. For example state 2, which is defined by a flip/flop, say “state_2”, could be defined by

```

// make the state 2 flip flop
always @ (posedge clk or negedge reset_bar)
if (reset_bar == 1'b0) // keys are active low
    state_2 <= 1'b0;
else
    state_2 <= state_2_d;

//logic for entering state 2
always @ *
if( (state_1 == 1'b1) && (timer == 6'd1) )
    entering_state_2 <= 1'b1;
else
    entering_state_2 <= 1'b0;

//logic for staying in state 2
always @ *
if( (state_2 == 1'b1) && (timer != 6'd1) )
    staying_in_state_2 <= 1'b1;
else
    staying_in_state_2 <= 1'b0;

// make the d-input for state_2 flip/flop
always @ *
if( entering_state_2 == 1'b1 )
    // enter state 2 on next posedge clk
    state_2_d <= 1'b1;
else if ( staying_in_state_2 == 1'b1 )
    // stay in state 2 on next posedge clk

```

```
        state_2_d <= 1'b1;
else      // not in state 2 on next posedge clk
        state_2_d <= 1'b0;
```

One-hot finite state machines are much easier to modify and expand. Their nature allows them to be easily subdivided and be designed by a team of engineers working independently on different parts of a finite state machine. This is a major reason why one-hot coding is used in large finite state machines.

It is strongly recommended that you use one-hot coding in your design for two reasons. The first is that this type of coding is preferred in industry and the second is that it is much easier to add states and modify the operation of the machine, which will be required as you progress through this lab and write the lab exam.

It is also recommended the combinational logic that determines the on/off status of each light (e.g. eastbound red) be in a separate always statement. For example the logic that determines the status of the red light controlling eastbound traffic could be

```
always @ *
if ( (state_3 | state_4 | state_5 | state_6) == 1'b1 ) eastbound_red = 1'b1;
else eastbound_red = 1'b0;
```

It is nearly always easier to debug a circuit when it is described in one-output-per-always-statement style. The reason for this is rooted in the way an engineer debugs a circuit. The engineer checks the outputs to see if they function properly. If one output does not, then the only place to check is the always statement that constructed the output.

7.2 LAB 5B: Adding a Left Turn Arrow to the Traffic Lights

Instructions

Design a controller that controls the lights at an intersection that, in addition to the normal red, green and amber traffic lights, has an advanced left turn arrow light for southbound traffic. The left turn arrow is not lit as part of the regular light cycle. It is only lit if the left turn sensor is active at the end of state 3. Table 4 shows the states and duration of each state for controller for lights. Note that there is now one more state, which is state 4a. The progression is ... state 3, state 4a, state 4, ... if the left turn sensor is active at the end of state 3. The progression is ... state 3, state 4, ... if the sensor is not active. Note that the left turn light, which is indicated as <- in Table 4, is

Table 4: The timing sequence for intersection with an advanced left turn

Duration in seconds	60	6	2	20	60	6	2
State	1	2	3	4a	4	5	6
North Bound	R	R	R	R	G	A	R
South Bound	R	R	R	<-	G	A	R
East Bound	G	A	R	R	R	R	R
West Bound	G	A	R	R	R	R	R

only lit when the controller is in state 4a.

The intersection in question differs from the one in Part A of this lab in that it has a sensor buried in the pavement that produces a binary logic signal. The sensor is buried in the left turn lane for automobiles approaching the intersection from the north and wishing to head east. It is active when an automobile is stopped in the left turn lane waiting to turn left and head east. In practice, the left turn sensor consists of two parts: a coil of wire buried just beneath the pavement in the left turn lane and an electronic circuit that measures the inductance of the coil. If an automobile is over the coil, the presence of the iron increases the inductance of the coil beyond some preset threshold and the sensor output goes active to indicate there is an automobile in the left turn lane. It would be reasonable to name the sensor output “left_turn_request”. For purposes of this laboratory “left_turn_request” will be generated by a push-button on the DE2-115 board, the one that is next to the reset button (i.e. the second button from the right).

The left turn arrow commonly used in traffic lights is a three segment display much like <-. Such a display can not be replicated on a 7-segment display. For this lab a sort of arrow-like pointer is constructed from the 7-segment display by lighting the two right most vertical segments and the middle horizontal segment

to get the sort of arrow-like display $-|$. The left turn arrow controls the southbound traffic so should be added to the HEX display representing the lights that control southbound traffic.

8 LAB 6: Controller for Traffic and Pedestrian Lights

Learning Outcomes

Upon completion of this lab the students should be able to:

1. Generate state diagrams for complicated one-hot finite state machines.
2. Design, build and debug complicated one-hot finite state machines.
3. Design a timer that can be synchronized by (i.e. controlled by) entrances to and exits from states in the finite state machines.
4. Design and incorporate additional circuitry that is used only in the debugging process to facilitate debugging.
5. Set up and trigger SignalTap to gather information useful in debugging a one-hot finite state machine.
6. Design a complete traffic light controller with walk/don't-walk lights and advanced or retarded left turns.

Pre-Laboratory Preparation

Time required: 1.5 hours

Read the laboratory then generate a state diagram for your revised design. Write a Verilog HDL that implements your state diagram.

You are to hand in your state diagram and your Verilog HDL description of the state diagram upon entering the Laboratory.

Instructions

This part of the Laboratory builds on the previous two laboratories. You may be asked in the lab exam to reproduce the designs you did in the previous two laboratories so it is important that you leave the project files for those two labs in tact. Feel free to copy any of the files to your project for this lab.

Design a controller for a set of lights that controls both automobile and pedestrian traffic. The set of lights in question is the same as those in the previous two laboratories except they now include walk and don't-walk lights. The walk light is either lit or dark, but the don't walk light has three modes: lit, flashing on-off, and dark. Including the walk and don't-walk light has added 6 states as shown in Table 5. These are states 1w, 1fd, 1d, 4w, 4fd, and 4d. There are three new symbols used in Table 5: 'W' for walk light lit, 'D' for don't-walk

light lit and ‘FD’ for a flashing don’t-walk light.

NOTE: When the don’t walk light is flashing it is flashing with 1 second cycle: 0.5 seconds on - 0.5 seconds off in regular mode and flashing with a 0.1 second cycle in debug mode (i.e. 0.05 seconds on - 0.05 seconds off).

Table 5: The timing sequence for intersection with walk lights and an advanced left turn

Duration in seconds	10	20	30	60	6	2	20	10	20	30	60	6	2
State	1w	1fd	1d	1	2	3	4a	4w	4fd	4d	4	5	6
North Bound	R	R	R	R	R	R	R	G	G	G	G	A	R
NBnd Walk	D	D	D	D	D	D	D	W	FD	D	D	D	D
South Bound	R	R	R	R	R	R	<–	G	G	G	G	A	R
SBnd Walk	D	D	D	D	D	D	D	W	FD	D	D	D	D
East Bound	G	G	G	G	A	R	R	R	R	R	R	R	R
EBnd Walk	W	FD	D	D	D	D	D	D	D	D	D	D	D
West Bound	G	G	G	G	A	R	R	R	R	R	R	R	R
WBnd Walk	W	FD	D	D	D	D	D	D	D	D	D	D	D

These lights also include 4 push-buttons, one at each corner of the intersection. These push buttons are wire-ored together to produce a single logic signal that goes high while any of the buttons are pushed. A pedestrian must push one of these buttons to get a walk light. The controller sets a flip/flop when any of the push-buttons are pressed i.e. the signal resulting from the or of the 4 push-buttons is high. A good name for this flip/flop would be ‘walk_request’.

The logic for introducing the walk light into the light cycle is straight forward. If ‘walk_request’ is active at the end of state 6 then the progression is ... state 6, state 1w, state 1fd, state 1d, state 2, ... If ‘walk_request’ is not active at the end of state 6 then the progression is ... state 6, state 1, state 2, ... Note that states 1w, 1fd, and 1d replace state 1 when ‘walk_request’ is active at the end of state 6.

The controller clears flip/flop ‘walk_request’ upon entering a walk state which is either state 1w or state 4w. If a walk button is pressed immediately after state 1w is entered then flip/flop ‘walk_request’ is set again. Setting ‘walk_request’ will cause state 4w to be entered at the appropriate time in the cycle.

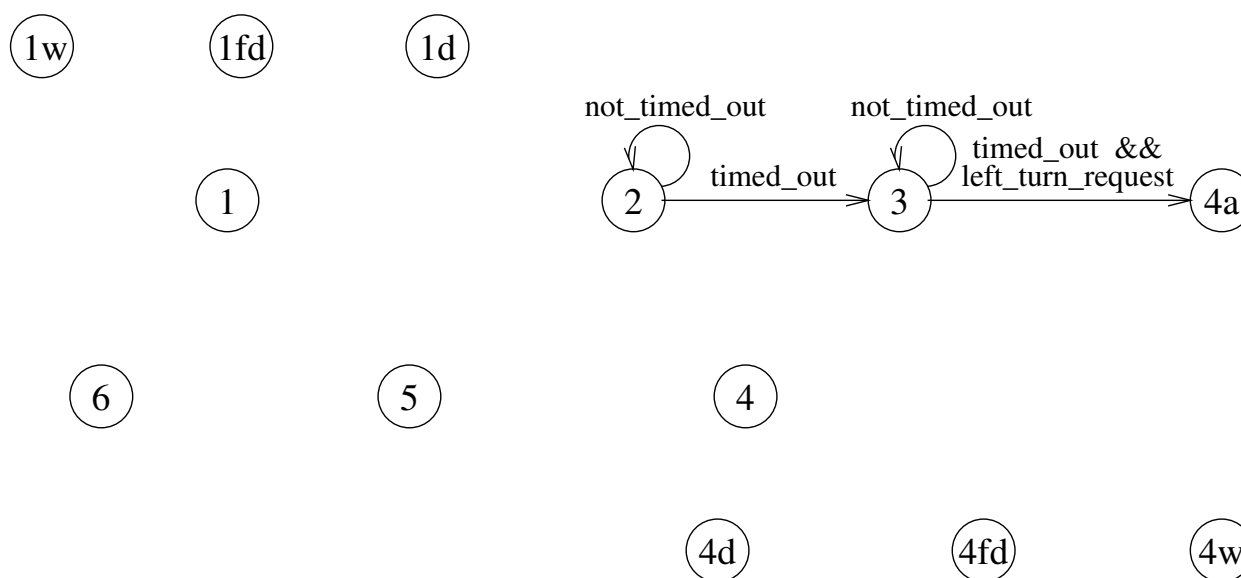


Figure 15: Skeleton state diagram for a controller with walk lights

The same logic applies to replace state 4 with state 4w, 4fd and 4d except it is slightly more complicated because state 4 can be entered from state 3 or state 4a. If “left_turn_request” is not active at the end of state 3 and “walk_request” is active at the end of state 3, then state 4w is entered instead of state 4. If “walk_request” is active at the end of state 4a, then state 4w is entered instead of state 4. Again, the controller clears flip/flop “walk_request” upon entering state 4w.

