18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

Lab1: Combinational Logic Design

Objective and Overview

This lab provides an opportunity to learn how to design, implement, debug and evaluate a combinational logic circuit. You will design minimized SOP and POS implementations of a combinational 6-input function. You will write SystemVerilog code to describe the circuit and implement it on the FPGA board.

As you design and implement the function, you will sharpen the digital design skills you first tried out in Lab 0 — writing multi-module SystemVerilog code, using the simulator and synthesizing designs for download to the DE2-115 FPGA board.

Schedule and Scoring

One week of lab time is scheduled for this exercise. A demo and lab report are due in subsequent weeks.

30 Jan – 1 Feb	Simulation / Synthesis. Briefing due by 10:00pm.
6 – 8 February	Demo deadline, 7:00pm on the day of your lab 2 session.
7 – 9 February	Lab report is due at 7:00pm on the day AFTER your lab 2 session.

The 18-240 lab room is available during the day for you to work. You are free to go into the lab to work on your own! Lab office hours are held on Tuesday – Thursday from 6:00-7:00pm. During office hours, TAs are available for help and for demos.

Advice: Don't show up in lab and say "I wonder what this lab is about." Unlike Lab 0, the remaining labs in this course require time to design, implement, and test. Your lab time is precious because it is limited and because the TAs are easily available. Come to lab prepared to take advantage of your time.

The lab is worth 70 points:

15 points: A briefing to your TA is due by the end of your lab session.

- Show the lab TA the SystemVerilog description of your testbench and explain your testing strategy. (See the Appendix, with suggestions on building a testbench).

- Show the lab TA your truth-table, K-map, minimized POS expression, and (possibly still buggy) SystemVerilog description of your combinational function.

20 points: The final demo is due by the beginning of the Lab 2 session. You must be checked off for the demo to receive any credit for this lab.

- Demonstrate the simulation of your design against your testbench to show the design is correct. Download and demonstrate your design on the DE2-115 board.

10 points: Your SystemVerilog code is well-written, using good style, and follows the course coding guidelines (see **`SystemVerilog_CodingStandards.pdf`** on Canvas)

25 points: A lab report is required, one per group. You must do a lab report to get any credit for this lab. Submit your lab report to Gradescope (make sure to include your partner in the submission). In the uploaded PDF, you may include pages that are scans of neatly-drawn schematic diagrams. You will also submit all your code using **`handin240`**.

- 10 points will be based on a write-up documenting your work including logic equations, K-maps, logic diagrams corresponding to the SystemVerilog, gate count, documented SystemVerilog code, and simulation results from the demonstrated design. Be sure to address any questions specifically posted in this document.

- 5 points will be based on how well you minimized the number of gates. Note if you don't correctly follow the rules on using the set of allowable gates, we can't give you any points here!

- 10 points will be based on your SOP simulation work in Step 6.

As a guideline, your lab report needs to have at least the following:

- A discussion of how you worked through the lab.
  - Your truth table/K-map/minimized SOP equation
  - Your truth table/K-map/minimized POS equation

- Report and compare on the costs of the two implementations.

- The thoroughly documented SystemVerilog descriptions for this lab, including the testbench. Submit as separate .sv files, not in your PDF report.

- The discussion of how you developed the tester.

Late demo: You must demo to get any points for the lab. You will lose 10% for each day late up to a week late. After a week, you will get zero points. We don't count any days that do not have office hours available (Saturdays, primarily).

Late lab report: You must do a lab report to get any points for a lab, including the demo. You will lose 10% for each day late—weekends not included—up to a week late. After a week late, you will get zero points for the lab.

Collaboration: Recall from the Syllabus that you are expected to only turn in work that is fully your own. This policy applies to labs, with the obvious exception that you are expected to work with your lab partner. You may get advice from TAs and talk in general terms with other people. But, every bit of the lab code and reports must be your own work.

