18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

Lab2: Advanced Comb. Logic Design – Change We Can Believe In

Objective and Overview

The purpose of lab 2 is to further develop your design techniques for larger and larger combinational logic circuits. It provides an opportunity to learn an extremely important engineering technique, divide-and-conquer, as you 1) decompose a complicated design into multiple simpler parts, 2) design, implement, debug and evaluate each submodule, and 3) connect the whole design together from the submodules. Hierarchical, synthesizable, procedural modeling in SystemVerilog will be used to specify the design. The design will be simulated and downloaded to the FPGA for demonstration in the lab.

Schedule and Scoring

All parts of the lab are to be demonstrated to your TA by the end of your 2nd session of Lab 2. The schedule for the lab is shown below.

6 – 8 February	Start your design. System diagram due in lab.
13 – 15 February	Simulation results due at start of lab. Final demos due in lab.
19 – 20 February	Midterm 1. No late penalties assessed on these two days.

Lab 2 is worth 100 points:

System level diagram (25 points): For this lab, no report is required. **However,** in its place, you will draw and maintain a block diagram describing your system. This diagram should show the sub-systems of your design and the connections between them (which modules are instantiated inside of which, as well). All signals should be carefully defined and labeled. Each subsystem should have a sentence or two ***precisely*** defining what should occur within[1]. Subsystems are the modules you will construct. They will eventually include `always_comb` blocks and standard components, but you need not draw to the level of such components. This diagram is due by the end of the 1st session of Lab 2 (show it to your TA for credit) and should be complete prior to you starting any code. This diagram can be extremely useful to you as you build the system, especially if you are willing to keep it updated as you change your design. Assign one team member to update it as the other team member is updating the code. You will submit the diagram to Gradescope soon after (as in, minutes after) your final demo.

[1] None of this should look like SystemVerilog! You should draw this diagram and write out this pseudo code PRIOR to writing your SystemVerilog.

Simulation results (30 points): Your results show your design is correct. Include individual simulations to test each of the submodules. We don't want a printout with all rows of the truth table! If you don't have all modules working in simulation in time, show us what you have working. This is due at the beginning of the 2nd session of Lab 2. It must be in your TA's hands at 7:00 to receive full credit – no exceptions![2]

It works! (35 points): A demonstration of your design working on the DE2-115 board. Show us the input vectors you designed to demonstrate it is working. Explain to the TA why you picked these ("because they worked" is not a good answer!). The TAs will try to break your design with some input test vectors. If your design is deemed to be working on your first request of the TA for a demo, you could get full points here. Demonstrate as much of the design as you can—i.e., there is partial credit for having a plan (that diagram mentioned above), and filling in pieces of it. TL;DR: you will be graded on your choice and explanation of input vectors, whether it works first time (**this is 5 points of the score**), and whether it works eventually (or partially works). This is due by the end of the 2nd session of Lab 2.

Code (10 points): Submit your code to Gradescope, using `handin240`. Your code should be clearly written, showing the hierarchical nature of this problem. Modules instantiate other modules when possible and are not just copy-and-pasted together. Code also shows proper coding style with descriptive variable names, use of whitespace, and comments. Code embodies the *18-240 SystemVerilog Coding Standards*.

Late Penalties (the standard drill)

Late demo: You will lose 10% for each day late – weekends not included – up to a week late. After a week, you will get zero points for your demo. (After which, you still need to complete the demo to pass the course.)

Late system diagram or simulation results: 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 this portion.

A Note about Teams and Collaboration

The same expectations of teamwork behavior apply to Lab 2 as were detailed in Lab 1. Be a good teammate, work together, learn together, and get a good grade together.

And, a reminder: all work you turn in for this lab is expected to be that of your team. You may ask other students for general assistance, but you may not copy their work. Likewise, you may not copy work from other sources.

[2] Metaphorically speaking, that is. In reality, your electronic submission must be submitted to Gradescope by this time.

Your Design Task

We're all accustomed to plunking some change or a dollar bill into a soda machine (or pop machine, depending on where you are) to get a tasty beverage. Well, how does a machine like this figure out how much change to give back to us? You will implement the combinational part of a "change machine" circuit for some pretty strange change.