A skeleton state flow diagram is given in Figure 15. Please complete the diagram to make sure you understand the operation of the controller you are designing. If you are not certain how to complete the state diagram, please ask the lab instructor or the professor in charge of the class for clarification before proceeding.

Push buttons on the DE2-115 boards will be used for walk request push-buttons. Since there are only 4 push buttons on the DE2-115 board and 2 of them are already in use only the two remaining push-button on the DE2-115 board (i.e., the two leftmost push buttons) will be used for walk request buttons.

The walk and don’t lights will be displayed on a separate seven segment display. The upper left vertical segment will be used for the walk light and the 5 segments that make a ‘d’ will be used for the don’t walk light. The 7-segment displays that are to be used for the different directions are shown in Figure 16.

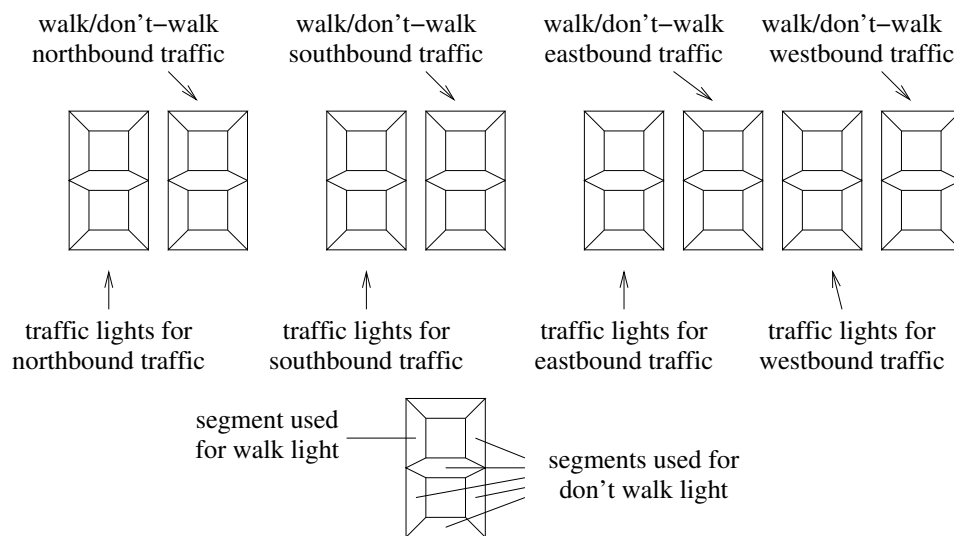


Figure 16: The 7-segment displays that are to be used for the traffic and pedestrian lights

9 LAB 7: Electronic Card Lock for Hotel Room Doors

Objectives and Learning Outcomes

The objectives of this lab are to:

1. Strengthen the students ability to design a circuit from a functional description that does not explicitly show the architecture in block diagram form.
2. Strengthen the student ability to debug circuits and design special circuits to be used only while debugging.
3. Learn how to describe a circular shift in a register in Verilog HDL.
4. Learn how to find the period of pseudo random sequences.
5. Learn how to generate a maximum length pseudo random sequence from a Linear Feedback Shift Register (LFSR).
6. Learn how to determine if the pseudo random sequence is indeed a maximum length sequence.
7. To get experience (The word “experience” is used here to mean “learn many things that can not be precisely identified and therefore can not be described”.) in converting a concept described in words to a well defined and well structured block diagram.
8. To learn the value of a block diagram.
9. To learn how to annotate a block diagram in sufficient detail for it to be constructed in Verilog HDL.
10. To learn that engineers learn as much from their mistakes as they do from their successes (i.e. To learn that design skills improve with experience.)

The learning outcomes are for the students to be able to

1. Design a fairly complex circuit from a functional description that does not explicitly show the architecture in block diagram form.
2. Design special circuits that are incorporated into the main circuit that facilitate debugging.
3. Describe a circular shift register in Verilog HDL.
4. Find the period of pseudo random sequence.
5. Design a circuit that generates a maximum length pseudo random sequence from a Linear Feedback Shift Register (LFSR).
6. Determine if a pseudo random sequence is indeed a maximum length sequence.

7. Convert a concept described in words to a well defined and well structured block diagram.
8. Annotate a block diagram in sufficient detail for it to be constructed in Verilog HDL.
9. Learn from their mistakes as well as from their successes.

Pre-Laboratory Preparation

Time required estimated to be 3 hours.

1. Read from the beginning to the end of the subsection on Linear Feedback Shift Registers (LFSR).
2. Make a folder called LFSR and in that folder create a Quartus project called LFSR. Create a verilog HDL file called LFSR.sv and in that file write a Verilog HDL module that describes the block diagram of the LFSR. Do not debug the Verilog HDL. That will be done in the Laboratory.
3. Continue with the careful reading to the end of the subsection on emulating the card reader.
4. Generate a schematic diagram (i.e. gate level diagram) for the card reader emulator and write the corresponding Verilog HDL description. Do not debug the Verilog HDL. That will be done in the Laboratory.
5. Read the remainder of the Lab carefully.
6. Generate a block diagram for the guest portion of the electronic lock. That is, assume the maid card and maid reset cards do not exist.
7. Write the Verilog HDL for your block diagram of the guest portion of the card lock.

The block diagrams and Verilog HDL done to this point are to be assembled into a single .pdf file and submitted to the Laboratory drop box prior to or upon entering the laboratory.

Partitioning the Problem

In this lab the student will design, build and test a circuit that mimics the operation of an electronic door lock that is often used on the guest rooms in hotels. To do that the students must understand, design, build and debug:

1. A Linear Feedback Shift Register (LFSR) circuit.
2. An emulator for the card reader.

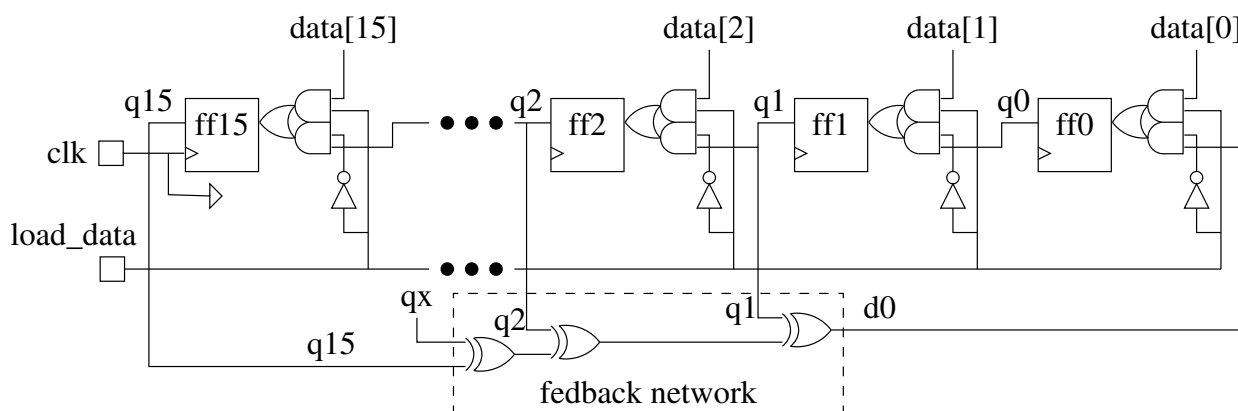


Figure 17: Linear Feedback Shift Register with synchronous Load

3. The circuit that surrounds the LFSR to make the electric card lock.

Each of these are done in separate subsections

9.1 PART A: Linear Feedback Shift Registers

Objectives for LFSR

The objectives are

1. To learn how to describe a circular shift in a register with Verilog HDL.
2. To learn how to find the period of pseudo random sequence.
3. To learn how to generate a maximum length pseudo random sequence from an LFSR.
4. To learn how to determine if a pseudo random sequence is indeed a maximum length sequence.

Theory of LFSRs

Figure 17 shows the schematic diagram for a linear feedback shift register with a synchronous load. The circuit consists of 16 flip/flops labeled $ff_0, ff_1, \dots, ff_{15}$ with outputs labeled q_0, q_1, \dots, q_{15} respectively. All flip/flops are clocked with a square clock called clk (not shown in Figure 17 to reduce the clutter). The data vector $data[15:0]$ is synchronously loaded into the flip flops on the rising edge of clk if $load_data$ is active. Otherwise the circuit acts as a shift register. In shift register mode q_n is shifted to q_{n+1} and d_0 is loaded into q_0 on each positive going clock edge.

The feedback circuit used in LFSRs is either a network of xor gates or a network of xnor gates. The LFSR in this lab uses a network of xor gates. In other words d_0 is the output of a network of xor gates, or equivalently the output of a multi-input

xor, with the inputs being taps from the shift register. If the appropriate set of taps are used in the xor network then q_{15} becomes a periodic binary sequence with a very long period. For a certain selection of taps the period is $2^N - 1$, where N is the number of flip/flops in the shift register. Such sequences are called maximal length sequences and q_{15} is a nearly random binary sequence, which means each bit in the sequence could have essentially been generated by flipping a coin.

The feedback circuit in Figure 17 has four inputs: q_{15} , q_2 , q_1 and some unknown tap referred to as q_x . This set of taps will make q_{15} a maximum length sequence if the correct tap is chosen for q_x . If some other tap is chosen for q_x the sequence will have a shorter period. Later in this lab you will be asked to find the tap for q_x that makes q_{15} a maximum length sequence, i.e. a sequence with period $2^N - 1$.

Maximal length sequences have two important properties that will be used to advantage in the design of the electronic lock.

1. A set of 16 flip flops can represent 2^{16} numbers, but the period of the maximal length sequence is $2^{16} - 1$. This means every possible number 16-bit number except one appears in the 16-bit register at some point in the sequence. The number that does not appear is 16'H0000.
2. The feedback network consists entirely of exclusive-ors. If all the inputs to a network of exclusive-ors are 0, then the output must be zero. This means if $[q_{15}, q_{14}, \dots, q_0]$ equals 16'H0000 then all the inputs to the exclusive-or feedback network are 0 and *do* must be 0. This implies if $[q_{15}, q_{14}, \dots, q_0]$ equals 16'H0000 before a shift it will be 16'H0000 after the shift and therefore remain 16'H0000 until it is synchronously loaded with some other value.

Clearly 16'H0000 must be the one number that does not appear in a maximal length sequence.

Helpful Hint: The concatenation operator, which is $\{ \}$, is useful for building shift registers when the input is a variable. Suppose the 16 flip/flops in LFSR are called $q[15:0]$, where $q[15]$ is the most significant bit, then the shifting part of the circuit can be build by

```
always @ (posedge clk)
q = { q[14:0], d0 } ;
```

Procedure for LFSR (i.e. pseudo random sequence generator)

See the Pre-Laboratory Preparation subsection for instructions on what to do in advance of the entering laboratory.

1. Create a Quartus project called LFSR for a module called LFSR.
2. Write a Verilog HDL module for the LFSR of Figure 17. Start by trying $q_x = q_3$. The inputs/outputs are to be

- Input clk: The 27 MHz clock.
- Input load_data: key 2, but inverted. Remember the keys are active low so an inverter must be placed between key 2 and load_data.
- Output q[15]: a pin of your choosing on the GPIO_0 connector.

