

Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 1: Full Adder

Introduction

In this lab you will design a simple digital circuit called a *full adder*. Along the way, you will learn to use the Altera field-programmable gate array (FPGA) tools to enter a schematic, simulate your design, and download your design onto a chip. You will also build your adder on a breadboard using discrete chips to get a more tactile sense of digital logic.

After completing the lab, you are required to turn in something from each part. Refer to the “What to Turn In” section at the end of this handout before beginning the lab.

The computer-aided design (CAD) tools required for this class are installed in the E85 lab (Parsons B183). If you would like to work from the convenience of your own computer, you can download the Altera Quartus II Web Edition software from the web. Note that the latest version to support the Cyclone 2 is version 13.0 (which is what is installed on the lab computers). Some computers may have different versions of Quartus II installed. Be sure you are using the “Web Edition” for the version you use to prevent running into licensing issues.

Background: Adders

An adder, not surprisingly, is a circuit whose output is the binary sum of its inputs. Since adders are needed to perform arithmetic, they are an essential part of any computer. The full adder will be an integral part of the microprocessor that you design in later labs.

A full adder has three inputs (A , B , C_{in}) and two outputs (S , C_{out}), as shown in Figure 1. Inputs A and B each represent 1-bit binary numbers that are being added, and S represents a bit of the resulting sum.

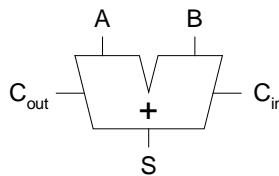


Figure 1. Full adder

The C_{in} (carry in) and C_{out} (carry out) signals are used when adding numbers that are more than one bit long. To understand how these signals are used, consider how you would add the binary numbers 101 and 001 by hand:

$$\begin{array}{r} 1 \\ 101 \\ + 001 \\ \hline 110 \end{array}$$

As with decimal addition, you first add the two least significant bits. Since $1+1=10$ (in binary), you place a zero in the least significant bit of the sum and carry the 1. Then you add the next two bits with the carry, and place a 1 in the second bit of the sum. Finally, you add the most significant bits (with no carry) and get a 1 in the most significant bit of the sum.

When a sum is performed using full adders, each adder handles a single column of the sum. Figure 2 shows how to build a circuit that adds two 3-digit binary numbers using three full adders. The C_{out} for each bit is connected to the C_{in} of the next most significant bit. Each bit of the 3-bit numbers being added is connected to the appropriate adder's inputs and the three sum outputs ($S_{2:0}$) make up the full 3-bit sum result.

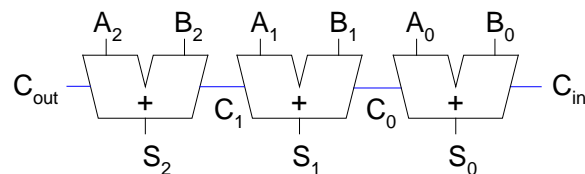


Figure 2. 3-bit adder

Note that the rightmost C_{in} input is unnecessary, since there can never be a carry into the first column of the sum. This would allow us to use a half adder for the first bit of the sum. A half adder is similar to a full adder, except that it lacks a C_{in} and is thus simpler to implement. To save you design time, however, you will only build a full adder in this lab.

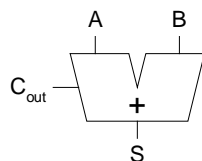


Figure 3. Half adder

1. Design

A partially completed truth table for a full adder is given in Table 1. The table indicates the values of the outputs for every possible input, and thus completely specifies the operation of a full adder. As is common, the inputs are shown in binary numeric order. The values for S (sum) are given, but the C_{out} (carry out) column is left blank. **Complete the table by filling in the correct values for C_{out} so that adders connected as in Figure 2 will perform valid addition.**

Inputs			Outputs	
C_{in}	B	A	C_{out}	S
0	0	0		0
0	0	1		1
0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		0
1	1	1		1

Table 1. Partially completed truth table for full adder

From the truth table, we now want to implement our design using logic gates. The sum output (S) can be produced from the inputs by connecting two 2-input XOR gates as shown in Figure 4. You should convince yourself that this circuit produces the outputs for S as given in the table.

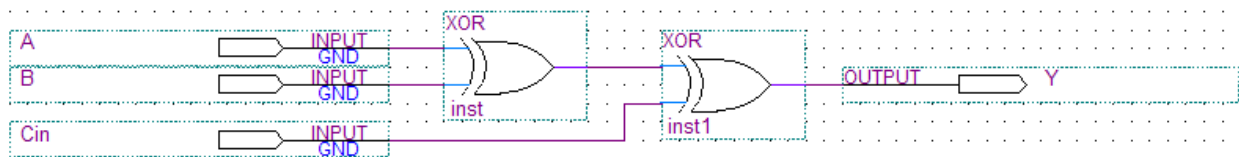


Figure 4. Schematic for sum logic

Using only two-input logic gates (AND, OR, XOR) and inverters (NOT), design a circuit that takes A , B , and C_{in} as its inputs and produces the C_{out} output. Try to use the fewest number of gates possible. Sketch your schematic.

2. Schematic

Now that you know how to produce both the sum (S) and carry out (C_{out}) outputs using simple logic gates, you will now construct a working full adder circuit using real hardware. One way to test your circuit before building it in hardware is to enter the schematic representation of your logic into a software package. You can then simulate the circuit and test that it works the way you expect it to. Some software packages are then capable of programming the schematic into an integrated circuit. This semester we will be using the Altera Quartus II 13.0 Web Edition software for these purposes. The Quartus software is a powerful and popular commercial suite of applications used by hardware designers.

First, you will learn how to start a new project. Start the Quartus software from the Start menu. If asked about the look and feel, choose Quartus II.

In the Getting Started Window, click on Create a New Project. In the New Project Wizard, set the working directory to a good place in your Charlie home directory. For example, if your Charlie directory is mapped to the H drive, choose H:\e85\lab1_xx, where xx are your initials. Name the project lab1_xx. Make sure there are no spaces or unusual characters in the path or file name; the tools may complain or silently misbehave if it has trouble with the file name. If prompted about whether to create the directory, say Yes.

Click Next to go to the Add Files page. You won't be using preexisting files, so click Next again to the Family & Device Settings to select a chip. You'll be using the Altera DE2 development board, which contains a Cyclone II EP2C35F672C6 FPGA. Set the family to Cyclone II. Scroll down and select the device (EP2C35F672C6) from the list of Available Devices. EP2C indicates the Cyclone II family of chip. The 35-series is a medium-sized chip with 33,216 (approximately 35k) logic elements. F672 indicates a fine-pitch 672-pin ball grid array package. C6 indicates a commercial-grade part (rated for operating temperatures of 0 – 85 °C) and 6 is the slowest (and cheapest) speed grade for this part.

Click Next to go to the EDA (Electronic Design Automation) Tool Settings. Set the Design Entry/Synthesis Tool name to ViewDraw using EDIF (Electronic Design Interchange Format) and the Simulation tool to Modelsim using Verilog HDL. Click Next and Finish to create your new project.

The Quartus window will open in a moment. You may wish to maximize the window. You will see three main panes, as shown in Figure 5 (and can bring them up from the View → Utility Windows menu if you accidentally close one):

- **Project Navigator:** Lists the current project's sources file and the chip in use.
- **Tasks:** Lists the processes to perform on the source selected in the Sources pane. For example, we will use this pane later to simulate your completed schematic.
- **Messages:** Lists the output of current processes, errors, and warning at the bottom of the screen. Keep an eye on these messages; important warnings appear here.