Your lab team has been approached by Zorgian aliens and asked to design a change box for their beverage dispenser (don't even ask what sort of things Zorgians drink, you don't want to know). The Zorgs are geometry nutcases, so their monetary system is a bit strange. Their monetary units (which we will call "change" and "coins") are: circles, triangles and pentagons. A triangle is worth 3 circles and a pentagon is worth 5 circles.[3]

Assume that a change box exists that's stocked with a number of circles, triangles and pentagons to return as change. Next, assume that you're given a price of an item and an amount of change that's been paid by the customer. You're going to design a combinational circuit that does two things:

- Determine if there's enough money in the change box to make complete change.
- Determine the two largest single coins that should be given out as part of the change. For some reason, this machine can only return 2 coins, so even if it is theoretically possible to make more change, limit it to only 2 coins.

For this lab, you will use procedural SystemVerilog descriptions to let you think about the solutions at a higher-level than the gate-level structural descriptions you used in previous labs. Your solution will most likely be many modules describing a multi-level hierarchical description.

Specifications

From the inputs specified below, generate the specified combinational outputs. As listed here, there will be 5 multi-bit inputs and 6 multi-bit outputs for your circuit. All numbers are unsigned. Each of the inputs and outputs are described below:

`Cost[3:0]` Four-bit input representing the cost of the item purchased in increments of circles. The maximum cost is 15 circles.

`Paid[3:0]` Four-bit input representing the amount of money paid by the customer in increments of circles. The maximum amount paid is 15 circles.

`Pentagons[1:0]` Two-bit number representing the number of pentagons available to make change with.

`Triangles[1:0]` Two-bit number representing the number of triangles available to make change with.

`Circles[1:0]` Two-bit number representing the number of circles available to make change with.

`FirstCoin[2:0]` This is the value (in circles) of the first coin to be returned. This can have the values **`3'b001`**, **`3'b011`**, or **`3'b101`** indicating circle, triangle, or pentagon, respectively. It can also have the value **`3'b000`** indicating that no coin should be given (no change is necessary).

[3] Have you noticed the relationship between the coin's value and the its geometry?

This should be the largest coin that can be returned and which is available in the change box. For example, if the person is due 8 circles, **FirstCoin** should indicate **3'b101** meaning that a pentagon should be returned. But it will only indicate that if there is a pentagon available (see **Pentagons** above). If there is no pentagon available, the **FirstCoin** should indicate **3'b011** for a triangle instead. (See discussion later for if there isn't a triangle available.)

SecondCoin[2:0] This is the value (in circles) of the second coin to be returned. This can have the values **3'b001**, **3'b011**, or **3'b101** indicating circle, triangle, or pentagon, respectively. It can also have the value **3'b000** indicating that no coin should be given (no change is necessary, or all of the change is covered by **FirstCoin**). This should be the largest coin that can be returned and which is available in the change box. For example, if the person, er... being, is due 10 circles, **FirstCoin** should indicate pentagon (assuming there is one) and **SecondCoin** should indicate a pentagon as well (assuming there is a second pentagon available). If there are no pentagons, then both **FirstCoin** and **SecondCoin** should both indicate triangle (assuming there are triangles available). (And of course, in this case, the Zorgian wouldn't get his/her/its full change.)

ExactAmount This is a single-bit output that, when asserted, indicates that the amount entered is equal to the cost. Thus, no change is necessary (i.e., **Paid == Cost**, and neither are **0**). Asserted means that the LED should light.

NotEnoughChange This is a single-bit output that, when asserted, indicates that change was needed and the first two coins returned (if they exist) were not enough. Asserted means that the LED should light. (Note that this has nothing to do with giving out two triangles instead of a pentagon. The issue here is that we owe the person something – they might punch out the machine!)

CoughUpMore This is a single-bit output that, when asserted, indicates that **Paid** is less than **Cost**. Asserted means that the LED should light.

Remaining[3:0] This is the amount in circles that indicates how much is left to give the person in change. (i.e., **Paid** – **Cost** – **FirstCoin** – **SecondCoin**)

All of your SystemVerilog should be written in a procedural style, using **always_comb** and **assign**. You should have NO individual gates in your SystemVerilog.

Hierarchical Design Development

Hopefully, I've motivated you sufficiently to think about your design before you jump in and start coding. Make sure you understand which submodules you will build and the complete functionality of each. Draw a complete high-level system diagram including all signal and wire names connecting the submodules. Your design will almost certainly have more than two levels of submodules.

Continue thinking about your design on paper (don't touch the SystemVerilog yet!). This problem doesn't actually take much work, if you end up thinking about it the right way.