Make the data to be loaded the constant 16'HFFFF, i.e. assign data = 16'HFFFF.

3. Set up SignalTap to display the 16-bit vector q and load_data. After loading the signals into the trigger table group the 16 signals q[15], q[14], ..., q[0] into a vector as follows:

- (a) Highlight the set of 16 signals that make up q[15:0].
- (b) Right click on the selected signals and select group.
- (c) Right click on the group name and select "Bus Display Format" and choose "Hexadecimal" as the radix.
- (d) Define the position of the LSB so the hexadecimal display knows which is the LSB. This is done as follows:
Click the "+" beside the group name to drop down the list of signals. Observe the location of the least significant bit then right click on the group name. Select "LSB on top, MSB on bottom" or "MSB on top, LSB on bottom", which ever is appropriate.

4. Configure the FPGA on the DE2-115 board. Then verify correct operation using SignalTap.

Note: An LFSR that generates a maximal length sequence never enters the state q=16'H0000. If the state q=16'H0000 is entered on start up, then the LFSR stays in that state. Pressing key_2 to load 16'HFFFF into q will get the LFSR working properly.

5. Add the 16-bit counter described below to the LFSR module. This counter will be used to measure the period of the sequence generated by the LFSR.

Add a 16 bit counter that increments by one on each clock edge except on the clock edges that occur when the flip/flops of the LFSR have value 16'HFFFF, i.e. when q == 16'HFFFF. The counter is to be set to 16'H0001 on clock edges occurring when q == 16'HFFFF.

If a circuit does not in some way affect the signal on a pin of the FPGA the Quartus compiler will optimize the circuit away. Since the counter does not in any way affect a pin on the FPGA it will be removed by the optimizer.

There are two ways to prevent Quartus from optimizing the counter out of the circuit:

- (a) Assign the most significant bit of the counter to a pin on the FPGA (e.g. PIN_AE22 which is LEDG[0]).

- (b) At compile time assign the synthesis attribute ‘noprune’ to the counter in the variable declaration as follows:

```
(* noprune *) reg[15:0] counter;
```

6. Revise SignalTap to display the output of the counter. Load the revised LFSR circuit, which includes SignalTap of course, into the FPGA.
7. Trigger SignalTap when $q = 16'HFFFF$ and observe the value in the counter at the time of the trigger. This will be the maximum value of the counter which is the period of the sequence generated by the LFSR. Record the value. If the period is $16'HFFFF$ the sequence is a maximal length sequence.
8. Change your Verilog HDL description of the LFSR so that q_x becomes q_4 . Recompile, reload the configuration file and remeasure the period.
9. Find the tap for q_x that produces a maximum length sequence and use that tap in your design.

9.2 PART B: Emulation of Card Reader

Objectives

There are no learning objectives for this part of the lab. The objective is to build a circuit that is necessary to complete the lab.

General Description

A door with a card reader and electric lock mounted on it is shown in Figure 18. To open the door the guest quickly jabs the key card into the slot on the top of the card reader. The card reader reads the magnetic strip on the card and then passes that information to the electronic lock mechanism housed inside the same box as the card reader in a format that will be described later. If the combination passed to the electronic lock is valid, the electronic lock trips the locking mechanism and frees the door handle to turn. The guest can then turn the door handle and enter the room.

This part of the lab emulates the card reader portion of the box shown in Figure 18. The magnetic strip carries an 18 bit-number, 16 of which are the combination and two of which carry other information that will be explained later. An actual card reader would read the 18-bit number from the magnetic strip and places it in a register of 18 flip/flops. The output of the register would be available to the electronic lock. After the number is placed in the register the card reader would generate a pulse whose rising edge would be used to transfer the contents of the register to another 18-bit register in the electronic lock called `card_number`.

In this lab there is no card reader. The number that would normally be put in the 18-bit register will taken from the 18 slide switches.

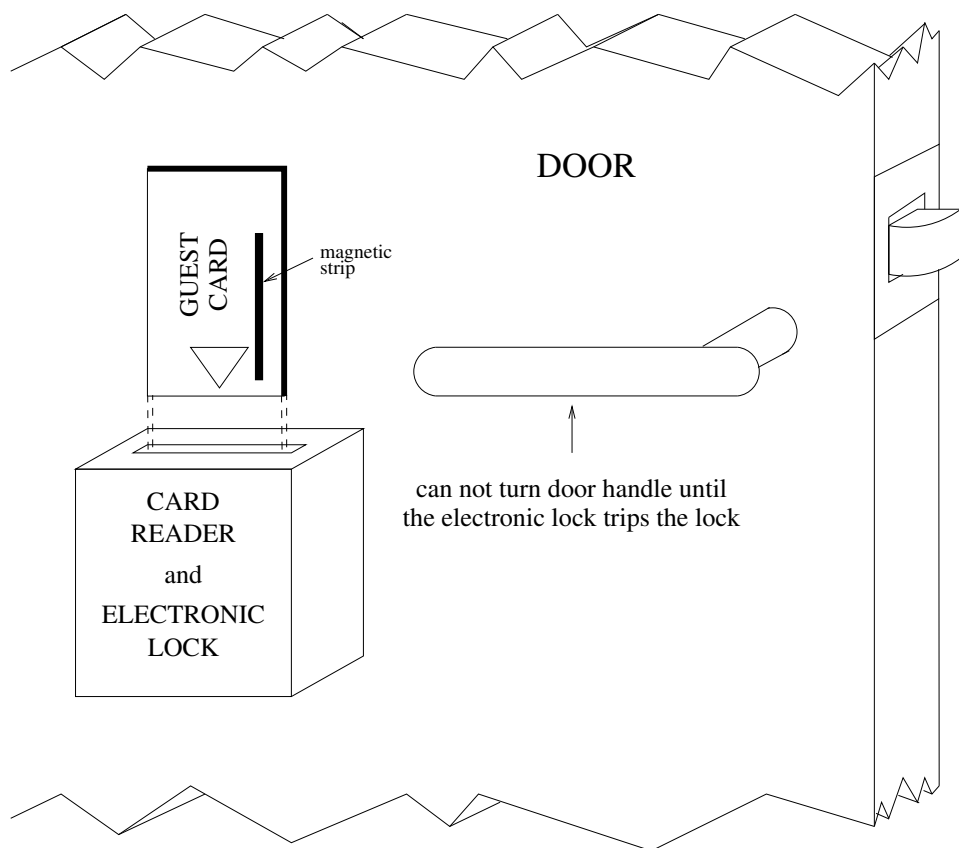


Figure 18: A door with an electric lock mounted on it

It is mentioned now, but will be explained later, that the rising edge of the pulse is also used to clock the LFSRs and any other flip/flops in the electronic lock.

The duration of the pulse in commercially available locks is about 2 seconds so that it can be used to drive the tripping mechanism that releases the door handle. This gives the guest 2 seconds to open the door. In this lab the pulse will be generated by a set-reset flip/flop under control of push buttons on the DE2-115 boards so its duration will not be any particular time.

Emulating the reading of the card and generating the pulse is actually quite trivial. The register will be switches `sw[17:0]`. The pulse, which will be referred to as `card_read`, will be done using a set clear flip/flop with the setting and clearing done by two push-button keys.

Procedure

See the Pre-Laboratory Preparation subsection for instructions on what to do in advance of the entering laboratory.

1. This subsection of the lab is the beginning of the electronic lock design. The Quartus project created in this subsection of the lab will be expanded in the next part of the lab to include the entire electronic lock.

The construction of the LSFR in Part A is a separate project and not part of the electronic lock design. In fact the Verilog HDL generated in Part A should not even be instantiated in the electronic lock.

Since the register that holds the card number is taken to be the 18 slide switches on the DE2-115 board, only the `card_read` pulse needs to be constructed in this part.

2. Create a new directory and a Quartus project called `electronic_card_lock`. Make the top entity for the project the file `electronic_card_lock.sv`.
3. The module declaration and port list should be as given below:

```
module electronic_card_lock (  
    input wire key_0, key_1,  
    output reg card_read    );
```

Write the Verilog HDL for the module so that the signal `card_read` is the output of a set clear latch that is cleared when `key_0` is pressed (remember the keys are active low) and set when `key_1` is pressed. The output `card_read` is to be assigned to the pin connected to the green LED `LEDG_0` which is also called `LEDG[0]`.

4. Configure the FPGA in the DE2-115 board and debug the circuit.

9.3 PART C: Electronic Card Lock

Objective

The objective of this part of the laboratory is:

1. To get experience (Here “experience” means learn many things that can not be precisely identified and therefore can not be described.) in converting a concept described in words to a well defined and well structured block diagram.
2. To learn the value of a block diagram.
3. To learn how to annotate a block diagram in sufficient detail for it to be constructed in Verilog HDL.
4. To learn that you learn from your mistakes (i.e. To learn your design skills improve with experience.)

General Description

The operation of card controlled electronic door locks used in hotels is surprisingly simple considering there is no connection between the PC in the lobby of the hotel that programmed the card and the electronic lock that must recognize the entry code and open the door. The key to the operation is in the way the entry code number is programmed into the card. The entry code is generated from an algorithm known to both the PC at the registration desk and the electronic lock at the door. Every time a new guest is issued a card for a particular room the next entry code in the sequence for that room is used. For this to work the electronic lock must know when a new card is used and change its “valid entry number” to the next in the sequence.

The way the electronic lock does this is simple. It saves the entry code for the current occupant of a particular room (say the current occupant of Room xxx is Mr. Jones) in a register and computes the entry code for the next occupant (say the next occupant of Room xxx will be Mr. Smith) using the algorithm for the sequence. It will recognize both the current occupant’s (Mr. Jones’) and the next occupant’s (Mr. Smith’s) number and open the door for both. However, if the card has the number for the next occupant (Mr. Smith’s) the electronic lock must recognize the room has a new occupant and refuse to recognize the number for Mr. Jones from that point forward. The number for the next occupant (Mr. Smith) must be transferred to the register containing the number for the current occupant (i.e. the number for Mr. Smith must replace the number for Mr. Jones). Stated more generically, once the next occupant’s card is used the previous occupant’s card will not be recognized.

The electronic lock uses a pseudo random number generator algorithm to calculate the entry code for the next occupant.

There must be a mechanism to reset the electronic lock after replacing the battery or after lightning or some other glitch causes the random number generator

to get out of sequence. Resetting the electronic lock in essence involves re-synchronizing it with the PC in lobby. For the lock designed here, the resetting is done by the manager. The manager has a special key card referred to as the “reset for guest” card.