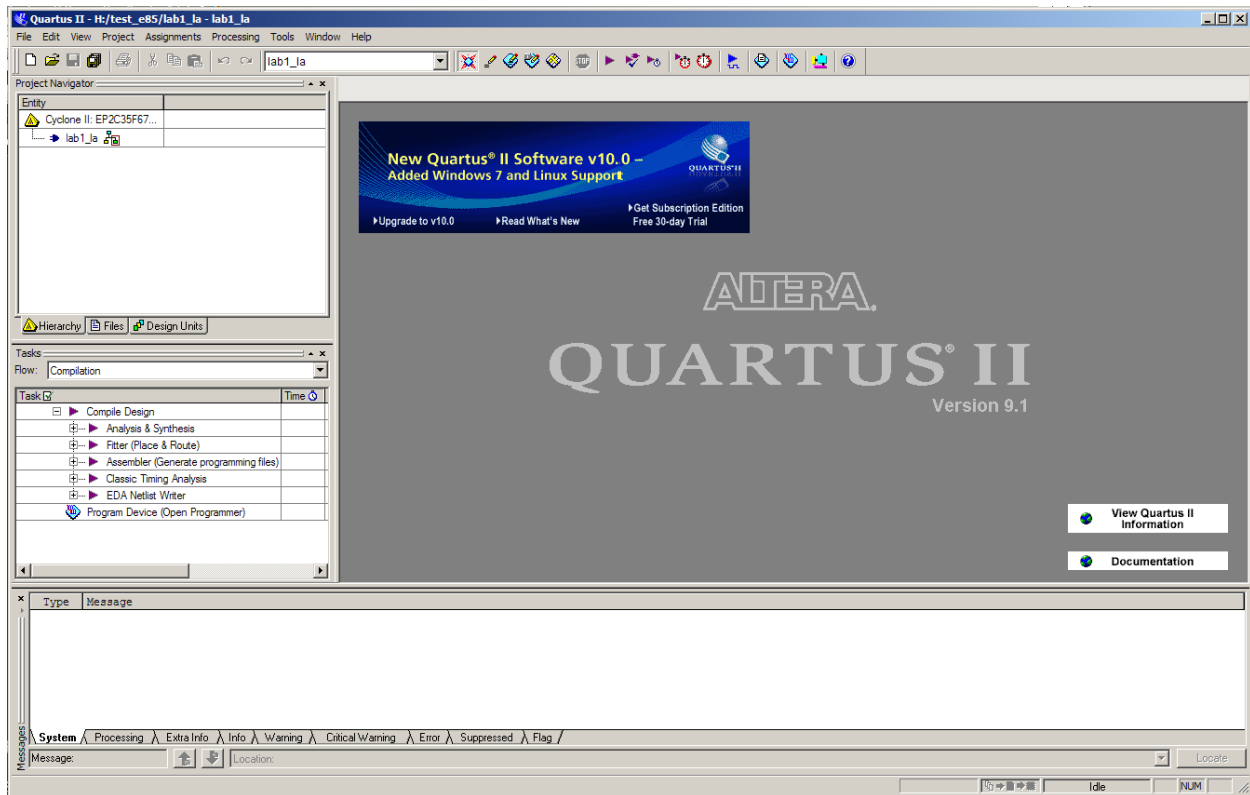



Figure 5. Quartus II window


We will describe some of the options for using these resources, but we also recommend exploring these resources on your own to become familiar with Quartus' capabilities. Use the Help menu for additional information.

Quartus has a basic and strikingly ugly schematic editor that we will use. It is not particularly sophisticated because designers today primarily use hardware description languages (HDLs) instead of schematics. However, understanding schematics is an important first step to mastering HDLs.

Create a new schematic by choosing File → New and selecting Block Diagram / Schematic File, and click OK. A new schematic window named Block1.bdf will appear.

First, place your logic gates. Click on the Symbol Tool icon (shaped like an AND gate). Expand the list of libraries in the upper left of the Symbol window by clicking on the arrow icons . Look under primitives → logic and choose xor. Click OK, then click twice on the schematic window to place two xor gates. Leave some room between the gates to draw a wire later. Press the Esc key or right click and choose Cancel to get out of the placement mode.

Click on the Symbol Tool again and choose primitives → pin → input. Place three input pins on the left side. Leave some space between the pins and the gates so that you can wire them together later. Then choose an output pin and place it on the right. Double click on one of the input pins and change its name to **A**. Leave the default value unchanged at VCC. Rename the other inputs to **B** and **Cin**. Rename the output to **S**.

Use the Orthogonal Node Tool () to wire the gates together. Click and drag to connect the pins to gates and the two gates together. At this point, your schematic should resemble Figure 4.

It's a good idea to click on the wire between the two XOR gates and give it a unique name such as *n1* or *mid* in case you need to debug later. (If you are using version 10 or higher, you can name a wire by right clicking on the wire, selecting Properties, and adding the name).


If you need to make corrections, use the Selection Tool to grab and move gates or wires. Zoom in and out by using the View menu or holding the Ctrl key while turning the mouse wheel. Use delete and undo as necessary.

Choose File → Save and save your schematic as lab1_xx.bdf.

You are now ready to complete your schematic of the full adder by drawing the logic for C_{out} that you designed in Part 1. Draw the necessary logic gates and wires to complete the circuit. Use the existing input terminals for **A**, **B**, and **C_{in}**, and add an output terminal for **C_{out}**. The symbols you may use to draw your logic gates are as follows: and2, and3, or2, or3, not, and xor.

Remember, do not add a second set of input ports for A, B, and C_{in}. Instead, note that you can connect multiple wires to the same input ports (or you can connect wires to other wires to create branches).

Select the Files tab in the Project Navigator pane to see a list of files of the project (presently just lab1_xx.bdf). If you need to reopen the file later, double-click on it here.

To check your design, click on Start Compilation  in the Task pane (Processing→Start Compilation). You'll see a compilation report indicating five pins and 2 logic elements. Review the warnings and errors carefully. You may get the following warnings that are harmless:

- Feature LogicLock only available with subscription.
- Ignored location or region assignments
- Found output pins without load capacitance
- Found invalid Fitter assignments
- Reserve All Unused Pins not specified

If you see other warnings or errors, track down their root cause before they lead you to grief later.

3. Simulation

One motivation for drawing your full adder schematic in Quartus is that you can now use the software to simulate the operation of the circuit. It is a good idea to verify the correctness of your design before actually building the circuit in hardware. In this part of the lab, you will simulate the design using **ModelSim**. Be sure to use ModelSim-Altera 10.0d **Starter Edition** to avoid licensing errors.

ModelSim expects a description of a circuit in a hardware description language (HDL) such as Verilog. To convert your schematic to Verilog, open the schematic and choose File → Create / Update → Create HDL Design File for Current File. Choose Verilog HDL. Your file should be written to lab1_xx.v. Watch for and correct any warnings or errors that arise.

Now fire up ModelSim SE 10.0d from the Windows start menu. Maximize the ModelSim window when it opens. If prompted, you may wish to associate file types with ModelSim but do not want to use Jumpstart.

Choose File → New → Project. Name the project lab1_xx and put it in the directory where you are working (e.g. H:/e85/lab1_xx). Accept the default library name of “work.” Then click “Add Existing File” and add lab1_xx.v.

You should see lab1_xx.v in the ModelSim project pane. Double-click on it to view it. The file should list the inputs and outputs and the wires (using default names if you didn’t name them yourself). It should then have a series of “assign” statements describing the gates. & indicates AND. | indicates OR. ^ indicates XOR. In future labs you will learn to write Verilog yourself.

Choose Compile → Compile All to compile the Verilog code into a form that ModelSim can simulate. Watch for and correct errors in the transcript pane. Then choose Simulate → Start Simulation. Click on Work to expand the library, and choose lab1_xx as your module to simulate. Uncheck “enable optimization” because it sometimes hides information that is useful during debugging. Click Ok.

ModelSim will open more panes including sim and Objects that help you select signals for the waveform viewer. In the sim pane, be sure lab1_xx is selected. In the objects window, you’ll see all the inputs, outputs, and internal wires. Shift-click to select them all. Then right-click and choose Add → To Wave → Selected Signals. A Wave pane will pop up with the signals.

Now it is time to apply the inputs. In the transcript pane at the bottom, type

```
force A 0
force B 0
force Cin 0
run 100
```




This will set all three inputs to 0 and simulate for 100 ns. (Note that Verilog is case-sensitive; “A” and “a” are different.) You should see all the inputs and outputs at a low level in the Wave pane. Next, raise A:

```
force A 1
run 100
```

You’ll see A rise. If your design is correct, S will also rise.

Continue with the six other patterns of inputs to check your truth table.

If you have errors, you may want to look at the internal nodes to track down the problem. Fix the schematic, then regenerate the Verilog file. Recompile and restart the simulation in ModelSim.

If the waveform is not visible, click on the + button in the top right corner of the “wave-default” pane to the right of the main ModelSim window (or choose View→Wave from the menu). Click the “Zoom Full” icon in the taskbar  to see the whole waveform of the simulation results. You can also use the “Zoom In” and “Zoom Out” icons:  . Check and see that the output values (S and C_{out}) are correct. If not, go back and fix your schematic and resimulate. When the output values are correct, you have a working full adder! Save an image of the waveform. Make sure the entire waveform is visible, select File→Export→Image..., and save the file. If needed, you can also print the waveform. Choose File→Print to print a copy of your waveforms to turn in. You can choose the start and end times in the bottom right of the print dialog box.

4. DE2 Board Implementation

Once your design simulates correctly, you may now close Modelsim and return to Quartus.

Your next goal is to download your circuit onto a DE2 board to test it on the FPGA. In hardware, particular pins on the FPGA will correspond to the inputs and outputs of your design. You'll need to assign the pins so that you can use switches to control the inputs and LEDs to display the outputs. Altera provides a file describing how the various circuits on the DE2 board are connected to the FPGA. To use this file, choose Assignments → Import Assignments. Set the file name to

```
\\Charlie.hmc.edu\Courses\Engineering\E85\Labs\DE2_pin_assignments.csv
```

Open your schematic. Rename the pins to match the names on the board. Rename the inputs from A, B, and Cin to SW[0], SW[1], and SW[2], respectively. Rename the outputs S and Cout to LEDR[0] and LEDR[1], respectively. Save your schematic.

In the Tasks pane, recompile the design.