A Note About Teams

As discussed in class, you will work in a randomly assigned team of two people. You are expected to work together. This means that you must communicate well (including answering email prior to lab), discuss your design and share responsibilities throughout the performance of the lab. Exhibiting good teamwork skills is inherently included in the grade you receive. Therefore, the following sorts of behaviors will cost you points:

- Not answering email from a partner who wishes to meet and prepare prior to lab.
- Showing up late or not at all for lab time. Leaving early when there is still work to do.
- One teammate daydreaming while the other teammate cranks in all the SV code.
- Showing up with all the work completed before you've even talked to your partner.
- Letting your teammate do all the work.
- Doing all the work without involving your teammate.

The Zorgian Language

As you know if you've been paying attention to the news reports (or Jimmy Fallon), mankind has just made contact with sentient space aliens. Scientists are in the process of decoding their language and need your help.

You've probably seen pictures of these strange creatures, with their hard, triangular bodies and three heads. The Zorgs "speak" by emitting a strange hissing / clicking / spitting sound from their seven mouths. Scientists have discovered that each mouth emits the same sound all the time. The seven sounds are: click, pop, hiss, shriek, whistle, bang, gargle.

Interestingly enough, Zorgs "think" in English (perhaps because they've been listening in on TV broadcasts. They really like *The Big Bang Theory* as it explains everything about human life). The Zorgs with seven mouths don't have any way to form sounds for English words, so their language elements (i.e. characters) are formed by a combination of two sounds, each "spoken" by a different mouth, one after the other. Each two-sound combination maps to specific English characters as shown on this chart.

		Second sound						
		click	**pop**	**hiss**	**shriek**	**whistle**	**bang**	**gargle**
		000	**001**	**010**	**011**	**101**	**110**	**111**
First Sound	**click**	B		I	L	P		
	pop		S			V	M	
	hiss			O		J	N	
	shriek	Q	R	E		F		
	whistle	Q	U		D	X		
	bang	Y	W		T	K		
	gargle	G		A	C	H		

Your Design Problem

You will design a combinatorial function to determine if a particular Zorgian sound combination is a valid English character. Your function will take 6 inputs, **`a`** – **`f`**. The first three inputs (**`a`** – **`c`**) represent the first sound encoded as a three-bit binary number. You can see the encoding under the second sounds in the table above. The second three inputs (**`d`** – **`f`**) use the same encoding to represent the second sound. Thus, a whistle-shriek (a truly hideous combination representing the letter **`D`**) is encoded **`101`** for the whistle and **`011`** for the shriek – **`101011`**.

Your function will have two outputs: **valid** and **vowel**. The signal **valid** will be asserted (i.e. it will be a '**1**') when **a** – **f** indicate a correct English character (i.e. not a blank space in the table above). The output **valid** will not be asserted when **a** – **f** indicate some other character. Note that your circuit is testing validity, so it should be ultra-conservative.[1]

The signal **vowel** will be asserted when the input represents one of the letters A, E, I, O, or U.

As an example, the input **110001** represents a consistent pattern (bang-pop, or the letter **W**), and the output of the function should be a **valid=1**, **vowel=0**.

On the other hand, the input **111001** is invalid; thus the output should be **valid=0**, **vowel=0** – there is no valid character (and thus no vowel) made of gargle then pop.

What to do?

Step 1: To begin your design, first work out the truth table for this 6-bit input, 2-bit output function. For the sake of consistency, keep the most significant input bit (**a**) on the left in your truth table. Also, show **valid** on the left for your outputs and **vowel** on the right. With 6 input bits (**a** through **f**), you have to consider two outputs for 64 different input combinations.

Step 2: Once you have the truth table, apply a K-map to derive a minimal POS equation for this combinational function. When developing the 64-entry K-map, you must follow the class guidelines (i.e. the **KmapFormats.pdf** document). As designs go, this design task isn't too bad, (though a bit annoying due to the character of this particular circuit). However, some of the things you will need to manipulate along the way are fairly complicated. Try to be systematic. We don't want to see ad hoc Boolean bit hacking. Double check your completed truth table against the TAs' before proceeding any further, since one wrong output may require a time consuming revision of your whole design.