This “reset for guest” card is permanently programmed with a number that is hardwired into the electronic lock (i.e. the number is not lost when the battery is removed). In this lab the “reset for guest” number is programmed into the FPGA at compile time as a constant. The hotel manager resets the electronic lock as follows:

1. First the manager programs a “guest card” using the PC at the registration desk.
2. Next the manager goes to the hotel room door that needs to be re-synchronized. Of course the manager takes both the newly programmed “guest card” as well as the “reset for guest” card.
3. The manager then jabs the “reset for guest” into the card reader. This puts the electronic lock in reset mode. In this mode the electronic lock will accept the next guest card that is read by the card reader as a new guest’s card.
4. The manager then jabs the guest card into the card reader. The electronic lock trips the lock (i.e. releases the door handle) and sets the entry code for the next guest to the number on the guest card.
5. The manager then jabs the guest card into the card reader a second time. Again this trips the lock (i.e. releases the door handle). However, this time it sets the entry code for the current guest to the entry code on the newly programmed card and generates a new code for the next guest.
6. The manager then returns the guest card to front desk where it will eventually be reprogrammed and issued to a future guest.
7. The PC at the registration desk and the electronic lock are now synchronized. The next card programmed at the registration desk will have the entry code for the next guest programmed on it.

The manager also has a “reset for maid” card for resetting the maids entry code. The maid service cards operate exactly the same way.

Details of Operation

The function of a door-mounted electronic card lock is to trip the door lock when a properly programmed magnetic card is inserted into the door-mounted card reader. The magnetic card is programmed at the front desk with a lengthy combination that changes each time a new guest is assigned the room. The door mounted lock/card-reader assembly is battery powered and no wires leave the door. Since the door-mounted

assembly is isolated from the PC at the registration desk in the hotel lobby, it must know the sequence of combinations that will be issued by the PC in the lobby.

The door-mounted assembly has four parts: a card reader, a number decoder (actually 4 number decoders), a random number generator (actually two random number generators) and a magnetically activated mechanism that trips the lock. In this lab only the number decoders and random number generators will be designed. The card reader will not be built. The number will be taken from switches `sw[17:0]`. The signal that trips the lock will be sent to Green LED `LEDG[3]`.

The operation of the card reader was emulated in a previous part of this lab. The emulator works as follows: The number on the card is entered using switches `sw[17:0]`. Once the switches have been set to the number on the card the `card_read` pulse is generated by hand using the push button keys and a set/clear flip/flop. The rising edge of the `card_read` pulse is used to clock both the number decoder circuit and the random number generator in the electronic lock.

The 18-bit card number has two fields: the card type field and the combination field. The combination field is in the least significant 16 bits of `card_number`, i.e. in `sw[15:0]`. The card type field is the most two significant bits, i.e. `sw[17:16]`. The card type field specifies the type of card that was inserted. The four card types are:

`sw[17:16]==2'b00` is type "guest card". These cards are issued to the guests.

The combination portion of the card is programmed by the PC in the lobby.

`sw[17:16]==2'b01` is type "maid card". These cards work exactly the same way as the guests cards, but are issued to the maids that service the rooms.

The combination portion of the card is programmed by the PC in the lobby using a separate random number generator.

`sw[17:16]==2'b10` is type "reset for guest card", these cards are issued to the hotel managers. The combination portion of the card is the constant 16'H8001.

These cards are programmed by the manufacturer of the card lock.

`sw[17:16]==2'b11` is type "reset for maid card", these cards are issued to the hotel managers. The combination portion of the card is also the constant 16'H8001. These cards are also programmed by the manufacturer of the card lock.

The card controlled lock can be viewed as two electronic lock circuits that control one tripper. One of the circuits responds to guest cards and the other to maid cards. The circuit for the guest card is the same as the circuit for the maid card. Each contain a LSFR referred to as `guest_LFSR` and `maid_LFSR`.

The circuit reacts to each of the four types of cards as follows:

reset for guest card : When a valid " reset for guest card" is inserted the 16 flip/flops (i.e. `guest_LFSR` (consisting of flip/flops `q[15:0]`) and the 16-bit `current_guest_combination` register (i.e. the register that holds

the combination for the current guest) are synchronously cleared on the rising edge of `card_read`. In normal operation `guest_LFSR` \neq `16'H0` so if `guest_LFSR == 16'H0` the circuit controlling the guests entry is in reset mode.

guest card : The reaction to the insertion of a “guest card” is as follows:

- If `guest_LFSR` is in reset mode, i.e. `guest_LFSR == 16'H0`, at the time a guest card is inserted (this guest card will have been inserted by the manager), then `guest_LFSR` is loaded with the combination on the card on the positive edge of `card_read` and the mechanism that trips the lock is “turned on” while `card_read` is high. The `current_guest_combination` register is not changed so remains cleared.
 Note: The manager is supposed to insert the guest card a second time. On the second insertion, because the reset has put the number of the card in `guest_LFSR`, the electronic lock thinks an new guest has arrived and acts accordingly. Such action is described in the next bullet.
- If a guest card is inserted and the combination on that card equals `guest_LFSR`, i.e. a new guest is using his card for the first time, then:
 1. The circuit saves the combination on the card in `current_guest_combination` on the positive edge of `card_read`.
 2. The combination for the next guest is generated by clocking `guest_LFSR` on the positive edge of `card_read`.
 3. The tripping mechanism for the door lock is activated while `card_read` is high.
- If a guest card is inserted and the combination on that card equals the number in `current_guest_combination`, i.e. the current guest has returned to the room, then the tripping mechanism for the door lock is activated for the duration that `card_read` is high and no other action is taken.
- If a guest card is inserted and the combination on that card does not equal either `guest_LFSR` or `current_guest_combination`, then no action is taken.

reset for maid card : Parallels reset for guest card.

maid card : Parallels guest card.

Procedure

See the Pre-Laboratory Preparation subsection for instructions on what to do in advance of the entering laboratory.

1. Expand the Verilog HDL module `electronic_card_lock` that at present emulates the card reader to include half of the electronic lock. Design the portion of the lock that handles the guest card and the guest reset card. The output that trips the lock should light green LED `LEDG[3]`. The port list should be as follows:

```
module electronic_card_lock (  
    input wire key_0, key_1,  
    input wire[15:0] entry_code_on_card, // switches sw[15:0]  
    input wire[1:0] card_type, // switches sw[17:16]  
    output reg card_read,  
    output reg trip_lock_for_guest // the output that trips the lock  
);
```

NB: It is advised that you construct the LFSR inside this module. Some students have tried to instantiate the LFSR that was built in Part A of this Lab. That LFSR was designed for the single purpose of explaining how LFSRs work and the port list does not have enough signals for this application.

2. Compile and assign pins.
3. After compiling (need to compile to define the nodes used in SignalTap) include an instantiation of SignalTap that displays card_type, entry_code_on_card, card_read, trip_lock_for_guest as well as internal signals guest_LFSR and current_guest_entry_code, which is the register with the entry code for the current guest.

NOTE: Do not explicitly instantiate SignalTap. SignalTap is implicitly instantiated when it is included in the project.

NB: SignalTap does not display the waveforms until the signal buffer is full. Since one sample is collected on each clock edge, several clock edges are required to fill the buffer. For this reason it is impractical to use card_read as the clock for SignalTap. For purposes of debugging it is best to use the 27 MHz clock. This means an input, say clk, will have to be added to bring in the 27 MHz clock for the debugging phase.

It is suggested that, at least initially, you set the trigger conditions to trigger on the rising edge of card_read.

4. Compile, configure the FPGA and debug the first half of the electronic door lock.
5. Expand the Verilog HDL module electronic_card_lock to include the maid entry portion of the electronic lock.
6. Compile, configure the FPGA and debug the entire electronic door lock.

10 OPTIONAL LAB 8: Arithmetic in Verilog

Objectives

The objective of this laboratory are to have the students learn:

1. The logic circuits for adders, multipliers and comparators for signed arithmetic are different than the circuits for unsigned arithmetic.
2. The size of the circuits that result from using a “+”, a “*” and “>” in the Verilog HDL is significant.
3. Multipliers are very costly in terms of the number of LUTs required.
4. Doubt as to how a circuit designed by a compiler functions can often be removed by designing a simple test circuit and generating an “as built” truth table.

Pre-Laboratory Preparation

Estimated time required: 1.5 hours.

Read the instructions prior to entering the Laboratory. Then fill in the blank cells in Table 6 and Table 7. There is sufficient information in each row of the table to complete the blank cells. The first two rows have been completed to serve as examples.

Create a description for the third part of the lab, the intent of which is to demonstrate the difference between comparators built for signed and unsigned inputs. As part of this description, Generate a table that parallels the Tables 6 and Table 7 and fill in all the cells.

Also be sure this section of the lab draws attention to the number of luts required for the comparators.

10.1 Adders

As stated in the pre-laboratory preparation section, prior to entering the laboratory complete Table 6.

Write a Verilog HDL that builds two adders with 3-bit inputs and 4-bit outputs. One adder is to be built for unsigned integer inputs the other is to be build for signed integer inputs. The inputs to the unsigned adder should be sw[2:0] and sw[5:3]. The output, which will be an unsigned integer, should be hex display 6. The inputs to the signed adder should be sw[17:15] and sw[14:12]. The output, which will be a signed integer, should be hex display 7. Just display the two's complement number on hex display 7 (i.e. do not display the number as sign and magnitude).

Compare the number of luts required for the two adders.

b

Table 6: Table illustrating differences in signed and unsigned adders

Input 1			Input 2			Output of Unsigned Adder		Output of signed Adder	
binary	Decimal value if unsigned	Decimal value if signed	binary	Decimal value if unsigned	Decimal value if signed	binary	decimal	binary	decimal
011	3	3	100	4	-4	0111	7	1111	-1
111	7	-1	111	7	-1	1110	14	1110	-2
	3			5					
	5				-2				
		-4			-4				
		-3		2					

Verify that Table 6 was completed correctly by setting the switches to the binary values in the table and observing the outputs.

10.2 Multipliers

As stated in the pre-laboratory preparation section, prior to entering the laboratory complete Table 7.

Write a Verilog HDL that builds two multipliers with 3-bit inputs and 6-bit outputs. One multiplier is to be built for unsigned inputs the other is to be build for signed inputs. The inputs to the unsigned multiplier should be sw[2:0] and sw[5:3]. The output should be the red leds LEDR[6:0] The inputs to the signed multiplier should be sw[17:15] and sw[14:12]. The output should be red leds LEDR[17:12].

Compare the number of luts required for the two multipliers.

Verify that Table 7 was completed correctly by setting the switches to the binary values in the table and observing the outputs.

10.3 Comparators

As stated in the pre-laboratory preparation section, generate a table similar to the Tables 6 and 7 prior to entering the laboratory.

Table 7: Table illustrating differences in signed and unsigned multipliers

Input 1			Input 2			Output of Unsigned Multiplier		Output of signed Multiplier	
binary	Decimal value if unsigned	Decimal value if signed	binary	Decimal value if unsigned	Decimal value if signed	binary	decimal	binary	decimal
011	3	3	100	4	-4	001100	12	110100	-12
111	7	-1	111	7	-1	110001	49	000001	1
	3			5					
	5				-2				
		-4			-4				
		-3		2					

Comparators are circuits with two inputs and two one-bit outputs. The inputs are often referred to A and B . One output is high if and only if (iff) $A > B$ and the other output is high iff $A == B$.

Design a lab that illustrates the differences between comparators build for unsigned input and those built for signed inputs. Verify the entries in table that was generated for this section.

