Homework #5 – Digital Logic Design

Due October 23rd at 5:00pm Eastern Time

Directions:

- For short-answer questions, submit your answers in PDF format to GradeScope assignment "Homework 5 written".
    - Please type your solutions. If hand-written material must be included, ensure it is photographed or scanned at high quality and oriented properly so it appears right-side-up.
    - Please include your name on submitted work.
- For Logisim questions, submit .circ files via GitLab or direct upload to GradeScope assignment "Homework 5 code":
    - **Circuits will be tested using an automated system, so you must name the input/output pins exactly as described, and submit using the specified filename!**
    - **You may only use the basic gates (NOT, AND, OR, NAND, NOR, XOR), D flip-flops, multiplexers, splitters, tunnels, and clocks. Everything else you must construct from these.**
    - Circuits that show good faith effort will receive a minimum of 25% credit.
- **Start by cloning the "homework5" git repo, similar to past assignments.**
- A Logisim Evolution circuit self-tester has been provided. It works much the same as previous self-test tools; you just need to have your .circ files in the directory with the tester. The tester is known to work in the Duke Linux environment, but may possibly work elsewhere. Additional info on the tester is included in three appendices at the end of this document. There are a few things that need to be done for the tester to work correctly:
    - Name the files and label the pins as per the directions given. The self-tester will NOT WORK with different names or labels.
    - For the FSM question, use the clock available in Logisim Evolution to run the DFFs.
    - Additionally, to run the self-tester you will have to place the Logisim Evolution files in the same folder as the python script, the jar file and the folder labelled tests.
    - You can use the command `./hwtest.py` in the following manner:

      ```
      ./hwtest.py <arguments>
      The following arguments can be used with that command:
      - ALL: Runs all the tests
      - circuit1a: Runs tests for circuit1a.circ
      - circuit1c: Runs tests for circuit1c.circ
      - my_adder: Runs tests for my_adder.circ
      - robot: Runs tests for robot.circ
      ```
    - Lastly, remember that the tests cases provided are not exhaustive so testing more cases manually would be recommended.
- **You must do all work individually, and you must submit your work electronically via GradeScope**.
    - All submitted circuits will be tested for suspicious similarities to other circuits, and the test will uncover cheating, even if it is "hidden."

Q1. Boolean Algebra

(a) [5 points] Write a truth table for the following function: Output=[(!C·B) ⊗ (A ⊗ (BC))] + A·!C

(b) [10] Use Logisim Evolution to implement and test the circuit from (a). Name this file circuit1a.circ. Your circuit must have the following pins:

Label	Type	Bit(s)
`A`	input	1
`B`	input	1
`C`	input	1
`result`	output	1

(c) [5 points] Write a sum-of-products Boolean function for the output (out) in the following truth table and then minimize it using Boolean logic, de Morgan's laws, etc. (You should use only AND, OR, and NOT gates.) You do NOT have to have a perfectly optimal circuit, but you must show some optimizations.

A	B	C	out
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(d) [10] Use Logisim Evolution to implement and test the circuit from (c). Name this file circuit1c.circ. Your circuit must have the following pins:

Label	Type	Bit(s)
`A`	input	1
`B`	input	1
`C`	input	1
`result`	output	1

Q2. Adder/Subtractor Design

[30] Use Logisim Evolution to build and test a 16-bit ripple-carry adder/subtractor. You must first create a 1-bit full adder that you then use as a module in the 16-bit adder. The unit should perform `A+B` if the `sub` input is zero, or `A−B` if the `sub` input is 1. The circuit should also output an overflow signal (`ovf`) indicating if there was a signed overflow.

Name the file my_adder.circ. Your circuit must have the following pins:

Label	Type	Bit(s)
A	input	16
B	input	16
sub	input	1
result	output	16
ovf	output	1

Note: To split the 16-bit inputs and to combine the individual outputs of the one-bit adders together, use Splitters.

Q3. Finite State Machine

You're an engineer at a robotics company that works with industrial robot arms, and you have been tasked to produce a finite state machine to control the arm. The arm moves left and right within a certain range of 4 positions. It starts at the far left at position 0, and it can go as far right as position 3. If you try to move the robot left from position 0, it just stays at position 0. Similarly, if you try to move the robot right from position 3, it just stays at position 3.

At the start, the robot is not moving. The robot has a speed input that specifies how many positions the robot moves in one cycle. The robot starts at speed 0 (but is not moving, so this is the speed it will use by default once it starts moving, unless the speed is changed).

You have two inputs: one enables you to choose between moving left and right, and the other enables you to control the speed. You have two outputs: the position of the robot and a signal to denote whether the robot was unable to move due to already being at the farthest position.