The DE2 boards should be all set up and you should have no reason to disconnect them this semester. However, you need to plug one in, follow the steps below:

- Connect the DE2 board to the computer using a USB cable. The cable should go into the leftmost USB jack on the board labeled BLASTER.
- Switch S9 should be in the RUN position.

Check that the board is turned on (press the red power button); the blue Power and Good LEDs should turn on.

Choose Tools → Programmer. Check that the Hardware Setup is set to USB-Blaster and the mode to JTAG. The file should be lab1_xx.sof and the program/configure box should be checked. Click Start to download your design.

You may ignore the large number of warning messages related to the unused pins.

Toggle SW0, SW1, and SW2 through the eight possible patterns. Check that LEDR0 and LEDR1 display the correct sums!

If any of the boards aren't working correctly for you, try another station. If that works, label the board as bad and email the instructor with a note so that your classmates don't suffer the same issue and the problem gets fixed! ☺

5. Breadboard Implementation

In the 1970's and 1980's, engineers built systems from many small-scale integration (SSI) chips that each contain a handful of logic gates. They connected the chips together with wires on a breadboard. This approach gives you more of a visceral feel for logic gates. Breadboarding remains a valuable skill for testing out circuits before you build your own printed circuit board.

In this part of the lab, you will implement your design in hardware using 74xx-series chips, commonly sold in 14-pin dual inline packages (DIPs). Each chip contains a number of logic gates. The inputs and outputs of the gates can be accessed through the chip pins, the metal legs on each side of the black plastic package. Section A.2 of *Digital Design and Computer*

Architecture has vital information about these chips, including the pinouts and how they should be used.

First, retrieve as many 74xx-series chips as you need to implement your 1-bit full adder. They are located in the black cabinet in the lab. Recall that each chip contains multiple gates. Wires and wirecutters are also located in the lab for your use. Build your circuit on the protoboard located next to each computer. You will likely want to print out your schematic and label your circuit, as shown in Figure 6 for the sum logic. The circuit could have used any of the four XOR gates on the 7486 chip. Label each gate with the chip you will be using (in this case, the 7486 chip). Label each input and output with the pin number that you will connect it to. S1, S2, and S3 indicate switches 1, 2, and 3 on the protoboard.

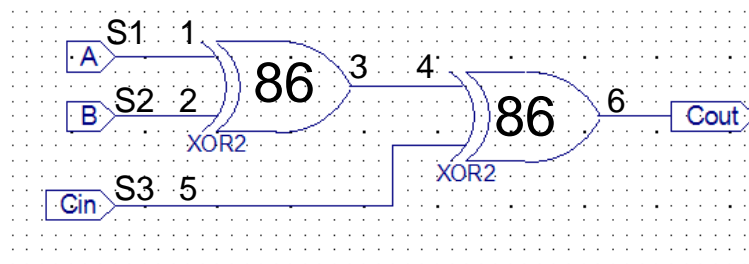


Figure 6. Example circuit schematic with pinout notations

Section A.7 of *Digital Design and Computer Architecture* explains how to place your chips on a breadboard and wire them together.

The protoboards in the lab have internal power supplies. On the top right corner of the protoboard is a red knob labeled ‘5 V’. It is internally connected to the top row of the breadboard. The black knob below it is labeled with the GND symbol and is connected to the row fourth from the top on the protoboard, as indicated by the white adjoining line.

After placing your chips on the protoboard, connect one vertical column to VDD and one vertical column to GND using wires. Note that the top half of each vertical column is inexplicably not connected to the bottom half. Either use wires to make the connection between the halves, or only use the top half. Now use short wires to connect power (VDD) and ground (GND) to your chips.

Use wires to connect the chips according to your 1-bit full adder design. Connect your inputs (A, B, and C_{in}) to the logic switches at the bottom of the protoboard (labeled “LOGIC SWITCHES”). Choose any of the switches, S₁ - S₈. It doesn’t matter which row you use. Be sure the black switch just above the input switches is switched up to +5, indicating they are 5 Volt inputs. Connect the outputs, S and C_{out} , to LED’s located in the “LOGIC INDICATORS” box on the top right of the protoboard. The red LED indicates that the output is HIGH (1), and the green LED indicates that the output is LOW (0). Be sure the switch just above the LEDs is switched up to +5 and that the switch below the LEDs is switched down to CMOS. These indicate which logic levels to use to turn on the LEDs.

After you have completed your circuit, toggle the inputs according to Table 1 and test that the outputs function correctly. If not, debug your circuit until it functions correctly. Sometimes,

connections on the breadboard can be loose or finicky. If the circuit is behaving suspiciously, it is helpful to check voltages with a multimeter and verify that they match your expectations.

It's possible to damage chips, especially by shorting an output to power or ground or another output. If your measurements reveal suspicious behavior, make sure that power and ground are attached to the chip. Disconnect all wires from the output. Apply known good inputs and check the output with

When your circuit works, you are all done. Congratulations on completing lab 1!

What to Turn In

You must submit an electronic copy of the following items via Sakai. Be sure to label each section and organize them in the following order. Messy or disorganized labs will lose points.

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Write a few sentences describing the purpose of this lab.
3. Include your completed truth table, including the values in the C_{out} column.
4. Include the following figures:
 - Your completed schematic, including the logic gates for both S and C_{out} . This can be produced using the **File→Export** feature in the Schematic Editor (you may need to select .bmp format).
 - Your simulation of the full adder, including all inputs and outputs. This can be produced using the **File→Export→Image...** feature of the ModelSim Simulator.
5. Did your full adder on the DE2 board pass work for all eight possible inputs?
6. Did your full adder on the breadboard pass work for all eight possible inputs?

If you have suggestions for further improvements of this lab, you're welcome to include them at the end of your lab.

Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 2: Seven-Segment Display

Introduction

In Lab 1, you became familiar with the Quartus schematic editor. In this lab you design a more complicated block of combinational logic. As you are completing the lab, also think about what methods will minimize design time. Use design practices that will make both designing *and* debugging efficient.

Many digital devices use seven-segment displays to represent numbers. Some examples include digital clocks and speedometers. One or more of the seven segments light up to display the desired number. In this lab, you will construct the seven-segment display decoder. It is called a decoder because it takes the four inputs and decodes them into seven outputs. You will also learn more about the schematic editor and ModelSim simulator, including how to draw busses and create formula test vectors.

As always, skim through the entire lab first, and don't forget to refer to the "What to Turn In" section at the end of the lab before you begin. There is also a set of hints on the last page of the lab.

Background

The objective of this lab is to design, simulate, and implement a seven-segment display, as shown in Figure 1. Each of the seven segments is labeled a through g. The numbers 0 through F light up the segments shown in Figure 2. For example, the number 0 lights up all but the middle segment, segment g.

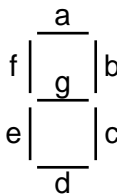


Figure 1. Seven-segment display

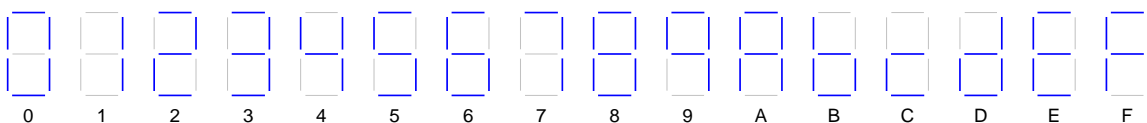


Figure 2. Seven-segment display function

You will build a seven-segment display decoder, shown in Figure 3. The circuit has four input bits, $D_{3:0}$ (representing a hexadecimal number between 0 and F), and produces seven output bits, $S_{a:g}$, that drive the seven segments to display the number. A segment of the display turns on when it is 0. Such an output is called a *low-asserted output* or *active low output*.

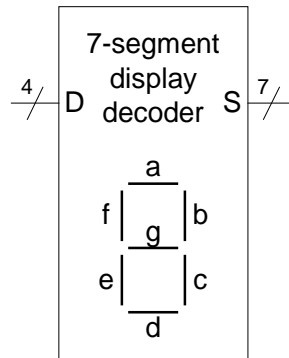


Figure 3. 7-segment display decoder

To design your seven-segment display decoder, you will first write the truth table specifying the output values for each input combination. We have started the truth table for you in Table 1. For example, when the input is $D_{3:0} = 0000$, all of the segments except g should be on. Because the outputs are active low, they must be $S_{g:a} = 1000000$. Complete the truth table for the 7-segment display decoder circuit. You will need to turn in your completed truth table.

Hexadecimal Digit	Inputs				Outputs							(in hex)
	D_3	D_2	D_1	D_0	S_g	S_f	S_e	S_d	S_c	S_b	S_a	
0	0	0	0	0	1	0	0	0	0	0	0	40
1	0	0	0	1								
2	0	0	1	0								
3	0	0	1	1								
4	0	1	0	0								
5	0	1	0	1								
6	0	1	1	0								
7	0	1	1	1								
8	1	0	0	0								
9	1	0	0	1								
A	1	0	1	0								
B	1	0	1	1								
C	1	1	0	0								
D	1	1	0	1								
E	1	1	1	0								
F	1	1	1	1								

Table 1. Truth table for 7-segment display decoder

After completing the truth table, write equations for each output segment. You should have seven separate equations for S_a through S_g . Next, translate your equations to logic gates and sketch

your design. You may use logic gates with any number of inputs. You may choose to optimize for design time or number of gates. Describe your design choice in one paragraph.