10.4 Circuit Sizes

To make realistic comparisons the inputs to the unsigned adder, multiplier and comparator should be 16 bits. There is no need to assign the inputs (or outputs) to pins since the information you are asked to gather comes from the compiler report. Of course there will be a warning informing you of all the unassigned pins, but in this case that is not a problem.

Make whatever changes necessary to increase the inputs to 16 bit vectors, recompile each prototype to get the circuit sizes from the compiler report.

Comment on the relative sizes of the circuits required to build adders, multipliers, and comparators.

11 **OPTIONAL LAB 9: Using Phase Lock Loops and SignalProbe**

This is an old Lab that has to be revised

Objectives

Upon completion of this Lab the students should be able to:

1. Tap into a signal that is already programmed into the FPGA without recompiling the project and connect that signal to a pin on the FPGA.
2. Understand and use a high-frequency voltage controlled oscillator with a narrow tuning range to generate square clocks of virtually any frequency.
3. To program the PLLs inside the FPGA to generate a square clock of any frequency.

Pre-Laboratory Preparation

None required.

Organization

This Lab is organized into two unconnected parts referred to as Parts A and B.

11.1 **PART A - Post Compile Routing of a Signal to a Pin**

In this part of the lab you will be routing (making tracks inside the FPGA) without recompiling. Tracking internal to the FPGA will connect an internal signal to a pin on the FPGA. This is useful when testing as it gives access to internal signals without changing internal routing for the circuit. For this feature to be used, the printed circuit board (PCB) on which the FPGA is mounted has to be laid out with it in mind. The PCB must be designed with a test header (set of test points that can be probed) that is connected to unused pins on the FPGA. During testing, signals internal to the FPGA can then be routed to the test points.

The usefulness of post compile routing is illustrated in this lab, but requires a bit of imagination. Imagine that Quartus project makes a big circuit that nearly fills the FPGA and that it takes 10 hours to compile. Imagine that for some reason, the ether net chip, which is connected to the FPGA, is not working properly and you suspect there is some timing issue, perhaps too much propagation delay, on one of the control signals. You would like to put a scope probe on the suspect pin of the ether net chip to verify your suspicion. However, nowhere on the PCB is the track connecting the FPGA pin to the ether net pin exposed. This would be the case if the package of both the ether net chip and FPGA were ball grid

array (pins on the bottom of the package) and the pins in question were connected through a track in a middle layer of the PCB board.

One option is to edit the Verilog HDLs making up the Quartus project to make a new output, which is a copy of that you would like to probe, and then assign that new output to a pin on one of the headers. To do this would require a recompile and that would take 10 hours. In addition the recompilation may change the routing of the original circuit and affect the propagation delay you are trying to measure.

A second option, which does the same thing without a recompile, is to use the post compile routing feature that Altera calls SignalProbe.

Altera describes their SignalProbe feature as:

SignalProbe Routing: The SignalProbe feature makes design verification more efficient by quickly routing internal signals to I/O pins without affecting the design. Starting with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

This means Quartus will route any internal signal to an output pin with as little propagation delay as possible and without changing any of the routing used to make the circuit. This allows you to monitor internal logic in an attempt to locate a bug without modifying the HDL or compiling the HDL which could change the circuit and mask the bug.

1. Open a project from a previous lab. Compile it and configure the FPGA.
2. Select a pin that you would like to tap. Open the signal probe assignment manager from Tools→SignalProbe Pins. Click on the top of the first column in the title field, which is labeled “numbers”. This will sort the pin numbers alphabetically making it much easier to find the pin of interest.
3. Scroll down the list of pins until you find a pin that runs to an expansion header. FPGA pin L19 connects to GPIO pin 40 (i.e. JP1_40), which is very accessible so it is a good choice. Give the signal probe entry a unique Pin name, say “test_probe”.
4. The internal signal you wish to connect to the “test_probe” pin is referred to as the Source. To specify the internal signal of interest click on the ellipsis (i.e. the "...") associated with the “Source” box. Ensure that the filter is set to “Post Compile” or “post synthesis” before clicking “List”. This will show all signals available for use by the Signal Probe feature.
Select signal that you want to connect to the “test_probe” pin. Move it to the pane on the right side and click OK.
5. In the “pin name” field type in a unique name for the pin to be added by SignalProbe. A name like “segment_a” is suitable. Click on the SignalProbe enable check box if it is not already checked. Click on add. This will

update the pin assignment information in the top window. Then click ‘OK’.

Note: In the process of assigning pin names it is possible to make mistakes that will inadvertently reserve a pin. This will remove the pin from the list of available pins in SignalProbe so you will not be able to find it. It may also cause other problems that reach beyond SignalProbe to affect the project. For example it may trip the compiler and cause an error upon compilation. To release the pin (unreserve it so to speak) open the pin assignment editor find the pin number of interest. Scroll across that row to the ‘reserve column’. If there is something in the reserve field of this row, click on it and delete it.

6. Perform a SignalProbe compilation by selecting Processing→Start→Start SignalProbe Compilation. This may generate a couple of warnings. Do not RECOMPILE as this will defeat the purpose of SignalProbe.
7. The SignalProbe compilation added the tracks in the FPGA, but it did not update the configuration file. To update the configuration file the project has to be assembled. Select Processing→Start→ Start Assembler to assemble the project.

Memory Aid: For some reason many students forget this step (i.e. to assemble) when they use SignalProbe on the first lab exam. It may be helpful to find a ‘memory jogger’ word that can be associated with SignalProbe. For example, in a bit of a stretch it can be argued that the word **MARMALADE** is associated with **SignalProbe**. The association is that both are impressive. SignalProbe is an impressive feature. Marmalade is big word used around the water cooler to impress people. Now you just have to remember the word **MARMALADE** contains three As, which stand for assemble, assemble, assemble.

8. Configure the FPGA.
9. Connect a scope probe to the header pin (i.e. JP1_40) where ‘test_probe’ has been connected. The signal signal should mimic the signal in your quartus project that you probed.

11.2 PART B: Using a PLL to create clocks

The Cyclone II has 4 phase-locked-loops. Each phase-locked-loop can be used to make 3 new clocks from an input clock. The frequency of a new clock is a constant times the frequency of the input clock. The constant must be a ratio of two integers.

The phase lock loop circuit (PLL) is shown in Figure 19. The block labeled ‘integrator’ and ‘voltage controlled oscillator’ are analog circuits, the block labeled ‘freq/phase detector’ is a mixed analog/digital circuit, and all the others blocks are digital circuits.

The block called “frequency/phase detector” has two binary-valued digital inputs and an analog output. The inputs are two clocks called the reference clock and the feedback clock. They have frequencies denoted f_{ref} and f_{fb} . The output of the “frequency/phase detector” depends on both the difference in frequency between the reference and feedback clocks and the difference in phase between these two clocks. The equation for output voltage is $K_f(f_{ref} - f_{fb}) + K_\phi(\phi_{ref} - \phi_{fb})$, where K_f and K_ϕ are constants intrinsic to the detector and ϕ_{ref} and ϕ_{fb} are the phases of the two input clocks.

The integrator integrates the output of the frequency-phase detector to produce the analog voltage that controls the frequency of the voltage controlled oscillator (VCO). The PLL operates in such a way that the frequency of the VCO increases or decreases as necessary to make the frequency and phase of the feedback clock identical to the frequency and phase of the reference clock. Since there is a divide by m in the feedback path, the frequency of the VCO will be m times the frequency of the feedback clock.

The frequency of the VCO output is m times the frequency of the reference clock. Since the reference clock is $1/n$ times the frequency of the input clock, the frequency of the VCO is m/n times the frequency of the input clock.

The three output clocks $c0$, $c1$ and $c2$ are generated by frequency dividing the VCO by g_0 , g_1 and g_2 . Therefore the frequencies of the output clocks are given by

$$f_{vco} = \frac{m}{n} f_{in} \quad (1)$$

$$f_{c0} = \frac{m}{n \times g_0} f_{in} \quad (2)$$

$$f_{c1} = \frac{m}{n \times g_1} f_{in} \quad (3)$$

$$f_{c2} = \frac{m}{n \times g_2} f_{in} \quad (4)$$

The VCO is a ring oscillator similar to the one shown in Figure 20. The frequency is controlled by changing the output impedance of the buffers/inverter. This in turn changes the propagation delay through each buffer. The period of oscillation is 8 buffer propagation delays. Such oscillators have a fairly narrow tuning range.

The ring oscillator in the Cyclone II uses 4 buffers with differential inputs and differential outputs. The propagation delays can be controlled to make the frequency of the VCO between 300 MHz and 1 GHz. This is a somewhat limited range of operation, but the PLL is designed with three frequency dividers to work around this limitation. The frequency dividers in the PLL can be programmed in such a way to make the frequency of the output virtually any frequency and yet have the VCO operate in its limited range. For example, suppose the input clock is 27 MHz and an output clock of 54 MHz is needed. The PLL could be set as follows: Make $c0$ the output clock. Set n to 1, set m to 20 and set g_0 to 10. The frequency



The other feature of a ring oscillator is there are two phases available for every stage. See Figure 20. The Cyclone II oscillator has 4 stages, which means 8 phases are available. These phases are used in the frequency dividers for c_0 , c_1 and c_2 to introduce a phase shift in the clocks. The resolution in terms of delay just depends of the frequency of the VCO. The resolution in delay is $1/(2 \times$

$no_of_stages \times f_{vco} = 1/(8f_{vco})$, which is between 0.125 ps and 0.417 ps.

The PLL circuit is not constructed with Verilog HDL. Altera provides a megafunction called “ALTPLL” for making clocks. Megafunctions are a variation of special proprietary prefabricated integrated circuits on some part of the FPGA. The Cyclone II has four complete phase locked loop circuits, one in each corner of the die. The designer customizes the megafunction with a GUI that is driven by the “megawizard plug-in manager”. The megawizard creates an augmented Verilog file (an augmented file has special non-verilog commands embedded in the comment field) that is only understood by Quartus.

The megawizard plug-in manager is found under the tools menu. Once the megawizard is open, it will ask you a series of questions starting with “Do you want to create a new megafunction or edit an existing one?”. If you select create a new megafunction, a window with a list of different types of Megafunctions appears. The type of interest is “I/O”. The specific megafunction of interest is “ALTPLL”. Once “ALTPLL” is highlighted the PLL is built by answering the questions presented by the megawizard.

After answering the question and clicking on finish, the megawizard will produce several files. One is a Verilog file that can be instantiated in a circuit. It will have special commands that are only understood by the router and the PLL is built during routing. The megawizard will also provide a file with an example instantiation if the appropriate box is checked.

The megawizard does not allow the engineer to specify the three constants n , m and g_0 . It asks for two integers: a multiplier and a divider. It will calculate the ratio of these two integers and then choose the values of m , n and g_0 to make $\frac{m}{n \times g_0}$ as close to that fraction as possible and still have the frequency of the VCO in its tuning range.

The procedure for building a PLL is given below.

