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
- ignition: Runs tests for ignition.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=((!A+!B)·!C) + ((A·!B) + (!C·B))

(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	0	1
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
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)
`A`	input	16
`B`	input	16
`sub`	input	1
`result`	output	16
`ovf`	output	1

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

Q3. Finite State Machine

Before:

After:

You're an engineer at a company that makes products to add modern amenities to classic cars. You've been asked to develop a kit to retrofit a car with traditional turn-key ignition to have modern electronic push-button ignition.

Another team is handling the electronic radio-frequency based key; you just need to handle the logic of actually starting and stopping the engine when requested.

To do so, you should understand a little bit about how cars and engines work (though I'll skip and simplify a lot). An internal combustion engine works by injecting a mist of gasoline and air into a cylinder, then exploding it by means of an electric spark passed over a spark plug. The voltage for this spark is provided by the **ignition coil**, which converts the low voltage of the car's electrical system into a high voltage to create an arc when needed. When the car is running, the reaction is self-sustaining (using an alternator to generate electricity from the gasoline engine's movement and a distributor to fire the piston's spark plugs in precise timing), but how do we actually *start* the car?

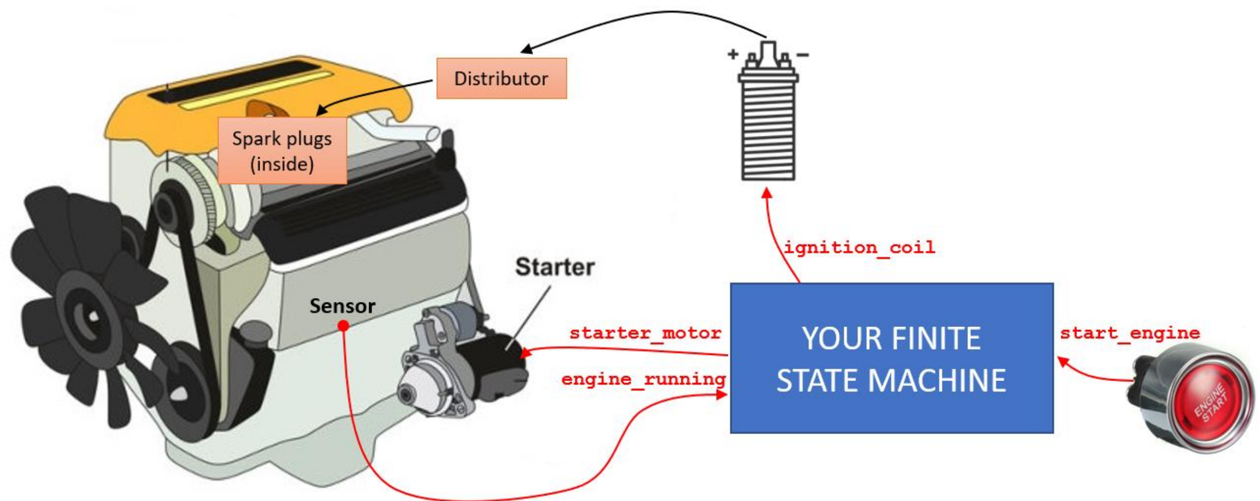
To do this, a small electric motor called the **starter motor** is used to turn the engine electrically until the pistons are firing normally and the reaction becomes self-sustaining. This is what you hear when you start a car – the "chug chug chug" is the starter forcing the engine to turn until the gasoline combustion reaction is running regularly.

In a classic key-turn ignition, the key energizes the starter motor *and* ignition coil when turned up to "start", and the operator holds it there until they hear that the engine is running, then releases it back to "on", whereupon the starter is disabled but the ignition coil stays energized so the engine keeps running. When the key is turned to "off", the ignition coil is disabled, so there are no more sparks, so no more gasoline detonations, and the engine stops.