Schematic Entry

Now you are ready to enter your design in the schematic editor. Create a new schematic project called lab02_xx as you did in Lab 1.

As in the last lab, choose input and output names so that the design will automatically connect to the switches and LEDs on the DE2 board. Name your inputs SW[3] through SW[0] and your outputs HEX0[6] through HEX0[0]. SW[3] corresponds to D₃ while SW[0] corresponds to D₀. HEX0[6] corresponds to S_g while HEX0[0] corresponds to S_a.

Draw your schematic. Label all of the nodes to make debugging easier. This process is rather tedious as you must zoom in and out. You will soon learn Verilog to make the job much easier.

You may find that you don't have the right sizes of gates available and will have to make your own. For example, you can build a 5-input OR by using a 6-input OR with one input tied to GND, or using a 4-input OR followed by a two-input OR.

Create a Verilog file for your schematic. Watch for warnings in the messages window and fix anything that looks suspicious. Inspect the Verilog file in a text editor such as WordPad. Make sure the design appears to be correct. If you can't tell, it is likely wrong.

Simulation

After you are done drawing your seven-segment decoder logic, your next step is to simulate and debug your design in ModelSim. Follow the directions from Lab 1.

Both input and output bus values can be set to be displayed in either Hexadecimal, Decimal or Binary. Do this by highlighting all of the inputs and outputs. Then right-click and choose **Radix** → **Binary/Decimal/Hexadecimal**. Choose hex to make your results easy to read.

You can use the force command to drive multi-bit inputs. For example, to test an input of 7, enter

```
force SW 0111
run 100
```

Apply all sixteen possible inputs and check that the outputs agree with your truth table. If they do not, track down and fix your logic errors in the schematic.

When your simulation functions correctly, capture an image of the waveform.

Hardware Implementation

Download your design to the DE2 board using the steps from Lab 1. Toggle switches 0-3 and check that the display labeled HEX0 cycles through the correct outputs.

What to Turn In

You must submit an electronic copy of the following items via Sakai. Submit to the Lab 2 Assignment, *not* to your dropbox. These should all be included in a single file (.pdf preferable, but .doc/.docx also acceptable). Be sure to label each section and organize them in the following order. Messy or disorganized labs will lose points.

1. Please indicate how many hours you spent on this lab. This will not affect your grade, but will be helpful for calibrating the workload for next semester's labs.
2. Your completed Table 1.
3. Your output equations for each of the 7 segments.
4. A hand-sketched or computer drawn schematic of your 7-segment display decoder.
5. A paragraph describing your design method and design choice.
6. An image of your Quartus schematic showing your 7-segment display decoder logic (as in Lab 1, File→Export usually works well).
7. An image of your simulation waveforms showing correct operation for all input combinations starting from 0 and going to F (File→Export→Image...). Your waveform should show your inputs on the top and your outputs on the bottom. All values should be displayed in hexadecimal for easy reading.
8. Did your design work correctly on the DE2 board?

Some Altera Notes:

- Make sure you don't have any spaces in any of your folder or file names.
- Some CAD tools are case sensitive and others are case insensitive. Never use two different capitalizations of the same signal because some tools may treat them as different signals while others may treat them as the same signal. The easiest solution is to be consistent in your choice of capitalization.

Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 3: Adventure Game

Introduction

In this lab may you will design a **Finite State Machine** (FSM) that implements an adventure game! You will then enter the FSM into the Schematic Editor in Xilinx ISE Project Navigator, then simulate the game using ModelSim, and finally you can play your game using ModelSim.

Please read and follow the steps of this lab closely. Start early and ask questions if parts are confusing. It is much easier to get your design right the first time around than to make a mistake and spend large amounts of time hunting down the bug. As always, don't forget to read the entire lab and refer to the "What to Turn In" section at the end of this lab before you begin.



You will design your FSM using the systematic design approach described in Section 3.4.5 of the textbook.

1. Design

The adventure game that you will be designing has seven rooms and one object (a sword). The game begins in the Cave of Cacophony. To win the game, you must first proceed through the Twisty Tunnel and the Rapid River. From there, you will need to find a Vorpall Sword in the Secret Sword Stash. The sword will allow you to pass through the Dragon Den safely into Victory Vault (at which point you have won the game). If you enter the Dragon Den without the Vorpall Sword, you will be devoured by a dangerous dragon and pass into the Grievous Graveyard (where the game ends with you dead).

This game can be factored into two communicating state machines as described in Section 3.4.4. One state machine keeps track of which room you are in, while the other keeps track of whether you currently have the sword.

The Room FSM is shown in Figure 1. In this state machine, each state corresponds to a different room. Upon `reset`, the machine's state goes to the Cave of Cacophony. The player can move among the different rooms using the inputs `n`, `s`, `e`, or `w`. When in the Secret Sword Stash, the `sw` output from the Room FSM indicates to the Sword FSM that the player is finding the sword. When in the Dragon Den, signal `v`, asserted by the Sword FSM when the player has the Vorpall Sword, determines whether the next state will be Victory Vault or Grievous Graveyard; the player must not provide any directional inputs. When in Grievous Graveyard, the machine generates the `d` (dead) output, and on Victory Vault the machine asserts the `win` output.

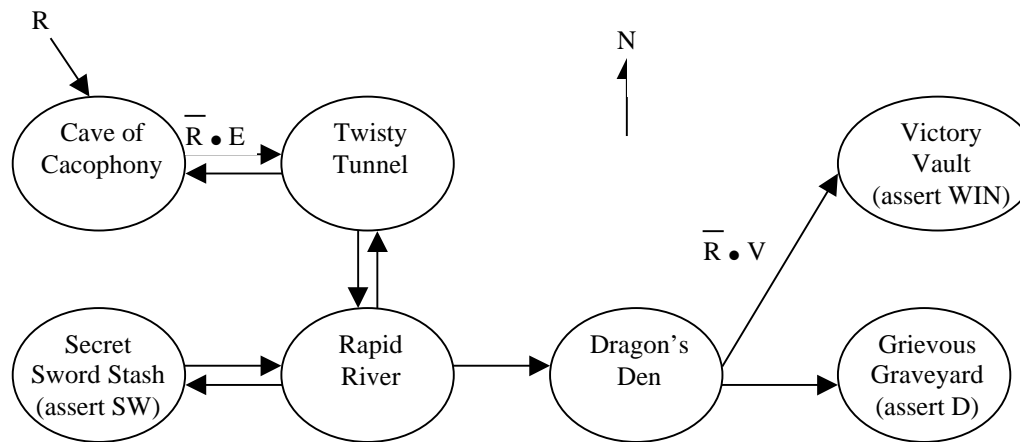


Figure 1. Partially Completed State Transition Diagram for Room FSM

In the Sword FSM (Figure 2), the states are “No Sword” and “Has Sword.” Upon `reset`, the machine enters the “No Sword” state. Entering the Secret Sword Room causes the player to pick up a sword, so the transition to the “Has Sword” state is made when the `sw` input (an output of the Room FSM that indicates the player is in the Secret Sword Stash) is asserted. Once the “Has Sword” state is reached, the `v` (vorpall sword) output is asserted and the machine stays in that state until `reset`.

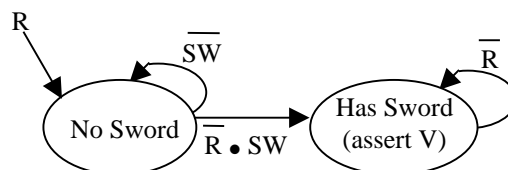


Figure 2. State Transition Diagram for Sword FSM

The state of each of these FSM's is stored using D flip-flops. Since flip-flops have a clock input, this means that there also must be a `clk` input to each FSM, which determines when the state transitions will occur.

So far, we have given an English description and a State Transition Diagram for each of the two FSM's. You may have noticed, however, that the diagram in Figure 1 is incomplete. Some of the transition arcs are labeled, while others are left blank.

Complete the State Transition Diagram for the Room FSM now by labeling all arcs so that the FSM operates as described.

The next step in the design process is to enumerate the inputs and outputs for each FSM. Figure 3 shows the inputs (on the left) and outputs (on the right) of the Room FSM and Figure 4 does this for the Sword FSM. Note that for navigational purposes the Room FSM should output $s0-s6$, indicating which of the seven rooms our hero is in. This is the last step of the design that will be given to you. Be sure to use input and output names (and capitalization) that exactly match these figures so that your design will play nicely with the testbench when you simulate later.