1. Make a new project called “PLL_clocks”. The project will involve building a circuit that has one PLL in it.
2. The circuit to be designed is illustrated in Figure 21. It consists of two separate 1-bit counters (A one-bit counter is a frequency divider) and the instantiation of a PLL that makes two clocks. Make the clock names in the top module “clock_0” and “clock_1”. Clock one of the counters with “clock_0” and the other with “clock_1”. Call the output of the counters (frequency dividers) “clock_0_by_2” and “clock_1_by_2”. The circuit will have 4 outputs and one input. The input is “clk”. The outputs are “clock_0”, “clock_1”, “clock_0_by_2” and “clock_1_by_2”.
3. Use the “ALTPLL” megafunction to generate a Verilog prototype for clocks “clock_0” and “clock_1”. Name the prototype and the file “two_clocks.sv”. The two clocks are to be made with the same PLL.
4. The specification for the two new clocks follows:
 - (a) Use the on board 27 MHz clock as the input clock (PIN_D13). There is the option to use more than one input clock. Just use one input clock.

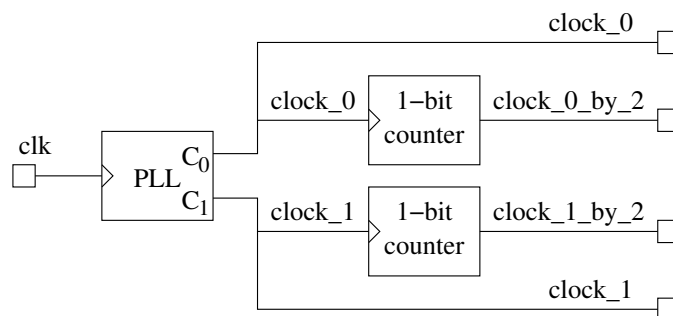


Figure 21: Schematic diagram for clock generation circuit.

There are also options for enable and reset. Do not select these options either.

- (b) Make c_0 an output clock with frequency $5/3$ that of the input clock.
 - (c) Make the phase of c_0 0 degrees.
 - (d) Make c_1 an output clock with frequency $5/3$ that of the input clock (same as c_0).
 - (e) Make the phase of c_1 as close to 90 degrees as possible.
 - (f) Also generate a sample instantiation. This file can then be opened to see how to instantiate “two_clocks.sv”.
5. Compile and simulate the circuit to make sure it works.
 6. Assign pins and recompile.
 7. Configure the FPGA.
 8. Verify proper operation using an oscilloscope with two probes. Make sure the frequency of c_0 and c_1 are $27 \times \frac{5}{3} = 45$ MHz and the phase shift between these two clocks is 90 degrees. At 45 MHz, the clock will be very square inside the FPGA, but at a pin, where there is extra capacitance, the clock may look somewhat sinusoidal or like a square wave with significant overshoot and ringing.

Also verify that the frequency of “clock_0_by_2” and “clock_1_by_2” is 22.5 MHz and that the phase between them is 45 degrees.

12 APPENDIX A - Cyclone IV E FPGA to DE2-115 board pin Mapping

The DE2-115 printed circuit board has several devices connected to the FPGA. These devices include:

1. Eight HEX displays (7 segment LED) named: HEX0[6:0], HEX1[6:0], ..., HEX7[6:0]
2. Eighteen slide switches named: SW[17:0].
3. Eighteen red LEDs (above the slide switches) named: LEDR[17:0]
4. Nine green LEDs named LEDG[8:0]
5. Two clocks. A 27 MHz clock named CLOCK_27 and a 50 MHz clock named CLOCK_50
6. Four active low momentary switches named KEY[3:0]
7. One 40-pin General Purpose Input/Output (GPIO) headers (male pin connectors). The 40-pin connectors have 2 pins connected to Vcc and 2 pins connected to ground. The other 36 pins on the header are connected to pins on the FPGA. The 36 connections between GPIO header and the FPGA pins are described by a 36-bit vectors named GPIO_0[35:0] The GPIO vectors (shown for GPIO_0[35:0] map to the header pins as follows:
GPIO_0[9:0] connect to header pins [10:1]
GPIO_0[25:10] connect to header pins [28:13]
GPIO_0[35:26] connect to header pins [40:31]
8. Several other devices that include DRAM, flash memory, ether net interface, LCD display, audio codec, UART, VGA interface.

DE2-115 board with Cyclone IV E FPGA

The FPGA PIN to DEVICE association in .csv format is given below for the DE2-115 board. The vector that describes the connections to the General Purpose Input/Output (GPIO) header is misleading as the vector index does not correspond to the GPIO pin number. For the GPIO_0 pin assignments for the DE2-115 board see Table 8 on page 106.

```
# Pin assignments for the DE2-115 board
# Note: The column header names i.e. ‘‘To, Assignment Name, Value ’’
# must be the first line of an assignment
To,      Assignment Name, Value
CLOCK_50, Location,      PIN_Y2
CLOCK2_50, Location,     PIN_AG14
CLOCK3_50, Location,     PIN_AG15
SMA_CLKIN, Location,     PIN_AH14
SMA_CLKOUT, Location,    PIN_AE23
LEDR[0], Location,      PIN_G19
LEDR[1], Location,      PIN_F19
LEDR[2], Location,      PIN_E19
LEDR[3], Location,      PIN_F21
LEDR[4], Location,      PIN_F18
LEDR[5], Location,      PIN_E18
LEDR[6], Location,      PIN_J19
LEDR[7], Location,      PIN_H19
LEDR[8], Location,      PIN_J17
LEDR[9], Location,      PIN_G17
LEDR[10], Location,     PIN_J15
LEDR[11], Location,     PIN_H16
LEDR[12], Location,     PIN_J16
LEDR[13], Location,     PIN_H17
LEDR[14], Location,     PIN_F15
LEDR[15], Location,     PIN_G15
LEDR[16], Location,     PIN_G16
LEDR[17], Location,     PIN_H15
LEDG[0], Location,      PIN_E21
LEDG[1], Location,      PIN_E22
LEDG[2], Location,      PIN_E25
LEDG[3], Location,      PIN_E24
LEDG[4], Location,      PIN_H21
LEDG[5], Location,      PIN_G20
LEDG[6], Location,      PIN_G22
LEDG[7], Location,      PIN_G21
LEDG[8], Location,      PIN_F17
KEY[0], Location,      PIN_M23
```

```

KEY[1], Location, PIN_M21
KEY[2], Location, PIN_N21
KEY[3], Location, PIN_R24
EX_I0[0], Location, PIN_J10
EX_I0[1], Location, PIN_J14
EX_I0[2], Location, PIN_H13
EX_I0[3], Location, PIN_H14
EX_I0[4], Location, PIN_F14
EX_I0[5], Location, PIN_E10
EX_I0[6], Location, PIN_D9
SW[0], Location, PIN_AB28
SW[1], Location, PIN_AC28
SW[2], Location, PIN_AC27
SW[3], Location, PIN_AD27
SW[4], Location, PIN_AB27
SW[5], Location, PIN_AC26
SW[6], Location, PIN_AD26
SW[7], Location, PIN_AB26
SW[8], Location, PIN_AC25
SW[9], Location, PIN_AB25
SW[10], Location, PIN_AC24
SW[11], Location, PIN_AB24
SW[12], Location, PIN_AB23
SW[13], Location, PIN_AA24
SW[14], Location, PIN_AA23
SW[15], Location, PIN_AA22
SW[16], Location, PIN_Y24
SW[17], Location, PIN_Y23
HEX0[0], Location, PIN_G18
HEX0[1], Location, PIN_F22
HEX0[2], Location, PIN_E17
HEX0[3], Location, PIN_L26
HEX0[4], Location, PIN_L25
HEX0[5], Location, PIN_J22
HEX0[6], Location, PIN_H22
HEX1[0], Location, PIN_M24
HEX1[1], Location, PIN_Y22
HEX1[2], Location, PIN_W21
HEX1[3], Location, PIN_W22
HEX1[4], Location, PIN_W25
HEX1[5], Location, PIN_U23
HEX1[6], Location, PIN_U24
HEX2[0], Location, PIN_AA25
HEX2[1], Location, PIN_AA26
HEX2[2], Location, PIN_Y25

```

HEX2[3],	Location,	PIN_W26
HEX2[4],	Location,	PIN_Y26
HEX2[5],	Location,	PIN_W27
HEX2[6],	Location,	PIN_W28
HEX3[0],	Location,	PIN_V21
HEX3[1],	Location,	PIN_U21
HEX3[2],	Location,	PIN_AB20
HEX3[3],	Location,	PIN_AA21
HEX3[4],	Location,	PIN_AD24
HEX3[5],	Location,	PIN_AF23
HEX3[6],	Location,	PIN_Y19
HEX4[0],	Location,	PIN_AB19
HEX4[1],	Location,	PIN_AA19
HEX4[2],	Location,	PIN_AG21
HEX4[3],	Location,	PIN_AH21
HEX4[4],	Location,	PIN_AE19
HEX4[5],	Location,	PIN_AF19
HEX4[6],	Location,	PIN_AE18
HEX5[0],	Location,	PIN_AD18
HEX5[1],	Location,	PIN_AC18
HEX5[2],	Location,	PIN_AB18
HEX5[3],	Location,	PIN_AH19
HEX5[4],	Location,	PIN_AG19
HEX5[5],	Location,	PIN_AF18
HEX5[6],	Location,	PIN_AH18
HEX6[0],	Location,	PIN_AA17
HEX6[1],	Location,	PIN_AB16
HEX6[2],	Location,	PIN_AA16
HEX6[3],	Location,	PIN_AB17
HEX6[4],	Location,	PIN_AB15
HEX6[5],	Location,	PIN_AA15
HEX6[6],	Location,	PIN_AC17
HEX7[0],	Location,	PIN_AD17
HEX7[1],	Location,	PIN_AE17
HEX7[2],	Location,	PIN_AG17
HEX7[3],	Location,	PIN_AH17
HEX7[4],	Location,	PIN_AF17
HEX7[5],	Location,	PIN_AG18
HEX7[6],	Location,	PIN_AA14
LCD_DATA[0],	Location,	PIN_L3
LCD_DATA[1],	Location,	PIN_L1
LCD_DATA[2],	Location,	PIN_L2
LCD_DATA[3],	Location,	PIN_K7
LCD_DATA[4],	Location,	PIN_K1
LCD_DATA[5],	Location,	PIN_K2


```

LCD_DATA[6], Location, PIN_M3
LCD_DATA[7], Location, PIN_M5
LCD_BLON, Location, PIN_L6
LCD_RW, Location, PIN_M1
LCD_EN, Location, PIN_L4
LCD_RS, Location, PIN_M2
LCD_ON, Location, PIN_L5
UART_TXD, Location, PIN_G9
UART_RXD, Location, PIN_G12
UART_CTS, Location, PIN_G14
UART_RTS, Location, PIN_J13
PS2_CLK, Location, PIN_G6
PS2_DAT, Location, PIN_H5
PS2_CLK2, Location, PIN_G5
PS2_DAT2, Location, PIN_F5
SD_CMD, Location, PIN_AD14
SD_CLK, Location, PIN_AE13
SD_WP_N, Location, PIN_AF14
SD_DAT[0], Location, PIN_AE14
SD_DAT[1], Location, PIN_AF13
SD_DAT[2], Location, PIN_AB14
SD_DAT[3], Location, PIN_AC14
VGA_HS, Location, PIN_G13
VGA_VS, Location, PIN_C13
VGA_SYNC_N, Location, PIN_C10
VGA_CLK, Location, PIN_A12
VGA_BLANK_N, Location, PIN_F11
VGA_R[0], Location, PIN_E12
VGA_R[1], Location, PIN_E11
VGA_R[2], Location, PIN_D10
VGA_R[3], Location, PIN_F12
VGA_R[4], Location, PIN_G10
VGA_R[5], Location, PIN_J12
VGA_R[6], Location, PIN_H8
VGA_R[7], Location, PIN_H10
VGA_G[0], Location, PIN_G8
VGA_G[1], Location, PIN_G11
VGA_G[2], Location, PIN_F8
VGA_G[3], Location, PIN_H12
VGA_G[4], Location, PIN_C8
VGA_G[5], Location, PIN_B8
VGA_G[6], Location, PIN_F10
VGA_G[7], Location, PIN_C9
VGA_B[0], Location, PIN_B10
VGA_B[1], Location, PIN_A10