Step 3: Implement the minimized 2-level POS logic circuit using only NOR gates (with four or fewer inputs per gate). You may use inverters to get the complement of your inputs if you need them. A part of your grade depends on minimizing the number of gates in your implementation. Inverters to complement primary inputs don't count. Write up the circuit in a structural-level SystemVerilog description.

Step 4: Testing is a very important part of the design process. You will need to write a SystemVerilog tester for your design. Hmm… how many cases are there? Do you need to exhaustively test?[2] Be prepared to show your TA how well it tests your circuit and also explain this in the lab report. See the Appendix section on *Making a tester* for more information.

Step 5: Once your design has been fully debugged and tested in simulation, synthesize the SystemVerilog circuit description and download it to the DE2-115 FPGA board. Demonstrate it to

[1] Ultra-conservative in this case doesn't have anything to do with Fascism. Instead, you should make sure that **valid** is only asserted for a sound combination that is actually valid.

[2] The answer is 'NO!' Exhaustive testing will lose you points on this lab.

your TA. Connect the six rightmost switches (**SW5...SW0**) to the 6 bits of your input in **a**, **b**, **c**, **d**, **e**, **f** order. Connect the **valid** output to **LEDR17** and **vowel** to **LEDR16**.

Your design should match the design in the reference board — a permanently programmed board that can be found in the lab.

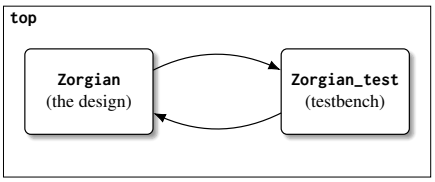
Step 6: As a part of your lab report only, design, simulate and synthesize, but do not download to the DE2-115 boards, a minimized 2-level SOP implementation using only NAND gates (with four or fewer inputs per gate) and inverters to get the complement of your inputs if you need them. Show your work to design this circuit. Include a gate-level SystemVerilog description of this circuit and a simulation trace showing the correct results. Answer these questions:

- How many NAND gates of each type? (How many 2 input gates, 3 input gates, 4 input gates, etc.) Don't count the inverters on the inputs.
- Compare the cost of the POS vs. the SOP implementation based on the number of gates.
- Compare the cost of the POS vs. the SOP implementation based the logic element usage reported by the QuartusII synthesis tool.[3] Does the comparison agree with your gate-count estimate? Why or why not?

[3] Find this value on the *Compilation Report,* that page that overlays your source code whenever you compile your design.

Appendix A: Making a Testbench

The first step toward effective debugging is to make sure that you have an accurate simulation. This process has been covered in class and was a major learning goal of Lab 0. However, understanding how to create testbenches is so critical, that I've included this short review as a reference and guide.



Creating a testbench involves making a test module and connecting it to the circuit that you designed. A common organization involves a **top** module that contains the module for your design and tester modules — shown here as **Zorgian** and **Zorgian_test**. The **Zorgian_test** has the same inputs and outputs as the **Zorgian** module only the directions are reversed; so if **a** is an input to the **Zorgian** module then it is an output of the **Zorgian_test**.

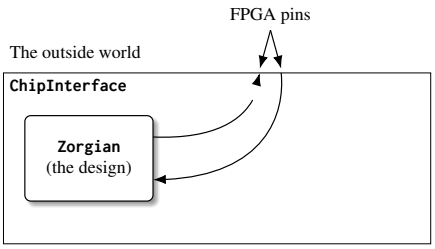
In operation, the **Zorgian_test** module will set the inputs of the **Zorgian** module (i.e. the outputs of **Zorgian_test**) and then check or display the outputs of the **Zorgian** module so that you can check it in the simulation output. The number of input combinations that you test is fully up to you, but you should be prepared to justify your decision. A poorly justified low number of tests will result in point deductions! Running through an exhaustive list of all potential inputs is also, in general, a poor decision (though possible for this lab, impossible as the number of inputs increases).

