18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

Lab2: Advanced Comb. Logic Design – Magic Squares

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.

19 – 21 September	Start your design. System diagram due in lab.
26 – 28 September	Simulation results due at start of lab. Final demos due in lab.
2 – 3 October	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.

Theory: Magic Squares

8	1	6
3	5	7
4	9	2

A *magic square* is a popular construct in recreational mathematics. It is an arrangement of n^2 distinct integers into an n by n square such that the sum of all rows, columns and both diagonals is the same constant value. The integers used in the formation of a normal magic square are in the range 1 to n^2. All 8 possible normal magic squares of order 3 are reflections or rotations of the example shown. Notice how, for each row, column and diagonal, the sum of the three values is the magic constant value, 15.

Your Design Task

In this lab, armed with your advanced combinational logic design skills, you will implement a circuit for checking if a square is a normal magic square or not. In other words, when given nine decimal-digit values stored in nine cells of a 3 x 3 square (numbered from the top left in row-major form), your circuit will assert a signal when the nine values happen to form a normal magic square. In addition, your circuit will output the magic constant value of the circuit.

In order to check for if the square is magic or not, your circuit will need to ensure that the 9 values entered are all valid (1-9) and unique. You will then need to check that the sums of each row, column and diagonal are all equal — do not just check them individually against the magic constant (15).

Each digit of the square is BCD encoded, yielding an astonishing 36 bits of input for the combinational circuit. Your circuit will output a single signal to check the magicness[3] of the square and two BCD sum values for a grand total of 9 bits of output. That's a pretty big truth table one would need to draw to fully describe this combinational circuit. Before you start pulling out 36-dimensional K-maps or running a huge Q-M algorithm 9 times, you'll find that divide-and-conquer is your friend. Sit down and think about this circuit. You can probably discover some patterns and structure in this seemingly immense combinational circuit. For instance, adding up the 3 BCD values on the top row is pretty much the same job as adding up the 3 BCD values from the second row. Another example is: "Checking for magicness involves determining if the 9 values meet each of the constraints described in the definition of a normal magic square of order 3." By decomposing the problem into smaller (and hopefully more manageable) modules, you will form a hierarchical approach to solving the entire problem.

For this lab, you will use procedural, behavioral 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.

As a further example of how the decomposition process develops this multi-level hierarchy, consider the job of computing the sum of the digits in the first row. This job consists of adding **`num1`**, **`num2`** and **`num3`**, each of which is a 4-bit BCD value. At the first level of decomposition, the triple-sum can be broken down into a module that adds **`num1`** to **`num2`** and another module that adds the result to **`num3`**. When designing the module that adds 2 BCD values, you will see that each sum can produce a two-digit BCD value, so a BCD adder should have 16-bits of input and 8-bit output. Again, before

[3] Yep, I'm inventing words again.

firing up a 16-dimensional K-map, think about how to decompose this task. Perhaps you should design a single-BCD value adder, that takes a carry-in and produces a carry-out. By decomposing the original 36-bit input task, you end up with a number of modules, each of which is significantly simpler to design.

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.

Specifications

The combinational checker module you need to design has the following interface:

```
module IsMagic
  (input  logic [3:0] num1, num2, num3, //top row, L to R
   input  logic [3:0] num4, num5, num6, //middle row
   input  logic [3:0] num7, num8, num9, //bottom row
   output logic [7:0] magic_constant,   //2 BCD digits
   output logic       it_is_magic);
```

The input values **num1** through **num9** represent the 9 cells of the 3x3 square, with **num1** in the top left, **num5** in the center and **num9** in the lower right. If the 9 numbers match all requirements of a normal magic square of order 3, then **it_is_magic** shall be asserted and the magic constant shall be output on **magic_constant** as a 2-digit BCD value. In all other cases, the value on **magic_constant** is undefined, though you might find it useful to use for debugging.

For this lab, you are required to NOT include any combinational logic in the **IsMagic** module itself. You are only allowed to instantiate submodules[4] and to specify signal connections between the submodules and the input / output ports.

[4] I'm using the term *submodule* here to designate a decomposed portion of the design. SystemVerilog doesn't know anything about the term submodule.

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.

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.

You are required to specify and incorporate the following two modules in your design (in addition to other modules of your own design). You might have seen one of these before somewhere.

```
module Comparator
  (input  logic [3:0] A, B,
   output logic       AeqB);

module BCDOneDigitAdd
  (input  logic [3:0] A, B,
   input  logic       Cin,
   output logic [3:0] Sum,
   output logic       Cout);
```

These two modules should be quite useful to you in computing the sums of the rows, columns and diagonals as well as determining if they are the same.

Synthesis and Download

To complete the lab demo, you need to download the **`IsMagic`** module to your FPGA board and demonstrate it. For the demonstration, you need to include a top level module that interfaces **`IsMagic`** 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, // magic_constant
                       HEX5, HEX4, HEX3, HEX2, HEX1, HEX0,
   output logic  [7:0] LEDG,
   input  logic [17:0] SW,
   input  logic  [3:0] KEY,
   input  logic        CLOCK_50); // needed for enter_9_bcd

  logic [3:0] num1, num2, num3,
              num4, num5, num6,
              num7, num8, num9;
  logic [7:0] magic_constant;
  logic       it_is_magic;
```

```
  enter_9_bcd e(.entry(SW[3:0]),
                .selector(SW[7:4]),
                .enableL(KEY[0]),
                .zeroL(KEY[2]),
                .set_defaultL(KEY[1]),
                .clock(CLOCK_50),
                .*);

  IsMagic im(.*);

  // Your code here.
  // Output it_is_magic to all 8 bits of LEDG
  // Display magic_constant on the 7 segment display

endmodule : ChipInterface
```

The **enter_9_bcd** module is provided to you in **/afs/ece/class/ece240/handout/lab3/**. This module allows one digit at a time to be entered by the user. **SW3** to **SW0** specify the BCD value to be entered (i.e. the value that goes into one of the square's cells). **SW7** to **SW4** select which digit is being entered (valid inputs are **4'b0001** through **4'b1001**). **KEY0** is then pressed to tell the **enter_9_bcd** module to remember the value for the selected digit. For instance, when **SW[7:0]** is **8'b0011_0110** and **KEY0** is pressed, the **enter_9_bcd** module will remember that **num3** is a BCD **6**. Note that the **num** values are counted from **4'0001**, not from zero.

In order to simplify the user interface a little bit, the **enter_9_bcd** module has two other buttons hooked up that will come in handy. If you press **KEY2**, all 9 values will be reset to zero. If you press **KEY1**, the magic square shown in the figure on page 1 will be loaded.

Whenever the values entered correspond in all requirements to a normal magic square of order 3, your circuit should light up all 8 green LEDs. At all times (i.e. if the values are a magic square or not), **magic_constant** should appear on the first 2 digits of the 7-segment display (i.e. **HEX7** and **HEX6**). The remaining digits can be any value you need (or none at all).

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?

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.