```

```

VGA_B[2], Location, PIN_C11
VGA_B[3], Location, PIN_B11
VGA_B[4], Location, PIN_A11
VGA_B[5], Location, PIN_C12
VGA_B[6], Location, PIN_D11
VGA_B[7], Location, PIN_D12
AUD_ADCLRCK, Location, PIN_C2
AUD_ADCDAT, Location, PIN_D2
AUD_DACLCK, Location, PIN_E3
AUD_DACDAT, Location, PIN_D1
AUD_XCK, Location, PIN_E1
AUD_BCLK, Location, PIN_F2
EEP_I2C_SCLK, Location, PIN_D14
EEP_I2C_SDAT, Location, PIN_E14
I2C_SCLK, Location, PIN_B7
I2C_SDAT, Location, PIN_A8
ENETCLK_25, Location, PIN_A14
ENETO_TX_DATA[0], Location, PIN_C18
ENETO_RX_DATA[0], Location, PIN_C16
ENETO_TX_DATA[1], Location, PIN_D19
ENETO_RX_DATA[1], Location, PIN_D16
ENETO_TX_DATA[2], Location, PIN_A19
ENETO_RX_DATA[2], Location, PIN_D17
ENETO_TX_DATA[3], Location, PIN_B19
ENETO_RX_DATA[3], Location, PIN_C15
ENETO_GTX_CLK, Location, PIN_A17
ENETO_TX_EN, Location, PIN_A18
ENETO_TX_ER, Location, PIN_B18
ENETO_INT_N, Location, PIN_A21
ENETO_RST_N, Location, PIN_C19
ENETO_RX_DV, Location, PIN_C17
ENETO_RX_ER, Location, PIN_D18
ENETO_RX_CRS, Location, PIN_D15
ENETO_RX_COL, Location, PIN_E15
ENETO_RX_CLK, Location, PIN_A15
ENETO_TX_CLK, Location, PIN_B17
ENETO_MDC, Location, PIN_C20
ENETO_MDIO, Location, PIN_B21
ENETO_LINK100, Location, PIN_C14
ENET1_TX_DATA[0], Location, PIN_C25,
ENET1_RX_DATA[0], Location, PIN_B23,
ENET1_TX_DATA[1], Location, PIN_A26
ENET1_RX_DATA[1], Location, PIN_C21
ENET1_TX_DATA[2], Location, PIN_B26
ENET1_RX_DATA[2], Location, PIN_A23

```

```

ENET1_TX_DATA[3], Location, PIN_C26
ENET1_RX_DATA[3], Location, PIN_D21
ENET1_GTX_CLK, Location, PIN_C23
ENET1_TX_EN, Location, PIN_B25
ENET1_TX_ER, Location, PIN_A25
ENET1_INT_N, Location, PIN_D24
ENET1_RST_N, Location, PIN_D22
ENET1_RX_DV, Location, PIN_A22
ENET1_RX_ER, Location, PIN_C24
ENET1_RX_CRS, Location, PIN_D20
ENET1_RX_COL, Location, PIN_B22
ENET1_RX_CLK, Location, PIN_B15
ENET1_TX_CLK, Location, PIN_C22
ENET1_MDC, Location, PIN_D23
ENET1_MDIO, Location, PIN_D25
ENET1_LINK100, Location, PIN_D13
TD_HS, Location, PIN_E5
TD_VS, Location, PIN_E4
CLOCK_27, Location, PIN_B14
TD_RESET_N, Location, PIN_G7
TD_DATA[0], Location, PIN_E8
TD_DATA[1], Location, PIN_A7
TD_DATA[2], Location, PIN_D8
TD_DATA[3], Location, PIN_C7
TD_DATA[4], Location, PIN_D7
TD_DATA[5], Location, PIN_D6
TD_DATA[6], Location, PIN_E7
TD_DATA[7], Location, PIN_F7
OTG_DATA[0], Location, PIN_J6
OTG_DATA[1], Location, PIN_K4
OTG_DATA[2], Location, PIN_J5
OTG_DATA[3], Location, PIN_K3
OTG_DATA[4], Location, PIN_J4
OTG_DATA[5], Location, PIN_J3
OTG_DATA[6], Location, PIN_J7
OTG_DATA[7], Location, PIN_H6
OTG_DATA[8], Location, PIN_H3
OTG_DATA[9], Location, PIN_H4
OTG_DATA[10], Location, PIN_G1
OTG_DATA[11], Location, PIN_G2
OTG_DATA[12], Location, PIN_G3
OTG_DATA[13], Location, PIN_F1
OTG_DATA[14], Location, PIN_F3
OTG_DATA[15], Location, PIN_G4
OTG_ADDR[0], Location, PIN_H7

```

```

OTG_ADDR[1], Location, PIN_C3
OTG_INT[0], Location, PIN_A6
OTG_INT[1], Location, PIN_D5
OTG_DREQ[0], Location, PIN_J1
OTG_DREQ[1], Location, PIN_B4
OTG_DACK_N[0], Location, PIN_C4
OTG_DACK_N[1], Location, PIN_D4
OTG_FSPPEED, Location, PIN_C6
OTG_LSPPEED, Location, PIN_B6
OTG_RST_N, Location, PIN_C5
OTG_CS_N, Location, PIN_A3
OTG_RD_N, Location, PIN_B3
OTG_WE_N, Location, PIN_A4
IRDA_RXD, Location, PIN_Y15
DRAM_BA[0], Location, PIN_U7
DRAM_BA[1], Location, PIN_R4
DRAM_DQM[0], Location, PIN_U2
DRAM_DQM[1], Location, PIN_W4
DRAM_DQM[2], Location, PIN_K8
DRAM_DQM[3], Location, PIN_N8
DRAM_RAS_N, Location, PIN_U6
DRAM_CAS_N, Location, PIN_V7
DRAM_CKE, Location, PIN_AA6
DRAM_CLK, Location, PIN_AE5
DRAM_WE_N, Location, PIN_V6
DRAM_CS_N, Location, PIN_T4
DRAM_DQ[0], Location, PIN_W3
DRAM_DQ[1], Location, PIN_W2
DRAM_DQ[2], Location, PIN_V4
DRAM_DQ[3], Location, PIN_W1
DRAM_DQ[4], Location, PIN_V3
DRAM_DQ[5], Location, PIN_V2
DRAM_DQ[6], Location, PIN_V1
DRAM_DQ[7], Location, PIN_U3
DRAM_DQ[8], Location, PIN_Y3
DRAM_DQ[9], Location, PIN_Y4
DRAM_DQ[10], Location, PIN_AB1
DRAM_DQ[11], Location, PIN_AA3
DRAM_DQ[12], Location, PIN_AB2
DRAM_DQ[13], Location, PIN_AC1
DRAM_DQ[14], Location, PIN_AB3
DRAM_DQ[15], Location, PIN_AC2
DRAM_DQ[16], Location, PIN_M8
DRAM_DQ[17], Location, PIN_L8
DRAM_DQ[18], Location, PIN_P2

```

DRAM_DQ[19],	Location,	PIN_N3
DRAM_DQ[20],	Location,	PIN_N4
DRAM_DQ[21],	Location,	PIN_M4
DRAM_DQ[22],	Location,	PIN_M7
DRAM_DQ[23],	Location,	PIN_L7
DRAM_DQ[24],	Location,	PIN_U5
DRAM_DQ[25],	Location,	PIN_R7
DRAM_DQ[26],	Location,	PIN_R1
DRAM_DQ[27],	Location,	PIN_R2
DRAM_DQ[28],	Location,	PIN_R3
DRAM_DQ[29],	Location,	PIN_T3
DRAM_DQ[30],	Location,	PIN_U4
DRAM_DQ[31],	Location,	PIN_U1
DRAM_ADDR[0],	Location,	PIN_R6
DRAM_ADDR[1],	Location,	PIN_V8
DRAM_ADDR[2],	Location,	PIN_U8
DRAM_ADDR[3],	Location,	PIN_P1
DRAM_ADDR[4],	Location,	PIN_V5
DRAM_ADDR[5],	Location,	PIN_W8
DRAM_ADDR[6],	Location,	PIN_W7
DRAM_ADDR[7],	Location,	PIN_AA7
DRAM_ADDR[8],	Location,	PIN_Y5
DRAM_ADDR[9],	Location,	PIN_Y6,
DRAM_ADDR[10],	Location,	PIN_R5
DRAM_ADDR[11],	Location,	PIN_AA5
DRAM_ADDR[12],	Location,	PIN_Y7,
SRAM_ADDR[0],	Location,	PIN_AB7
SRAM_ADDR[1],	Location,	PIN_AD7
SRAM_ADDR[2],	Location,	PIN_AE7
SRAM_ADDR[3],	Location,	PIN_AC7
SRAM_ADDR[4],	Location,	PIN_AB6
SRAM_ADDR[5],	Location,	PIN_AE6
SRAM_ADDR[6],	Location,	PIN_AB5
SRAM_ADDR[7],	Location,	PIN_AC5
SRAM_ADDR[8],	Location,	PIN_AF5
SRAM_ADDR[9],	Location,	PIN_T7
SRAM_ADDR[10],	Location,	PIN_AF2
SRAM_ADDR[11],	Location,	PIN_AD3
SRAM_ADDR[12],	Location,	PIN_AB4
SRAM_ADDR[13],	Location,	PIN_AC3
SRAM_ADDR[14],	Location,	PIN_AA4
SRAM_ADDR[15],	Location,	PIN_AB11
SRAM_ADDR[16],	Location,	PIN_AC11
SRAM_ADDR[17],	Location,	PIN_AB9
SRAM_ADDR[18],	Location,	PIN_AB8

```

SRAM_ADDR[19], Location, PIN_T8
SRAM_DQ[0], Location, PIN_AH3
SRAM_DQ[1], Location, PIN_AF4
SRAM_DQ[2], Location, PIN_AG4
SRAM_DQ[3], Location, PIN_AH4
SRAM_DQ[4], Location, PIN_AF6
SRAM_DQ[5], Location, PIN_AG6
SRAM_DQ[6], Location, PIN_AH6
SRAM_DQ[7], Location, PIN_AF7
SRAM_DQ[8], Location, PIN_AD1
SRAM_DQ[9], Location, PIN_AD2
SRAM_DQ[10], Location, PIN_AE2
SRAM_DQ[11], Location, PIN_AE1
SRAM_DQ[12], Location, PIN_AE3
SRAM_DQ[13], Location, PIN_AE4
SRAM_DQ[14], Location, PIN_AF3
SRAM_DQ[15], Location, PIN_AG3
SRAM_UB_N, Location, PIN_AC4
SRAM_LB_N, Location, PIN_AD4
SRAM_CE_N, Location, PIN_AF8
SRAM_OE_N, Location, PIN_AD5
SRAM_WE_N, Location, PIN_AE8
FL_ADDR[0], Location, PIN_AG12
FL_ADDR[1], Location, PIN_AH7
FL_ADDR[2], Location, PIN_Y13
FL_ADDR[3], Location, PIN_Y14
FL_ADDR[4], Location, PIN_Y12
FL_ADDR[5], Location, PIN_AA13
FL_ADDR[6], Location, PIN_AA12
FL_ADDR[7], Location, PIN_AB13
FL_ADDR[8], Location, PIN_AB12
FL_ADDR[9], Location, PIN_AB10
FL_ADDR[10], Location, PIN_AE9
FL_ADDR[11], Location, PIN_AF9
FL_ADDR[12], Location, PIN_AA10
FL_ADDR[13], Location, PIN_AD8,
FL_ADDR[14], Location, PIN_AC8
FL_ADDR[15], Location, PIN_Y10
FL_ADDR[16], Location, PIN_AA8
FL_ADDR[17], Location, PIN_AH12
FL_ADDR[18], Location, PIN_AC12
FL_ADDR[19], Location, PIN_AD12
FL_ADDR[20], Location, PIN_AE10
FL_ADDR[21], Location, PIN_AD10
FL_ADDR[22], Location, PIN_AD11