An example SystemVerilog description, corresponding to the diagram, is given below.

```
`default_nettype none
module Zorgian
  (input logic a, b, c, d, e, f,
   output logic valid, vowel);

  //Replace this comment with your work
endmodule: Zorgian

module Zorgian_test
  (output logic a, b, c, d, e, f,
   input logic valid, vowel);

  //blah blah. Replace this too
endmodule: Zorgian_test

module top();
  logic valid, vowel, a, b, c, d, e, f;

  Zorgian DUT(.a, .b, .c, .d, .e, .f, .valid, .vowel);
  Zorgian_test T(.a, .b, .c, .d, .e, .f, .valid, .vowel);

endmodule: top
```

Appendix B: Synthesizing a Circuit vs. Simulating It

When you are convinced that your design is correct in simulation, you can synthesize it. When synthesizing, you no longer need the tester (and you can't synthesize it anyway, because it has an initial block and lots of timing related statements). However, you will want to connect the ports of the design module to the physical I/O pins on the FPGA chip. Note that the design module hasn't changed at all from the simulation version. You simply want to connect its inputs and outputs to the FPGA pins. You have two choices on how to proceed:



- Synthesize the design module and then configure **`design`**'s ports directly to the pins.
- Introduce an interface module to connect your design to the FPGA pins. I commonly call this module **`ChipInterface`**, though you can pick another name if you like. The interface module lets you change names (and perhaps types) of your ports and has a bit more flexibility to it. Also, since the signals going to the FPGA pins can have a different name from the signal generated by the design, you can more easily use the provided **`.qsf`** file.

Should you choose the second method[4], your interface module would look something like:

```
module ChipInterface
  (output logic [17:0] LEDR,
   input logic [5:0] SW);

  Zorgian Z(.a(SW[5]), .b(SW[4]), .c(SW[3]), .d(SW[2]),
            .e(SW[1]), .f(SW[0]), .valid(LEDR[17]), .vowel(LEDR[16]));

endmodule: ChipInterface
```

Follow the same procedure as in Lab 0 to specify the required port-to-pin connections during synthesis. As an added bonus, if you use common naming conventions, you can reuse **`ChipInterface`** for many different designs, simply by changing the module instantiation statement-lines 5 and 6 above. This advantage doesn't sound like much right now, but as the labs get bigger and more complex, this strategy can pay off.

[4] and, you should

Appendix C: How to Debug "Hardware"

One of the few drawbacks of doing development using an FPGA versus using a protoboard is that you do not have access to the internal signals of your design. On a protoboard, you can probe any wire to see what the logic value is at that point. Debugging your design on the DE2-115 board requires that you give yourself access to the internal workings of your design by assigning internal signals to pins on the FPGA chip.

In the example code below, **mycircuit** has an internal wire (i.e. logic signal) called **temp** that is not visible on any pin on the FPGA. Trying to debug a problem with **temp** would require guess work to determine what part of the SystemVerilog description is buggy.

```
`default_nettype none
module mycircuit
  (input logic [2:0] X, Y,
   output logic [1:0] N);

   logic temp;

   not n( N[1], Y[0]);
   and a( temp, X[2], Y[1] );
   or o( N[0], temp, X[0] );

endmodule: mycircuit
```

To improve the debugability[5] of **mycircuit**, the signal **temp** can be brought outside the chip by adding it to the port list as an output and specifying which pin **temp** should connect to on the FPGA. Now the value of the signal connecting the AND and the OR gates could be determined by probing the exact corresponding pin on the FPGA. Rather than probe the pin directly (with a logic probe or o-scope or logic analyzer), drive an unused LED. The pins are much too small to reliably probe without carefully designed hardware.

Keep in mind, you should always fully debug your design in simulation where it is easy to diagnose an internal problem. This type of “hardware” debugging is your last resort when your design works in simulation but does not work in real life. This is very unlikely in the 18240 labs, provided you adhere to proper coding discipline for synthesizability[6].

[5] Don’t bug me about the validity of some of my words.

[6] See previous footnote