The formal names you must use in your circuit are shown below:

Pin name	Type	Meaning
in_leftright	2-bit input	01 if we want robot to move left; 10 if we want robot to move right; 00 if we don't want robot to move at all; never 11
in_speed	1-bit input	0 means move by one position per cycle; 1 means move by two positions per cycle. If in_speed=1 but only one position is available for movement (i.e., if requested to go left from position 1 or right from position 2), the robot will only move one position and then stop.

out_position	2-bit output	Current position of robot. Starts at 0. Cannot be less than 0 or greater than 3.
out_blocked	1-bit output	Set to 1 if robot was unable to move at all because it was requested to go left while at left end (position 0) or requested to go right while at right end (position 3). Set to 0 otherwise. Note that out_blocked is set to 0 if the robot is able to move but not as far as requested (i.e., if it is at in_speed=1 but only one position is available for movement).

To achieve the above goals, the finite state machine you make will follow these rules:

1. When the system starts up, the robot is not moving. Please call this state "start".
2. Implement your FSM as a "Moore" machine, meaning that the output should depend *only* on the current state and not on the current inputs.

For full credit, you must use the systematic design methodology we covered in class:

(a) [8] Draw a state transition diagram, where each state has a unique identifier that is a string of bits (e.g., states 00, 01, etc.) as well as the associated value for the outputs. Label all of the arcs between transitions with the inputs that cause those transitions. You may abbreviate the inputs and outputs on your diagram if you wish.

(b) [8] Draw a truth table for the state transition diagram. From a truth table perspective, the inputs are **in_leftright** and **in_speed,** and the current state bits (**Q0**, **Q1**, etc.); the outputs are **out_position** and **out_blocked**, and the next state bits (**D0**, **D1**, etc.).

(c) [4] Write out the logic expressions for your next-state bits (**D0**, **D1**, etc.) as well as the outputs. NOTE: Optimization here is *optional*. You may even use automated Boolean optimization tools if you wish, provided you cite and screenshot them in your write-up.

(d) [30] Use Logisim Evolution to implement and test this circuit. Name this file robot.circ. Your circuit must have the pins described in the earlier table, named precisely as shown.

Tip:

- Run a "Clock" component to all the clock inputs in the DFFs.

Appendix: Getting the tester to work locally

The tester will work out-of-box on the Docker environment.

However, if you want to test locally, you need the right version of Java set up and in your PATH. Note: support for this is best-effort; if you have trouble we can't resolve, you have the docker environment.

For Windows (with Ubuntu in Windows Subsystem for Linux) or Ubuntu Linux

We just need to install Java Runtime Environment 1.8, then update our config to use that Java by default. This will only effect your Linux-on-Windows environment.

```
sudo apt-get install -y openjdk-8-jre
sudo update-alternatives --config java
```

After the second command, you'll be asked to pick a Java. By number, choose "`/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java`".

You may get one spurious fail from the tester after initial setup, as the first time run it will print a little message. Subsequent tests runs should function normally.

For Mac

Mac machines tend to have a few different Javas lying around, and the tester does its best to find a suitable one. In the assignment directory, try:

```
java -jar logisim_ev_cli.jar
```

If you see "`error: specify logisim file to open`", you're good to go. If you see some big ugly crash, you probably need to switch Java versions. You likely have the required version on your system by virtue of having installed Logisim Evolution. Java 1.8.0 is known to work. Try following these directions to switch Java versions.

If you don't have an appropriate version of Java installed, you can install OpenJDK 8 from here.

Appendix: Tester info

The test system for Logisim Evolution assignments uses the same front-end tool as earlier assignments, but to have it control Logisim Evolution, a special command-line variant of Logisim Evolution is packaged with it. (Thanks for reading the assignment in full; put a picture of a possum in your Q1a solution for extra credit.) When you use the tester, it runs this with your circuit and a number of command-line options that tell it how to set the inputs to your circuit and how to print the outputs.

You can review the tests details by looking inside settings.json. If you see a line like this for my_adder:

```
{ "desc": "A=0x9BDF, B=0x8ACE, sub=0",
     "args": ["-c", "0", "-ip", "A=0x9BDF,B=0x8ACE,sub=0", "-of", "h"] },
```

Then it will run this:

```
java -jar logisim_ev_cli.jar -f adder.circ -c 0 -ip A=0x9BDF,B=0x8ACE,sub=0 -of h
```

The options run the circuit for 0 cycles (as it has no clock so there's no need to run it over time), set pins A and B to the given hex values and sub to zero, and set the output format to hex. The output will look like:

```
0       out         ovf                     0x01
0       out         result                  0x26ad
```

The fields are: cycle number, the type of output ("out", "probe", or a few others), the name of the pin ("ovf" and "result" here), then the value at that time.

For sequential circuits, output is shown per clock cycle, such as this example for the finite state machine:

```
0      out     out_position      00

0      out     out_blocked       0

1re    out     out_position      01

1re    out     out_blocked       0

2re    out     out_position      11

2re    out     out_blocked       0
```

Using this information, you can interpret the `actual` and `expected` files (and the resulting `diff`).