ECS 154A - Lab 3 SS1 2020

Logistics

Submission

Due by 20:00 on Wed, 2020-07-29. Note: This due date will be the same for lab 4, this is to allow you to plan your time. This lab has been significantly shortened for the current situation.

Turn in for the Logisim Evolution portion is on Gradescope. Submit the specified .circ files for each problem. The person submitting should specify their partner's name (if necessary) during the submission process.

Debugging Sequential Circuits

Debugging for the next two labs is going to look significantly different than the first two labs. As you will note, no vector files have been provided with this lab. This is because test vectors don't work with sequential circuits, so we can't use them to test anymore.

You will want to debug within Logisim as we have been doing for the last question from lab 2. Submitting via Gradescope will continue to grade your problem appropriately, but the autograder can't give you your output.

You can simulate debugging as is done in the autograder by creating tester circuits and expected output files. A simple tester file and output file is provided for some of the problems. This is there to give you a model of how to create your own tester files. For this lab this may not be necessary, for the final lab it will. Understanding how to test your circuits on your own is an important part of learning how to avoid mistakes during the design process.

To help you to continue debugging your circuits and learn how to do this yourself, we will provide you with two files for some of the circuits:

- tester circuits (in the *tester/* subdirectory)
- tester output files (in the *tsv/* subdirectory)

You should continue to implement your circuits in the base circuits provided. The autograder will continue to expect your submission files in the same format that you have been submitting them before.

Tester Circuits

In the *tester/* subdirectory, you will find tester circuits that will help you generate output to test your submission. To do so, you will need to load the circuit you are working on into the tester circuit. To load a circuit, follow the steps below:

1. Open the relevant tester circuit in Logisim Evolution.
2. In the menu bar of Logisim Evolution, click on *Project*.
3. From there, click on *Load Library*, then *Logisim-evolution Library...*
4. Select your submission file. It is now loaded into the tester circuit as a subcircuit.
5. In the side bar of Logisim Evolution with all the libraries, scroll down to the bottom.
6. Open the folder with your circuit name (*1*, *2*, so on), and click on the name of the subcircuit for that problem that shows up.
7. Place the subcircuit inside the tester circuit where it says "Place your subcircuit here," making sure the wires align.
    - You should not have to edit anything inside the tester circuit for it to work.
    - If all the pins don't align or there is a wire mismatch, you moved, added, removed, or otherwise changed a pin inside your submission circuit.

If you make edits to your submission circuit and want to retest it, you should unload your circuit and then reload it back into the tester circuit. To unload your circuit, follow the steps below:

1. Delete your subcircuit that you placed inside the tester circuit.
    - Logisim Evolution won't let you unload your circuit if a subcircuit is still inside the tester circuit.
2. Right-click the containing folder that you opened earlier to place the subcircuit.
3. Select *Unload Circuit* to unload it.

Once you have unloaded your circuit, reload it following the steps above. Note that you'll see a "Reload Circuit" option. In my experience, it's not very consistent in correctly reloading the circuit, so I wouldn't recommend it.

Do not modify anything inside the testing circuit! The only thing you should do is load your submission and place it in the indicated spot. Just like with the autograder, moving pins around can cause your output to change. The output of `diff` (more on that below) won't be helpful if you move anything around.

Testing Output Files

In the *tsv/* subdirectory, you will find appropriately named TSV (tab-separated values) files with the expected output of your circuit. Once you have generated an output using the testing circuit, you will `diff` your output against the correct one. See the next section for how to generate the output and how to `diff` them.

The output files look similar to the test vector files that you've been using in the past. Each row represents one set of values at a given time. As an example, the first few lines of *2.tsv* are:

```
1001 1010	0	0000 0000
1111 0111	1	1001 1010
1111 0111	0	1001 1010
1000 1111	1	1111 0111
```

Unfortunately, Logisim Evolution does not attach signal names to its output. From left to right, the signals will be in the same order as the ones listed in the description for each problem. Input values will always be to the left of output values. Values with more than 4 bits will have a space in between every 4 bits, while columns will have tabs between them. As an example, the first eight bits above correspond to *eightbitinput*, then the next bit is for *sysclock*, and finally the last eight bits correspond to *registerval*.

The nice thing about TSV files is that you can open them in a spreadsheet editor like Excel. If you make a copy of it, you can then add the signal names or any other metadata you want. This might make it easier for you to debug, but make sure not to modify the original TSV file.

Generating Your Own Output Files

Once you have loaded your circuit, it's time to generate your output file to check if your submission is correct. In your favorite terminal (I use Konsole on Kubuntu), use the following commands:

```
java -jar 'path-to-logisim-evolution'.jar 'testing-circuit'.circ -tty table > 'output-file-name'.txt
diff 'output-file-name'.txt 'tester-output'.txt
```

Replace the portions with quotes with the appropriate paths. Java needs to be in your system path if it isn't already. These commands do the following:

- run the main circuit of the testing circuit in the background
- print out the output of the pins in the circuit to `stdout`
- pipe `stdout` to your specified file
- `diff` that file with the correct output for that problem

If you're familiar with bash scripting, you can turn the above into a script to save some time. Using command line arguments for that script could let you reuse the script between circuits.