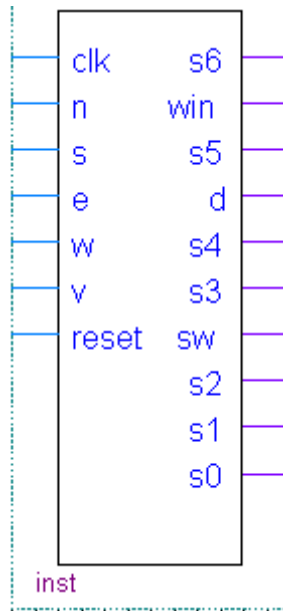


Figure 3. Symbol for Room FSM, showing its inputs and outputs

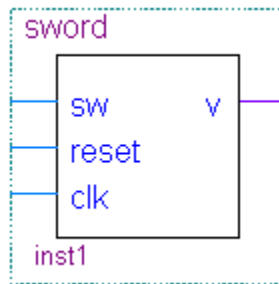


Figure 4. Symbol for Sword FSM, showing its inputs and outputs

Next, draw a state transition table for each FSM showing how the current state and inputs determine next state. The left side of the tables should have a column for the current state, and separate columns for each of the inputs. The right side should have a column for the next state. Also draw outputs tables, with the current state on the left, and the output(s) on the right. These tables are a way of representing the FSM's that is an alternative to the diagrams in Figure 3 and Figure 4.

On the left side of the table for the Room FSM, you do not need to fill in every possible combination of values for all inputs (that would make for a rather large number of rows

in your table!). Instead, for each state you only need to show the combinations of inputs for which there is an arc leaving that state in the state transition diagram. For example, when the input N is asserted and the current state is Twisty Tunnel, the behavior of the FSM is unspecified and thus does not need to be included in the table.¹ Also, you do not need to show rows in the table for what happens when more than one of the directional inputs is specified at once. You can assume that it is illegal for more than one of the *N*, *S*, *E*, and *W* inputs to be asserted simultaneously. Therefore, you can simplify your logic by making all the other directional inputs of a row “don’t care” when one legal direction is asserted. By making careful use of “don’t cares,” your table need not contain more than a dozen rows.

The next step in FSM design is to determine how to encode the states. By this, we mean that each state needs to be assigned a unique combination of zeros and ones. Common choices include binary numeric encoding, one-hot encoding, or Gray encoding. A one-hot encoding is recommended for the Room FSM (i.e. Cave of Cacophony=0000001) and makes it trivial to output your current state $s_0 \dots s_6$, but you are free to choose whichever encoding you think is best. Make a separate list of your state encodings for each FSM.

Now rewrite the table using the encoding that you chose. The only difference will be that the states will be listed as binary numbers instead of by name.

You are now approaching the heart of FSM design. Using your tables, you should be able to write down a separate Boolean logic equation for each output and for each bit of the next state (do this separately for each FSM). In your equations, you can represent the different bits of the state encoding using subscripts: S_1 , S_2 , etc. Depending on which state encoding you chose, a different number of bits will be required to represent the state of the FSM, and thus you will have a different number of equations. Simplify your equations where possible.

As you know, you can translate these equations into logic gates to implement your FSMs directly in hardware. That is what you will do in the next section.

2. Schematics

Start Quartus and create a new project named “lab03_xx” (where xx are your initials).

By now, you are familiar with the Schematic Editor. In this lab, however, you will learn how to create **hierarchical** schematic designs. In the same way that you can add symbols such as AND and OR gates to your schematic, you can add sub-components that are themselves specified by schematics. This creates a hierarchy of schematics.

Note that the flip-flop in the schematic editor is called DFF. It has asynchronous active-low set and reset inputs named PRN and CLRN. Both of these should be connected to VCC so that the element behaves as an ordinary flip-flop. (Yes, this gets pretty annoying...)

¹ Since the behavior of the FSM is unspecified in cases like this, the actual behavior of the FSM that you build in these cases is up to you. In a real system, it would be wise to do something reasonable when the user gives illegal inputs. In this game, we don’t care what your game does when given bad inputs.

You will use a hierarchical design for your adventure game by doing the following:

1. Create a schematic for the Sword FSM. Name it sword.bdf.
2. Use File -> Create/Update -> Create Symbol Files for Current File to make a symbol for your schematic. Name it sword.bsf.
3. Create a schematic and symbol for the Room FSM.
4. Create a top-level schematic named lab3_xx.bdf. Place symbols for the Room and Sword FSMs. They are accessed using the Symbol Tool just like logic gates, but appear under your project tab at the top of the list of symbols. Wire these together. The inputs and outputs of your top level schematic will determine which signals will be available in the simulator when you play the game, so you should make sure to include at least `clk`, `reset`, `n`, `s`, `e`, and `w`, as inputs and the current room `s0-s6` as an output.

If you modify the inputs or outputs for a block that you have already created, regenerate the symbol file for the block. If the symbol has already been used in a higher level of the hierarchy, right click on the symbol and choose Update Symbol or Block... to update it with the modified symbol.

When you are done, generate Verilog files for each of your three schematics. (You can switch between schematics using the Files tab in Project Navigator). Inspect the files in a text editor to make sure they look reasonable.

3. Simulation

Now it is time to fire up your game in ModelSim and play it.

It would be possible to apply the inputs and clock signal using force statements as you did in the previous labs, but this becomes rather tedious. A better approach is to use a SystemVerilog testbench. Copy the testbench from

[\\charlie.hmc.edu\Courses\Engineering\E85\Labs\lab3die_tb.sv](http://charlie.hmc.edu/Courses/Engineering/E85/Labs/lab3die_tb.sv)

to your lab3_xx directory.

Create a new project in ModelSim and add all three Verilog modules generated from your schematics along with the testbench module.

Edit the lab3die_tb.sv testbench file. Study the file until you understand it. Lines beginning with // are comments. The file defines inputs and outputs of type “logic.” It then instantiates (creates) the adventure game module. It assumes certain signal names; if they don’t match your design, correct your design to match. Modify the name of the module to match your initials. It uses a forever statement to generate a clock with a period of 100 units (nanoseconds). For example #50 indicates a delay of 50 nanoseconds. It then uses another initial block to apply the inputs every cycle. The inputs take the poor player straight to the dragon’s den to meet a hideous fate.

Compile all of the modules and start the simulation on the testbench. Be sure you have unchecked the “Enable optimization” box in the Start Simulation dialog or the signals may not appear in the Objects window. Add all of the inputs and outputs to the wave window. Also, go into your lab3_xx module (in the upper left pane) and add the sw and v signals. Type run 800. The testbench will automatically create the clock and apply the inputs so you don’t need to type force statements. Check that the player goes through the expected states and then dies. Print the results.

If this doesn’t work, debug your design. If you make changes to the inputs or outputs of a block, be sure to regenerate the symbol and update the blocks where the symbol is used.

Save a copy of the testbench in case you need to refer to it later. Modify the testbench so that the player first fetches the vorpel sword before confronting the dragon. Recompile all the files. Type restart -f in the transcript window to restart the simulation without having to add all the waves. Run for 800 ns again. If all goes well, you can celebrate your victory.

What to Turn In

You must submit an electronic copy of the following items (in the correct order, and each part clearly labeled) via Sakai. Submit to the Lab 3 Assignment. These should all be included in a single file (.pdf).

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught
2. A completed State Transition Diagram for the “Room” FSM. Scanned, hand drawn diagrams are acceptable so long as they are neat and clearly readable.
3. Your tables listing (1) next state in terms of current state and inputs and (2) output in terms of current state. You need tables for each FSM.
4. A list (one for each FSM) of your binary encoding for each state.
5. The revised copy of your tables, using your binary encoding.
6. Your Boolean logic equations for the outputs and each bit of the next state in terms of the previous state and inputs.
7. An image of your schematics for both the “Room” and “Sword” FSM’s.
8. An image of your schematic for the game (built by connecting both FSM).
9. Two images of your simulation waveforms: one that shows you playing the game and winning (entering “Victory Vault”), and another that shows an example of losing the game (entering the “Grievous Graveyard”). **Your signals must be printed in the following order: `clk, n, s, e, w, r, win, d, s6...s0, sw, v`.** Please place the images in a landscape orientation so that they fit better on the page.
10. EXTRA CREDIT: It is a little known fact that the Twisty Tunnel is located beneath the dining commons and that by heading north one can reach the dormitories. Extend your adventure game with more interesting rooms or objects. There will be a prize for the most interesting working enhancement!

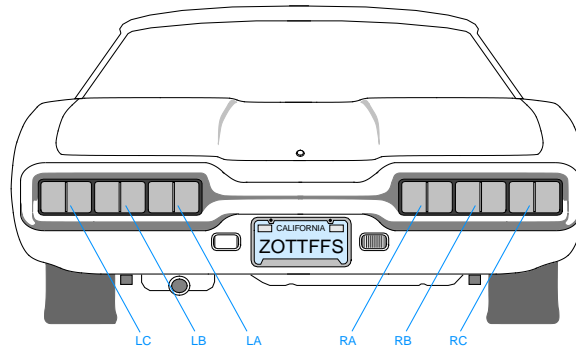
Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 4: Thunderbird Turn Signal

Introduction

In this lab, you will design a finite state machine in SystemVerilog to control the taillights of a 1965 Ford Thunderbird¹. There are three lights on each side that operate in sequence to indicate the direction of a turn. Figure 1 shows the tail lights and Figure 2 shows the flashing sequence for (a) left turns and (b) right turns.



Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

Figure 1. Thunderbird Tail Lights

Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

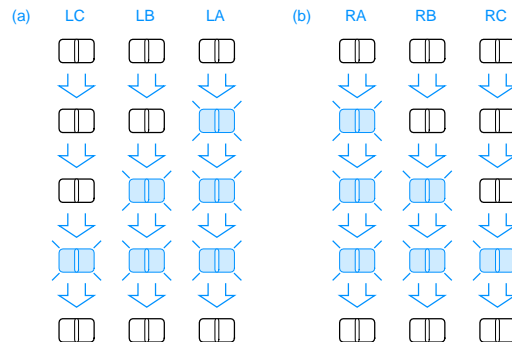


Figure 2. Flashing Sequence (shaded lights are illuminated)

¹ This lab is derived from an example by John Wakerly from the 3rd Edition of Digital Design.

This lab is divided into four parts: design, SystemVerilog entry, simulation, and implementation. If you follow the steps of FSM design carefully and ask questions at the beginning if a part is confusing, you will save yourself a great deal of time. As always, don't forget to refer to the "What to Turn In" section at the end of this lab before you begin.

1) Design

Your FSM should have the following module declaration:

```
module lab4_xx(input logic clk,
               input logic reset,
               input logic left, right,
               output logic la, lb, lc, ra, rb, rc);
```

where xx are your initials. You may assume that clk runs at the desired speed (e.g. about 1 Hz).

On reset, the FSM should enter a state with all lights off. When you press **left**, you should see LA, then LA and LB, then LA, LB, and LC, then finally all lights off again. This pattern should occur even if you release **left** during the sequence. If **left** is still down when you return to the lights off state, the pattern should repeat. **right** is similar. It is up to you to decide what to do if the user makes **left** and **right** simultaneously true; make a choice to keep your design *easy*.

Sketch a state transition diagram. Define your state encodings. **Hint: with a careful choice of encoding, your output and next state logic can be quite simple.**

Choose a set of state encodings. At this point, you could write state transition and output tables and then a set of next state and output equations as you did in Lab 3. Instead, we will design it in an HDL and let the synthesis tool choose the gate-level implementation.

2) SystemVerilog Entry

Create a new project named lab4_xx, where xx are your initials. Choose the usual EP2C35F672C6 FPGA. From this lab forward, you'll use SystemVerilog rather than schematics. Instead of choosing ViewDraw, set the synthesis tool to <None> to use the built-in SystemVerilog compiler.

Create a new SystemVerilog HDL file and save it as lab4_xx.sv. Enter Verilog code for your FSM.

3) Simulation

Create a testbench_xx.sv file that convincingly demonstrates that the FSM performs all functions correctly. Simulate your FSM in ModelSim with your testbench.

You'll probably have errors in your SystemVerilog file or testbench at first. Get used to interpreting the messages from ModelSim and correct any mistakes. In fact, it's good if you have bugs in this lab because it's easier to learn debugging now than later when you are working with a larger system!

4) Hardware Implementation

Download your design onto the DE2 board and test it in hardware.

Import the DE2 pin assignments as you did in previous labs.

You'll need to rename the inputs and outputs to connect them to switches and LEDs. This could be done with a global search and replace of signal names. But a cleaner option is to create a wrapper module that does the renaming.

Copy [\\charlie.hmc.edu\courses\Engineering\E85\Labs\lab4_wrapper.v](http://charlie.hmc.edu/courses/Engineering/E85/Labs/lab4_wrapper.v) to your directory and add it to your Quartus project. Update the wrapper to account for the name of your FSM (your initials). In the Files tab of the Project Navigator pane, right-click on lab4_wrapper.v and choose Set As Top-Level Entity to tell Quartus that you'd like to use this module. Compile the design. Fix any errors that might be found.

Look at the compilation report. Under Analysis & Synthesis, look at the resource utilization summary. Check that the number of register and I/O pins matches your expectations.

Download the design to the DE2 board. Check the wrapper to see which buttons are used for which inputs. Note that a pushbutton switch is used to create a clock. Test your design and watch the LEDs.

Note: the switches sometimes experience a phenomenon called *bounce*, in which the mechanical contacts bounce as the switch is opening or closing, creating multiple rapid rising and falling pulses rather than a single clock edge. If your lights seem to skip through multiple states at a time, it is probably because of switch bounce on the clock switch. With a bit of practice, you can learn to push the switch in a way that bounces less. It is also possible to build a circuit to “debounce” a switch, but that is beyond the scope of this lab.

What to Turn In

Please turn in each of the following items, clearly marked and in the following order as a single pdf file on Sakai:

1. Please indicate how many hours you spent on this. This will help in calibrating the workload for next time the course is taught
2. Your FSM design, including a completed state transition diagram for your FSM. Scanned images are acceptable so long as they are easily readable.
3. Your lab4_xx.sv code.
4. Your testbench_xx.sv code.
5. Image of your simulation waveforms demonstrating that your FSM performs all tasks correctly. Please display your signals in the following order: clk, reset, left, right, lc, lb, la, ra, rb, rc.
6. How many registers and I/O pins does your design use? Does it match your expectations?
7. Briefly describe how you tested the system on the DE2 board and whether it worked according to the specifications. Did you observe switch bounce?

You've now implemented your first design in SystemVerilog!

Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 5: 32-Bit ALU and Testbench

Introduction

In this lab, you will design the 32-bit Arithmetic Logic Unit (ALU) that is described in Section 5.2.4 of the text. Your ALU will become an important part of the MIPS microprocessor that you will build in later labs. In this lab you will design an ALU in SystemVerilog. You will also write a SystemVerilog testbench and testvector file to test the ALU.

Background

You should already be familiar with the ALU from Chapter 5 of the textbook. The design in this lab will demonstrate the ways in which SystemVerilog encoding makes hardware design more efficient. It is possible to design a 32-bit ALU from 1-bit ALUs (i.e., you could program a 1-bit ALU incorporating your full adder from Lab 1, chain four of these together to make a 4-bit ALU, and chain 8 of those together to make a 32-bit ALU.) However, it is altogether more efficient (both in time and lines of code) to code it succinctly in SystemVerilog.

1) SystemVerilog code

You will only be doing ModelSim simulation in this lab, so you can use your favorite text editor (such as WordPad) instead of Quartus if you like.

Create a 32-bit ALU in SystemVerilog. Name the file `alu.sv`. It should have the following module declaration:

```
module alu(input  logic [31:0] a, b,
          input  logic [1:0]  ALUControl,
          output logic [31:0] Result,
          output logic [3:0]  ALUFlags);
```

The four bits of ALUFlags should be TRUE if a condition is met. The four flags are as follows:

ALUFlag bit	Meaning
3	Result is negative
2	Result is 0
1	The adder produces a carry out
0	The adder results in overflow

An adder is a relatively expensive piece of hardware. Be sure that your design uses no more than one adder.

2) Simulation and Testing

Now you can test the 32-bit ALU in ModelSim. It is prudent to think through a set of input vectors

Develop an appropriate set of test vectors to convince a reasonable person that your design is probably correct. Complete Table 1 to verify that all 5 ALU operations work as they are supposed to. Note that the values are expressed in **hexadecimal** to reduce the amount of writing.

Test	ALUControl[1:0]	A	B	Y	ALUFlags
ADD 0+0	0	00000000	00000000	00000000	4
ADD 0+(-1)	0	00000000	FFFFFFFF	FFFFFFFF	8
ADD 1+(-1)	0	00000001	FFFFFFFF	00000000	6
ADD FF+1	0	000000FF	00000001		
SUB 0-0	1	00000000	00000000	00000000	6
SUB 0-(-1)		00000000	FFFFFFFF	00000001	0
SUB 1-1		00000001			
SUB 100-1		00000100			
AND FFFFFFFF, FFFFFFFF		FFFFFFFF			
AND FFFFFFFF, 12345678		FFFFFFFF	12345678	12345678	0
AND 12345678, 87654321		12345678			
AND 00000000, FFFFFFFF		00000000			
OR FFFFFFFF, FFFFFFFF		FFFFFFFF			
OR 12345678, 87654321		12345678			
OR 00000000, FFFFFFFF		00000000			
OR 00000000, 00000000		00000000			

Table 1. ALU operations

Build a self-checking testbench to test your 32-bit ALU. To do this, you'll need a file containing test vectors. Create a file called alu.tv with all your vectors. For example, the file for describing the first three lines in Table 1 might look like this:

```
0_00000000_00000000_00000000_4  
0_00000000_FFFFFFFF_FFFFFFFF_8  
0_00000001_FFFFFFFF_00000000_6
```

Hint: Remember that each hexadecimal digit in the test vector file represents 4 bits. Be careful when pulling signals from the file that are not multiples of four bits.