In the push-button ignition system you're developing, a finite state machine will do this task. To do so, we'll need two inputs: the **start button** itself and an **"engine running"** sensor to indicate when the engine is fully started and running properly. In terms of outputs, there will be the **starter motor** and the **ignition coil**. The formal names you must use are shown below:

Pin name	Type	Meaning
`start_engine`	1-bit input	1 if the button is pressed, else 0.
`engine_running`	1-bit input	1 if the engine is running, else 0.
`starter_motor`	1-bit output	Set to 1 to energize the starter motor.
`ignition_coil`	1-bit output	Set to 1 to energize the ignition coil.

Note: This document sometimes describes the **`start_engine`** input as a button. This is meant physically in real life – you should *not* model it as a Logisim button ( ), but rather as a regular input pin ( )!



The rules governing this system are as follows.

1. When the ignition system begins, the car is assumed to be off. The **ignition_coil** and **starter_motor** are off.
    - The **engine_running** signal is ignored here.
    - If the **start_engine** button is pressed, begin starting the engine as described in #2 below.
    - Otherwise, remain in this state.
2. When the system is starting the car, the **ignition_coil** and **starter_motor** should be on.
    - As long as **engine_running** is off, remain in this state.
    - To allow the user to "force" the starter to keep running (e.g. if the engine has trouble staying running for the first few seconds on a cold day), remain in this state as long as **start_engine** remains held down, regardless of the **engine_running** signal.
    - When **engine_running** is true and the **start_engine** button has been released, the car is running as described in #3 below.
3. When the car is running, the **ignition_coil** should be on, and the **starter_motor** is off.
    - If the engine spontaneously shuts off (i.e., the **engine_running** sensor turns off), this is known as a *stall*, and the system should head directly back to the "off" condition described in #1 above; this is true regardless of the **start_engine** button state.
    - If, while **engine_running** is true, the user presses the **start_engine** button, the user wishes to turn off the car, and so the system should proceed to the "stopping" state as described in #4 below.
    - As long as **engine_running** is true and **start_engine** is false, remain in this state.
4. When turning the car off, both the **ignition_coil** and **starter_motor** should be disabled.
    - The system should remain in this state until the engine is stopped (**engine_running** becomes false) and the button is released (**start_engine** becomes false), at which time it will revert to the "off" state described in #1 above.

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 outputs **`starter_motor`** and **`ignition_coil`**. Label all of the arcs between transitions with the inputs **`start_engine`** and **`engine_running`** that cause those transitions. You may abbreviate the inputs and outputs (**`starter_motor`**="SM", **`ignition_coil`**="IC", etc.) on your diagram if you wish.

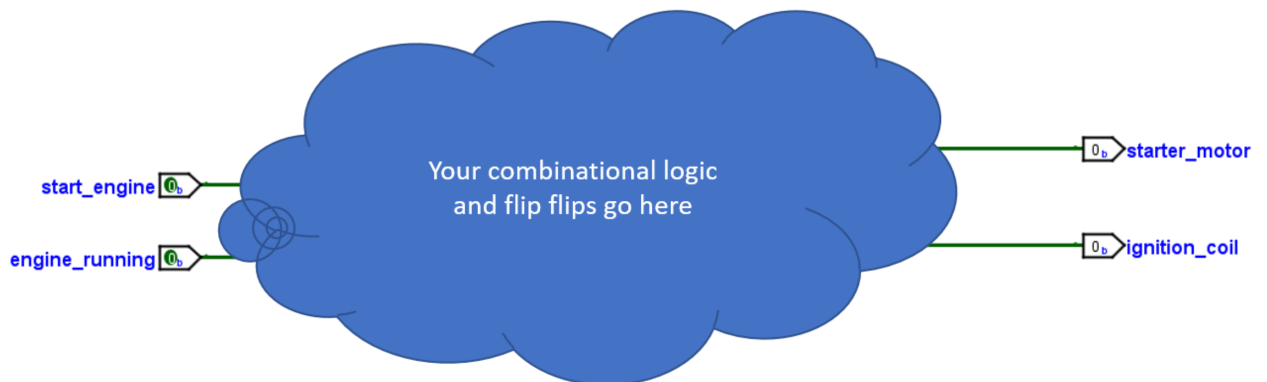
(b) [8] Draw a truth table for the state transition diagram. From a truth table perspective, the inputs are **`start_engine`**, **`engine_running`** and the current state bits (**`Q0`**, **`Q1`**, etc.); the outputs are **`starter_motor`**, **`ignition_coil`**, and the next state bits (**`D0`**, **`D1`**, etc.).