```

```

FL_DQ[0], Location, PIN_AH8
FL_DQ[1], Location, PIN_AF10
FL_DQ[2], Location, PIN_AG10
FL_DQ[3], Location, PIN_AH10
FL_DQ[4], Location, PIN_AF11
FL_DQ[5], Location, PIN_AG11
FL_DQ[6], Location, PIN_AH11
FL_DQ[7], Location, PIN_AF12
FL_CE_N, Location, PIN_AG7
FL_OE_N, Location, PIN_AG8,
FL_RST_N, Location, PIN_AE11,
FL_RY, Location, PIN_Y1,
FL_WE_N, Location, PIN_AC10
FL_WP_N, Location, PIN_AE12
GPIO_0[0], Location, PIN_AB22
GPIO_0[1], Location, PIN_AC15
GPIO_0[2], Location, PIN_AB21
GPIO_0[3], Location, PIN_Y17,
GPIO_0[4], Location, PIN_AC21
GPIO_0[5], Location, PIN_Y16
GPIO_0[6], Location, PIN_AD21
GPIO_0[7], Location, PIN_AE16
GPIO_0[8], Location, PIN_AD15
GPIO_0[9], Location, PIN_AE15
GPIO_0[10], Location, PIN_AC19
GPIO_0[11], Location, PIN_AF16
GPIO_0[12], Location, PIN_AD19
GPIO_0[13], Location, PIN_AF15
GPIO_0[14], Location, PIN_AF24
GPIO_0[15], Location, PIN_AE21
GPIO_0[16], Location, PIN_AF25
GPIO_0[17], Location, PIN_AC22
GPIO_0[18], Location, PIN_AE22
GPIO_0[19], Location, PIN_AF21
GPIO_0[20], Location, PIN_AF22
GPIO_0[21], Location, PIN_AD22
GPIO_0[22], Location, PIN_AG25
GPIO_0[23], Location, PIN_AD25
GPIO_0[24], Location, PIN_AH25
GPIO_0[25], Location, PIN_AE25
GPIO_0[26], Location, PIN_AG22
GPIO_0[27], Location, PIN_AE24
GPIO_0[28], Location, PIN_AH22
GPIO_0[29], Location, PIN_AF26
GPIO_0[30], Location, PIN_AE20

```

```

GPIO_0[31], Location, PIN_AG23
GPIO_0[32], Location, PIN_AF20
GPIO_0[33], Location, PIN_AH26
GPIO_0[34], Location, PIN_AH23
GPIO_0[35], Location, PIN_AG26
HSMC_CLKOUT0, Location, PIN_AD28
HSMC_CLKIN0, Location, PIN_AH15
HSMC_D[0], Location, PIN_AE26
HSMC_D[1], Location, PIN_AE28
HSMC_D[2], Location, PIN_AE27
HSMC_D[3], Location, PIN_AF27
HSMC_TX_D_P[0], Location, PIN_D27
HSMC_RX_D_P[0], Location, PIN_F24
HSMC_TX_D_N[0], Location, PIN_D28
HSMC_RX_D_N[0], Location, PIN_F25
HSMC_TX_D_P[1], Location, PIN_E27
HSMC_RX_D_P[1], Location, PIN_D26
HSMC_TX_D_N[1], Location, PIN_E28
HSMC_RX_D_N[1], Location, PIN_C27
HSMC_TX_D_P[2], Location, PIN_F27
HSMC_RX_D_P[2], Location, PIN_F26
HSMC_TX_D_N[2], Location, PIN_F28
HSMC_RX_D_N[2], Location, PIN_E26
HSMC_TX_D_P[3], Location, PIN_G27
HSMC_RX_D_P[3], Location, PIN_G25
HSMC_TX_D_N[3], Location, PIN_G28
HSMC_RX_D_N[3], Location, PIN_G26
HSMC_TX_D_P[4], Location, PIN_K27
HSMC_RX_D_P[4], Location, PIN_H25
HSMC_TX_D_N[4], Location, PIN_K28
HSMC_RX_D_N[4], Location, PIN_H26
HSMC_TX_D_P[5], Location, PIN_M27
HSMC_RX_D_P[5], Location, PIN_K25
HSMC_TX_D_N[5], Location, PIN_M28
HSMC_RX_D_N[5], Location, PIN_K26
HSMC_TX_D_P[6], Location, PIN_K21
HSMC_RX_D_P[6], Location, PIN_L23
HSMC_TX_D_N[6], Location, PIN_K22
HSMC_RX_D_N[6], Location, PIN_L24
HSMC_TX_D_P[7], Location, PIN_H23
HSMC_RX_D_P[7], Location, PIN_M25
HSMC_TX_D_N[7], Location, PIN_H24
HSMC_RX_D_N[7], Location, PIN_M26
HSMC_CLKOUT_P1, Location, PIN_G23
HSMC_CLKIN_P1, Location, PIN_J27

```


HSMC_CLKOUT_N1,	Location,	PIN_G24
HSMC_CLKIN_N1,	Location,	PIN_J28
HSMC_TX_D_P[8],	Location,	PIN_J23
HSMC_RX_D_P[8],	Location,	PIN_R25
HSMC_TX_D_N[8],	Location,	PIN_J24
HSMC_RX_D_N[8],	Location,	PIN_R26
HSMC_TX_D_P[9],	Location,	PIN_P27
HSMC_RX_D_P[9],	Location,	PIN_T25
HSMC_TX_D_N[9],	Location,	PIN_P28
HSMC_RX_D_N[9],	Location,	PIN_T26
HSMC_TX_D_P[10],	Location,	PIN_J25
HSMC_RX_D_P[10],	Location,	PIN_U25
HSMC_TX_D_N[10],	Location,	PIN_J26
HSMC_RX_D_N[10],	Location,	PIN_U26
HSMC_TX_D_P[11],	Location,	PIN_L27
HSMC_RX_D_P[11],	Location,	PIN_L21
HSMC_TX_D_N[11],	Location,	PIN_L28
HSMC_RX_D_N[11],	Location,	PIN_L22
HSMC_TX_D_P[12],	Location,	PIN_V25
HSMC_RX_D_P[12],	Location,	PIN_N25
HSMC_TX_D_N[12],	Location,	PIN_V26
HSMC_RX_D_N[12],	Location,	PIN_N26
HSMC_TX_D_P[13],	Location,	PIN_R27
HSMC_RX_D_P[13],	Location,	PIN_P25
HSMC_TX_D_N[13],	Location,	PIN_R28
HSMC_RX_D_N[13],	Location,	PIN_P26
HSMC_TX_D_P[14],	Location,	PIN_U27
HSMC_RX_D_P[14],	Location,	PIN_P21
HSMC_TX_D_N[14],	Location,	PIN_U28
HSMC_RX_D_N[14],	Location,	PIN_R21
HSMC_TX_D_P[15],	Location,	PIN_V27
HSMC_RX_D_P[15],	Location,	PIN_R22
HSMC_TX_D_N[15],	Location,	PIN_V28
HSMC_RX_D_N[15],	Location,	PIN_R23
HSMC_TX_D_P[16],	Location,	PIN_U22
HSMC_RX_D_P[16],	Location,	PIN_T21
HSMC_TX_D_N[16],	Location,	PIN_V22
HSMC_RX_D_N[16],	Location,	PIN_T22
HSMC_CLKOUT_P2,	Location,	PIN_V23
HSMC_CLKIN_P2,	Location,	PIN_Y27
HSMC_CLKOUT_N2,	Location,	PIN_V24
HSMC_CLKIN_N2,	Location,	PIN_Y28

Table 8: FPGA connections to GPIO_0 Header (also called header JP1) on both DE2 and DE2-115 Boards. The FPGA pins for the DE2-115 board are listed in the columns labeled “Cyclone IV E Pin” while the FPGA pins for the DE2 board, which is no longer used in this lab, are listed in the columns labeled “Cyclone II Pin”.

Cyclone IV E Pin	Cyclone II Pin	JP1		Cyclone II Pin	Cyclone IV E Pin
AB22	D25	1	2	J22	AC15
AB21	E26	3	4	E25	Y17
AC21	F24	5	6	F23	Y16
AD21	J21	7	8	J20	AE16
AD15	F25	9	10	F26	AE15
Vcc5	Vcc5	11	12	GND	GND
AC19	N18	13	14	P18	AF16
AD19	G23	15	16	G24	AF15
AF24	K22	17	18	G25	AE21
AF25	H23	19	20	H24	AC22
AE22	J23	21	22	J24	AF21
AF22	H25	23	24	H26	AD22
AG25	H19	25	26	K18	AD25
AH25	K19	27	28	K21	AE25
Vcc33	Vcc33	29	30	GND	GND
AG22	K23	31	32	K24	AE24
AH22	L21	33	34	L20	AF26
AE20	J25	35	36	J26	AG23
AF20	L23	37	38	L24	AH26
AH23	L25	39	40	L19	AG26