You can create the test vector file in any text editor, but make sure you save it as text only, and be sure the program does not append any unexpected characters on the end of your file. For example, in WordPad select **File**→**Save As**. In the “Save as type” box choose “Text Document – MS-DOS Format” and type “alu.tv” in the File name box. It will warn you that you are saving your document in Text Only format, click “Yes”.

Now create a self-checking testbench for your ALU. Name it testbench.sv.

Compile your alu and testbench in ModelSim and simulate the design. Run for a long enough time to check all of the vectors. If you encounter any errors, correct your design and rerun. It is a good idea to add a line with an incorrect vector to the end of the test vector file to verify that the testbench works!

What to Turn In

Please turn in each of the following items as a single pdf to the class Sakai site (in the following order and clearly labeled):

1. **Please indicate how many hours you spent on this lab.** This will be helpful for calibrating the workload for next time the course is taught. Failure to provide may result in a loss of points.
2. Your table of test vectors (Table 1).
3. Your alu.sv file.
4. Your alu.tv file.
5. Your testbench.sv file.
6. Images of your test waveforms. Make sure these are readable and that they're printed in hexadecimal. Your test waveforms should include only the following signals in the following order, from top to bottom: ALUControl, a, b, Result, ALUFlags.
7. If you have any feedback on how we might make the lab even better for next semester, that's always welcome. Please submit it in writing at the end of your lab

Digital Design and Computer Architecture: ARM Edition

Harris & Harris, © Elsevier, 2015

Lab 8: ARM Single-Cycle Processor

Introduction

In this lab you will build a simplified ARM single-cycle processor using SystemVerilog. You will combine your ALU from Lab 5 with the code for the rest of the processor taken from the textbook. Then you will load a test program and confirm that the system works. Next, you will implement two new instructions, and then write a new test program that confirms the new instructions work as well. By the end of this lab, you should thoroughly understand the internal operation of the ARM single-cycle processor.

Please read and follow the instructions in this lab carefully. In the past, many students have lost points for silly errors like not printing all the signals requested.

Before starting this lab, you should be very familiar with the single-cycle implementation of the ARM processor described in Section 7.3 of the Chapter 7 ARM draft, *Digital Design and Computer Architecture*. The single-cycle processor schematic from the text is repeated at the end of this lab assignment for your convenience. This version of the ARM single-cycle processor can execute the following instructions: ADD, SUB, AND, ORR, LDR, STR, and B.

Our model of the single-cycle ARM processor divides the machine into two major units: the control and the datapath. Each unit is constructed from various functional blocks. For example, as shown in the figure on the last page of this lab, the datapath contains the 32-bit ALU that you designed in Lab 5, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

1. ARM Single-Cycle Processor

The SystemVerilog single-cycle ARM module is given in Section 7.6 of the text. Use the electronic versions of all these files are in the class directory. Copy them to your own lab8_xx folder.

Study the files until you are familiar with their contents. Look in arm.sv. The top-level module (named top) contains the arm processor (arm) and the data and instruction memories (dmem and imem). Now look at the processor module (called arm). It instantiates two sub-modules, controller and datapath. Now take a look at the controller module and its submodules. It contains two sub-modules: decode and condlogic. The decode module produces all but three control signals. The condlogic module produces those remaining three control signals that update architectural state (RegWrite, MemWrite) or determine the next PC (PCSrc). These three signals depend on the condition mnemonic from the instruction

(Cond_{3:0}) and the stored condition flags (Flags_{3:0}) that are internal to the `condlogic` module. The condition flags produced by the ALU (ALUFlags_{3:0}) are updated in the flags registers dependent on the S bit (FlagW_{1:0}) and on whether the instruction is executed (again, dependent on the condition mnemonic Cond_{3:0} and the stored value of the condition flags Flags_{3:0}). Make sure you thoroughly understand the controller module. Correlate signal names in the SystemVerilog code with the wires on the schematic.

After you thoroughly understand the controller module, take a look at the `datapath` SystemVerilog module. The `datapath` has quite a few submodules. Make sure you understand why each submodule is there and where each is located on the ARM single-cycle processor schematic. You'll notice that the `alu` module is not defined. Copy your ALU from Lab 5 into your `lab9_xx` directory. Be sure the module name matches the instance module name (`alu`), and make sure the inputs and outputs are in the same order as in they are expected in the `datapath` module.

The instruction and data memories instantiated in the `top` module are each a 64-word \times 32-bit array. The instruction memory needs to contain some initial values representing the program. The test program is given in Figure 7.60 of the draft textbook. Study the program until you understand what it does. The machine language code for the program is stored in `memfile.dat`.

2. Testing the single-cycle ARM processor

In this section, you will test the processor with your ALU.

In a complex system, if you don't know what to expect the answer should be, you are unlikely to get the right answer. Begin by predicting what should happen on each cycle when running the program. Complete the chart in Table 1 at the end of the lab with your predictions. What address will the final STR instruction write to and what value will it write?

Simulate your processor with ModelSim. Refer to your earlier lab handouts if you need a refresher on how to use ModelSim. Be sure to add all of the `.sv` files, including the one containing your ALU. Add all of the signals from Table 1 to your waves window. (Note that many are not at the top level; you'll have to drill down into the appropriate part of the hierarchy to find them.)

Run the simulation. If all goes well, the testbench will print "Simulation succeeded." Look at the waveforms and check that they match your predictions in Table 1. If they don't, the problem is likely in your ALU or because you didn't properly add all of the files.

If you need to debug, you'll likely want to view more internal signals. However, on the final waveform that you turn in, show **ONLY** the following signals in this order: `clk`, `reset`, `PC`, `Instr`, `ALUResult`, `WriteData`, `MemWrite`, and `ReadData`. **All the values need to be output in hexadecimal and must be readable to get full credit.**

After you have fixed any bugs, print out your final waveform.

3. Modifying the ARM single-cycle processor

You now need to modify the ARM single-cycle processor by adding the EOR and LDRB instructions. First, modify the ARM processor schematic/ALU at the end of this lab to show what changes are necessary. You can draw your changes directly onto the schematics. Then

modify the main decoder and ALU decoder as required. Show your changes in the tables at the end of the lab. Finally, modify the SystemVerilog code as needed to include your modifications.

4. Testing your modified ARM single-cycle processor

Next, you'll need a test program to verify that your modified processor works. The program should check that your new instructions work properly and that the old ones didn't break. Use memfile2.asm below.

```
; memfile2.asm
; david_harris@hmc.edu and sarah_harris@hmc.edu 3 April 2014
MAIN      SUB    R0, R15, R15
          ADD    R1, R0, #255
          ADD    R2, R1, R1
          STR    R2, [R0, #196]
          EOR    R3, R1, #77
          AND    R4, R3, #0x1F
          ADD    R5, R3, R4
          LDRB   R6, [R5]
          LDRB   R7, [R5, #1]
          SUBS   R0, R6, R7
          BLT    MAIN
          BGT    HERE
          STR    R1, [R4, #110]
          B      MAIN
HERE      STR    R6, [R4, #110]
```

Figure 1. ARM assembly program: memfile2.asm

Convert the program to machine language and put it in a file named memfile2.dat. Modify imem to load this file. Modify the testbench to check for the appropriate address and data value indicating that the simulation succeeded. Run the program and check your results. Debug if necessary. When you are done, print out the waveforms as before and indicate the address and data value written by the final STR instruction.

What to Turn In

Please turn in each of the following items, clearly labeled and in the following order:

1. **Please indicate how many hours you spent on this lab.** This will not affect your grade (unless omitted), but will be helpful for calibrating the workload for next semester's labs.
2. A completed version of Table 1.
3. An image of the simulation waveforms showing correct operation of the processor. Does it write the correct value to address 100?

The simulation waveforms should give the signal values in hexadecimal format and should be in the following order: clk, reset, PC, Instr, ALUResult, WriteData,

MemWrite, and ReadData. While you may print more signals during debug, do not display any other signals in the waveform you submit. Check that the waveforms are zoomed out enough that the grader can read your bus values. Unreadable waveforms will receive no credit. Use several pages and multiple images as necessary.

4. Marked up versions of the datapath schematic and decoder tables that adds the EOR and LDRB instructions.
5. Your SystemVerilog code for your modified ARM computer (including EOR and LDRB functionality) with the changes highlighted and commented in the code.
6. The contents of your memfile2.dat containing your machine language code.
7. An image of the simulation waveforms showing correct operation of your modified processor on the new program. What address and data values are written by the final STR instruction?