(c) [4] Write out the logic expressions for your next-state bits (**`D0`**, **`D1`**, etc.) as well as the outputs **`starter_motor`** and **`ignition_coil`**. 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 ignition.circ. Your circuit must have the pins described in the earlier table, named precisely as shown.

Tips:

- Implement your FSM as a "Moore" machine, meaning that the output should depend *exclusively* on the current state. In other words, your output should be written on the state nodes in the state transition diagram rather than on the edges. When writing the truth table for this, the **`starter_motor`** and **`ignition_coil`** columns should be based *just* on the current state columns.
- Run a "Clock" component to all the clock inputs in the DFFs.
- A compliant circuit will look something like this:



Note: I oversimplified and omitted a lot of details about cars here. If you're a car knower and you're bothered by the technical accuracy of this problem, then: sorry I didn't make the problem more complicated and therefore harder, I guess?

Appendix: Getting the tester to work locally

The tester will work out-of-box on the login.oit.duke.edu environment (if run as "python tester.py") and 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 above two environments.

For Windows (with Ubuntu in Windows Service 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. 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         ignition_coil          0
0        out         starter_motor          0

1re      out         ignition_coil          1
1re      out         starter_motor          1

2re      out         ignition_coil          1
2re      out         starter_motor          1

3re      out         ignition_coil          1
3re      out         starter_motor          1

4re      out         ignition_coil          1
4re      out         starter_motor          1

5re      out         ignition_coil          1
5re      out         starter_motor          1
```

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

Appendix: Ignition test case details

Because the test cases for plastic 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

#	start_engine	engine_running
2	1	0
4	0	0
6	0	1
8	1	1
10	0	1
12	0	0

Test 5

#	start_engine	engine_running
2	1	0
3	0	0
4	0	1
5	1	1
6	0	1
7	0	0

Test 6

#	start_engine	engine_running
2	1	0
4	0	1
6	0	0
8	1	0
10	1	1
12	0	1

Test 7

#	start_engine	engine_running
2	1	0
4	1	1
7	0	1
10	1	1
13	1	0
16	0	0
19	1	0
22	1	1

Test 8

#	start_engine	engine_running
2	0	0
3	1	1
4	1	1
5	1	1
6	1	0
7	1	1
8	1	0
9	0	1
10	0	0
11	1	1
12	0	1
13	1	0
14	0	1
15	0	1
16	0	0
17	1	0
18	0	0
19	1	0
20	0	1
21	1	0
22	1	0
23	0	1
24	1	1
25	0	0
26	0	1
27	0	0
28	0	0
29	1	0
30	1	1
31	1	0
32	0	1
33	0	0
34	1	0
35	0	0
36	0	1
37	1	0
38	1	0
39	0	0
40	1	1
41	1	1
42	0	1
43	0	0
44	1	1
45	0	1
46	0	1
47	0	1
48	1	0
49	1	1
50	0	0
51	0	1
52	1	1
53	1	0
54	0	0
55	1	1
56	1	0
57	0	0
58	1	1
59	1	1
60	0	0
61	0	1
62	0	1
63	1	1
64	0	0
65	1	0
66	0	0
67	1	0

68	1	0
69	1	1
70	1	0
71	1	1
72	1	1
73	1	1
74	1	1
75	0	0
76	0	1
77	1	1
78	0	1
79	1	0
80	0	0
81	0	1
82	0	0
83	0	1
84	0	1
85	1	0
86	0	1
87	1	1
88	0	0
89	1	1
90	0	1
91	1	1
92	1	1
93	0	0
94	0	1
95	0	1
96	0	0
97	0	0
98	1	0
99	1	0