First draw a diagram that contains submodules that do little pieces of the whole calculation. This is a circuit diagram of the major combinational blocks of your design. Note the word blocks here — we're not looking for gates. Rather we're looking for the major functional pieces as suggested above (e.g., **`Change=Paid-Cost`**). Design at this submodule level, putting a single calculation or two in each submodule. Like a combinational circuit, the input values should "flow" left-to-right on your diagram resulting in the outputs. There should be no feedback as combinational logic does not contain feedback! (Another way to say the same thing: the graph of computational blocks should be acyclic.) This system level diagram will serve as the plan for your whole design. Then begin coding each submodule in SystemVerilog. The computation in each submodule should become an **`assign`** or **`always_comb`** block (or possibly several) in a SystemVerilog module. (No primitive logic gates like in lab 1.) Test each submodule before moving on.

Simulate your design (no, we don't want a print out of the whole truth table!). Convince yourself it works before trying to download it. How did you do this? You probably don't need to show all rows of the truth table to yourself either, so make a case to yourself (and to us) how you tested the pieces of your design as well as the whole thing. Be systematic. One way to think about this is to look for corner cases for the various modules and check them. For instance, if a situation where keywordPaid > Cost works, you don't have to test every situation where keywordPaid > Cost. Just check that the output of the module is correct for the three different cases (<, >, ==). And check that the modules it connects to works for these three situations.

Synthesis and Download

To complete the lab demo, you need to download your module to your FPGA board and demonstrate it. For the demonstration, you need to include a top level module that interfaces your module to the particular inputs and outputs available on the board (switches, LEDs, etc). This module should look like:

```
module ChipInterface
  (output logic [6:0] HEX7, HEX6, HEX5, HEX4,
   output logic [17:0] LEDR,
   input logic [17:0] SW);

  // Your code here.

endmodule : ChipInterface
```

Your module should make the following connections:

Inputs:	Outputs:
Cost[3:0] uses **SW[17:14]**	**FirstCoin[2:0]** drives **HEX7**
Pentagons[1:0] uses **SW[13:12]**	**SecondCoin[2:0]** drives **HEX6**
Triangles[1:0] uses **SW[11:10]**	**Remaining[3:0]** drives **HEX5** and **HEX4**
Circles[1:0] uses **SW[9:8]**	**ExactAmount** uses **LEDR17**
Paid[3:0] uses **SW[3:0]**	**NotEnoughChange** uses **LEDR16**
	CoughUpMore uses **LEDR15**

Note that **Remaining[3:0]** is a hex value for which all values are valid. Display it as two BCD values on **HEX5/4**. If the most-significant digit is zero, blank the 7-segment display.[4]

All other **HEX** and **LED** outputs should be off. You are welcome to use them for debugging, but turn them off for your demo.

Try to make sure your first demo to your TA will be your only (and correct) demo. Use a well-chosen set of test cases to convince your TA that your design is working. How big is your entire design? How can you get some idea of circuit size from the Quartus II tool?

[4] Hmm... Where do you suppose you can get something that drives a 7-segment display?

Testing

As the design becomes larger, it becomes much harder to exhaustively test it. One has to do what is called *unit testing* where each submodule of a design is fully tested and then only a smaller number of well-chosen test cases are needed at the top level to verify the design's overall correctness. This is one of the many added benefits of hierarchical design and design partitioning. In this lab, you are ****required**** to use simulation to test each module separately before putting them together.

After each module has been individually tested go ahead and assemble the full design. You now need to simulate and test your full design at the top level. We don't want to see a print out of the whole $2^{whatever}$ entry truth table, or even a for-loop that exhaustively tests the entire truth table! Design your test inputs intelligently. Make use of the fact that you have tested the submodules individually.

Hints and Happy Thoughts

Think about implementing this lab like you did Lab 1. Then be happy you are not doing truth table or gate-level designs. The time spent running the synthesis tool is far less than the time to design and "wire up" this in gate level SystemVerilog, and far less error prone. Use the simulator outside of lab to get the design finished and working. Come into lab to demonstrate your project. Work smart. Try to have the design correct before downloading to the board. Strive to get it right the first time.

Use the waveform viewer to help you visualize the components and how they are connected throughout the design and for the time of your testbench execution.

Test submodules thoroughly with well thought-out test cases before using them to build bigger things. This strategy is common and called "unit testing."

If there are problems that you cannot see a way to solve, TA and office hours is a good way to find helps/hint. Do not hesitate to come see us.