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	Flags3:0 [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	8	8	0	0	?	1	8	0	x
2	0	04	ADD R2, R0, #5 E2802005	0	5	0	5	?	1	x	0	x
3	0	08	ADD R3, R0, #12 E280300C	0	C	0	C	?	1	x	0	x
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												

Table 1. First nineteen cycles of executing armtest.asm (all in hexadecimal, except Flags3:0 in binary)

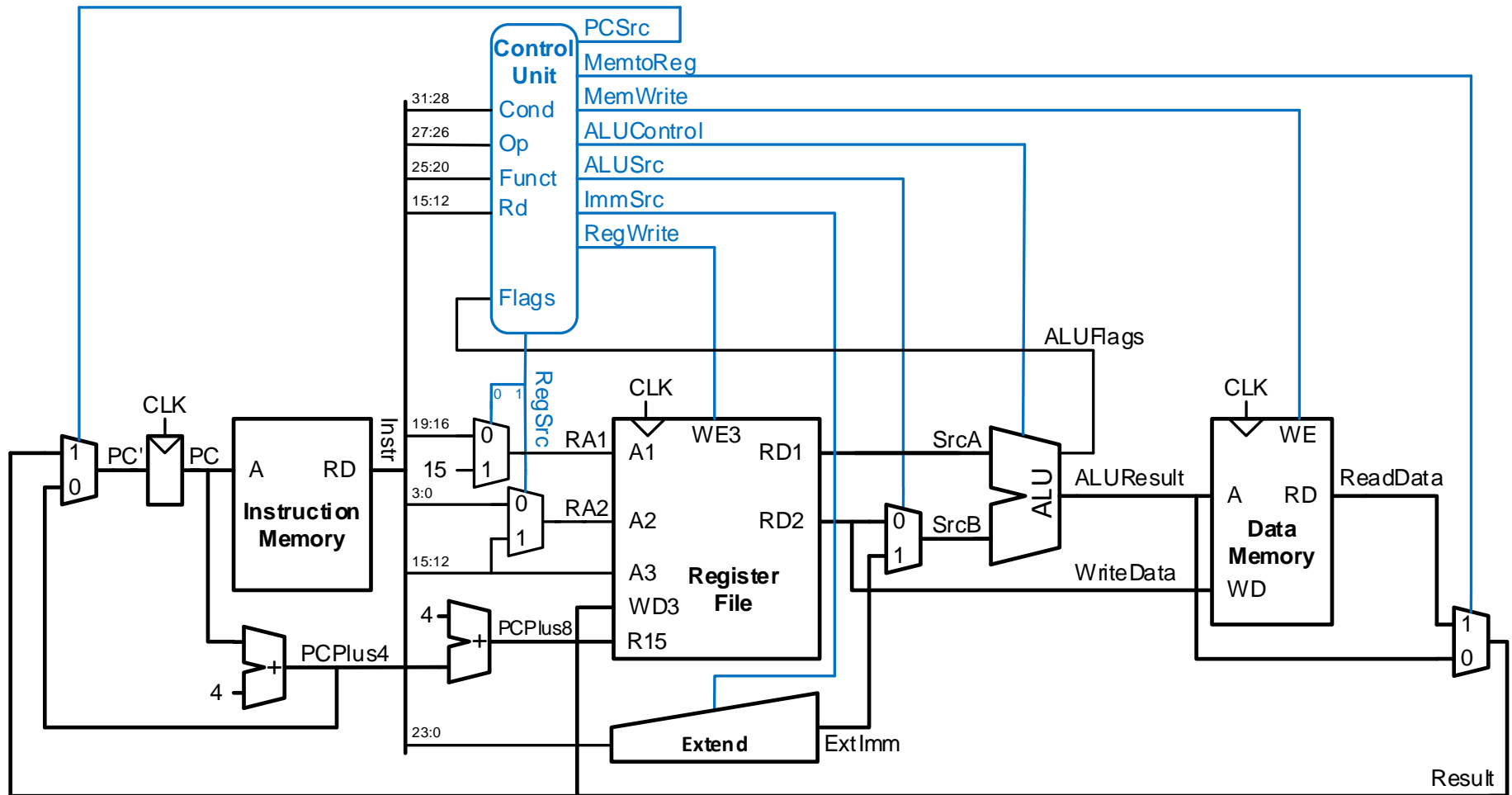


Figure 2. Single-cycle ARM processor

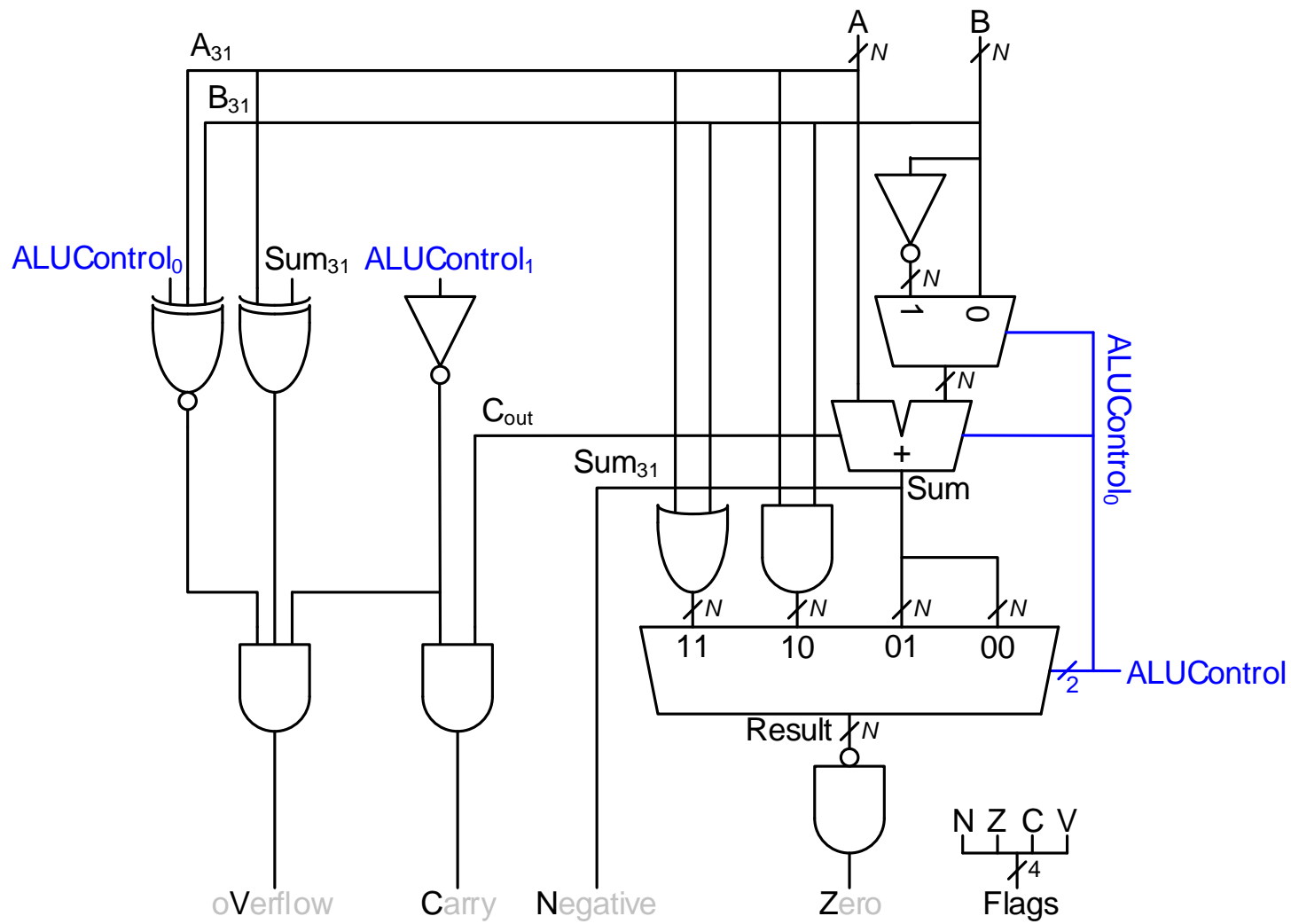


Figure 3. ARM ALU

Table 2. Extended functionality: Main Decoder

Op	Funct ₅	Funct ₀	Type	Branch	MementoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Table 3. Extended functionality: ALU Decoder

ALUOp	Funct _{4:1} (cmd)	Funct ₀ (<u>S</u>)	Notes	ALUControl _{1:0}	FlagW _{1:0}
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

ESE 2005: LAB 7: INTERFACE HARDWARE DESIGN: GPIOs, ADC

- provide your analog circuit designs for GPIO inputs, outputs, with and without internal pullup/down resistors, as needed by Labs 5-7 of ESE2025
- provide your analog amplifier/buffer designs for ADC readings as needed in Labs 6-8 of ESE2025

ESE 2005: LAB 8: ARM ASSEMBLY PROGRAMMING: PART I

- develop an assembly language program that employs sysfs for GPIO read/write and configuring

ESE 2005: LAB 9: ARM ASSEMBLY PROGRAMMING: PART II

- develop an assembly language program to perform ADC readings via sysfs
- cross-assemble both programs (LAB 8 and LAB 9), deploy to your Beaglebone and test.
- provide your test results.