

# Homework #3 – Digital Logic Design

Due date: see course website

Directions:

- For short-answer questions, submit your answers in PDF format to GradeScope assignment “Homework 3 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 3 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 “homework3” 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
    - press: Runs tests for press.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:  $\text{Output} = (!C \cdot (!A \cdot !B + A \cdot B)) + !A \cdot !B \cdot C + B \cdot C$
- (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 both outputs in the following truth table and then minimize them 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	out1	out2
0	0	0	1	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	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
out1	output	1
out2	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)
<b>A</b>	input	16
<b>B</b>	input	16
<b>sub</b>	input	1
<b>result</b>	output	16
<b>ovf</b>	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 – Moore machine

We desire to design a vending machine to dispense energy pellets to eager computer science and engineering students working late on their important homework assignments. We only have one flavor of pellet (so, unlike a more full-featured vending machine, we don't need to specify which kind of pellet we want, an obvious future extension!). Our vending machine will be controlled by a Moore Finite State Machine.

Each pellet costs, after allowing for a suitable profit to those of us who designed the machine, 15 cents. Our machine accepts nickels and dimes only.

Our machine does NOT make change. It will accept any combination of nickels and dimes until (at least) 15 cents is deposited, at which time it will set its DISPENSE output to 1, causing exactly one pellet to be ejected.

Never missing an opportunity to comment on human behavior, our machine also notes two additional conditions.

1. If a user inputs 20 cents to purchase their 15-cent pellet, the machine additionally activates a DESPERATE output to acknowledge the overpayment (and the extra 5 cents profit for us). Activating this signal causes a light to turn on and appropriate noises to be played, but you don't worry about that part, just turn on the signal.
2. If a user inputs exactly Nickel – Nickel – Dime, the machine additionally activates a DOH output to acknowledge the, um, lack of wisdom of this particular user. Again, you just have to get the DOH signal turned on, other parts of the machine will take care of the appropriate lights and noises.

After a pellet is dispensed, and mockery activated if warranted, the next user must press a RESET button to reset the machine to be ready for the next customer. If RESET is pressed before any coins have been inserted, it is ignored. It is also ignored if we have currently seen 5 or 10 cents deposited in the machine so far.

Our single coin slot guarantees that either the N signal for Nickel or D signal for Dime is activated as each coin is inserted; it is impossible for N and D to be on at the same time. When no coins are present, both N and D are 0.

**Implement your FSM as a “Moore” machine**, meaning that the output should depend only on the current state and not on current inputs. In other words, your output should be written on the nodes in the state transition diagram rather than on the edges.

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

Pin name	Type	Meaning
<b>N</b>	1-bit input	A Nickel has been detected in the coin slot
<b>D</b>	1-bit input	A Dime has been detected in the coin slot
<b>RESET</b>	1-bit input	Pushed to reset machine after a pellet is issued
<b>DISPENSE</b>	1-bit output	Set to cause the machine to dispense exactly one pellet (fully automatically)
<b>DESPERATE</b>	1-bit output	User paid 20 cents for pellet, mockery ensues
<b>DOH</b>	1-bit output	User paid exactly N-N-D, additional mockery ensues

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

- [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 outputs **motor** and **warning**. Label all of the arcs between transitions with name(s) of the input(s) that cause those transitions.
- [8] Draw a truth table for the state transition diagram. From a truth table perspective, the inputs are N, D, RESET, and the current state bits (**Q0**, **Q1**, etc.); the outputs are DISPENSE, DESPERATE, DOH, and the next state bits (**D0**, **D1**, etc.).
- [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.
- [30] Use Logisim Evolution to implement and test this circuit. Name this file vend.circ. Your circuit must have the pins described in the earlier table, named precisely as shown.

Tips:

- Run a “Clock” component to all the clock inputs in the DFFs.
- A compliant circuit will look something like this:

## 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      motor     0  
0      out      warning   0  
  
1re    out      motor     0  
1re    out      warning   0  
  
2re    out      motor     1  
2re    out      warning   1  
  
3re    out      motor     1  
3re    out      warning   1  
  
4re    out      motor     0  
4re    out      warning   1  
  
5re    out      motor     0  
5re    out      warning   0
```

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

## Appendix: `press` test case details

Because the test cases for `press` are a bit long and the command line option format is somewhat cryptic, here's a nicer presentation of them.

Test cases 0-3 simply apply constant values to the inputs – these are shown in the test description.

Test cases 4-7 apply changing values to the inputs to try to put the finite state machine through its paces.

Test case 8 is long and randomly generated.

The tables below show these cases. Here, “#” is the cycle in which the input is changed.

### Test 4

#	go	limit
2	0	1
4	1	1
6	1	0
8	1	1
10	0	1
12	0	0

### Test 6

#	go	limit
0	0	1
3	1	1
6	0	1
9	1	1
12	0	1

### Test 5

#	go	limit
2	1	1
3	1	0
4	1	1
5	0	1
6	0	0

### Test 7

#	go	limit
0	0	0
3	1	0
6	0	0
9	1	0
12	0	0



**Test 8**

#	go	limit
2	1	1
3	1	1
4	1	1
5	0	1
6	1	0
7	1	1
8	0	0
9	0	1
10	1	0
11	0	0
12	0	0
13	1	0
14	1	0
15	1	1
16	1	0
17	1	0
18	1	0
19	1	1
20	0	0
21	0	0
22	1	0
23	1	1
24	1	0
25	1	1
26	0	0
27	1	1
28	0	0
29	1	0
30	0	1
31	0	0
32	1	0
33	0	0
34	1	1
35	0	1
36	1	0
37	0	0
38	1	1
39	0	0
40	1	1
41	1	0
42	0	1
43	1	0
44	0	0
45	1	1
46	1	1
47	0	1
48	1	1
49	0	0
50	0	1
51	0	1
52	1	1
53	0	1
54	1	0

55	1	0
56	0	1
57	0	0
58	0	1
59	0	0
60	1	1
61	1	0
62	0	0
63	0	1
64	0	1
65	0	0
66	1	1
67	0	0
68	0	1
69	1	1
70	1	1
71	1	0
72	0	1
73	1	1
74	0	1
75	0	0
76	1	1
77	1	0
78	1	1
79	0	0
80	0	0
81	1	1
82	0	1
83	0	1
84	0	0
85	0	1
86	1	0
87	0	1
88	0	1
89	0	0
90	0	1
91	0	1
92	0	0
93	0	1
94	1	0
95	0	0
96	0	0
97	0	1
98	0	0
99	1	1
100	1	1
101	1	1