Checking for Correctness

Let's do an example. Say you wanted to test *base/1.circ* after loading it into *tester/1tester.circ* and `diff` that with *tsv/1.tsv*. The commands would look like the following:

```
java -jar logisim-evolution.jar tester/1tester.circ -tty table > test1.txt
diff test1.txt tsv/1.tsv
```

Assuming you haven't moved anything in the tester circuit, when you generate your output, only the columns for output pins should have any differences. You can use the output `diff` generates to determine where your circuit is going wrong.

If `diff` reports no changes, then congratulations! Your implementation is correct and you should get full credit when you submit to Gradescope. Make sure to submit the circuit files you've been working on and not the tester circuits; the autograder will ignore those.

Line Endings

Note that the TSV files have Unix (LF) line endings. If you are running Windows, you'll need to be careful about this, as evidenced by one post on Campuswire already. Windows (CRLF) line endings will cause `diff` to fail as the output does not match exactly. It will look like there's an error on every single line when there really isn't. This StackOverflow post contains more information about it in case you're curious.

The autograder doesn't have line ending problems, as your output is generated and then checked with `diff` in a purely Unix environment. If it doesn't pass the autograder, then there's another issue going on. If your circuit passes the autograder but is "incorrect" on every single line when you use `diff`, then it's a line ending problem. If you're running on Windows and are experiencing this issue, you have a couple of options:

- clone the repository using GitHub Desktop or another Git client that will properly handle line endings on Windows for you
- use a text editor like Sublime Text, Atom, or Notepad++ to convert the TSV files to CRLF line endings
- copy the contents of the TSV file and paste into Notepad/other text editor to resave it with CRLF line endings

PowerShell Issues

This section is only relevant if you're using PowerShell on Windows. See how Microsoft keeps coming up as a problem?

I've had significant problems with trying to use `diff` or `Compare-Object` in PowerShell. I would recommend using WinMerge on Windows instead. It's a far better `diff` tool.

The default file encoding for output files via redirects in PowerShell is UTF-16. The original TSV files are encoded in UTF-8, since that's just an extension of ASCII. If your text editor attempts to interpret the file as UTF-8, you'll get mojibake.

Thankfully, `diff` will make it pretty clear that there's an issue if the encoding is different. If you're running on Windows and are experiencing this issue, use a text editor like Sublime Text, Atom, or Notepad++ to change the encoding to UTF-8 and resave.

Constraints

For these problems, you must use designs relying on only the following, unless specified otherwise:

- AND, OR, and NOT gates
- flip flops of any type
- the Logisim Evolution wiring library

If you want to use a NOR/NAND gate, implement it via AND, OR, and NOT gates. You will get a 0 on a problem if you violate the constraints above, unless specified otherwise.

Certain problems have constraints on the number of certain types of gates or flip flops that you may use. Make sure to read those carefully. Those gates may have any number of inputs, but you are limited on how many of them you can use. This is to enforce minimization of the circuits. If you exceed the limit, you will lose a significant amount of points for that problem.

Logisim Evolution Problems [75]

1. Quick introduction to sequential circuitry [10]

- Submission file for this part: *1.circ*
- Main circuit name: *chaining*
- Input pin(s): *q* [1], *sysclock* [1]
- Output pin(s): *q1ago* [1], *q2ago* [1]

I highly recommend starting this problem early. The intent of this problem is to introduce the testing mechanisms for sequential circuits and making sure you understand those well. See the Debugging Sequential Circuits section above for more information on those.

Hook up two D flip flops in sequence; this is equivalent to a two-bit shift register. The input into the first flip flop is *q*. The output of the first flip flop and input into the second flip flop becomes *q1ago*, the value of *q* one clock cycle ago. The output of the second flip flop becomes *q2ago*, the value of *q* two clock cycles ago. This chaining/shifting will be reflected in your output file from the tester, assuming you've done it correctly.

Note that we provide a *sysclock* input to hook up to the Clock pins of your flip flops. This is required so that the tester circuit and your circuit run in lockstep. However, if you are testing inside your own circuit file, it will be more convenient for you to use a regular Clock element rather than *sysclock*.

If you do test inside your own circuit, some helpful commands are below:

- `Ctrl-T` ticks the clock once forwards.
- `Ctrl-K` repeatedly ticks the clock at a specified frequency. You can change this frequency by clicking on *Simulate* in the top menu bar, then clicking on *Tick Frequency.*
- `Ctrl-R` resets the circuit and all your flip flops.
- `Ctrl-D` duplicates a component.

You may only use two D flip flip flops to store your values. You may not use any other module from the Memory library or use more than 2 D flip flops. Doing so will result in a 0.

2. Register implementation [15]

- Submission file for this part: *2.circ*
- Main circuit name: *tregister*
- Input pin(s): *eightbitinput* [8], *sysclock* [1]
- Output pin(s): *registerval* [8]

Design a eight-bit register that uses T flip flops to store its values. This implementation differs from the one talked about in lecture, which used D flip flops. The register starts out with 0000 0000 (all zeroes) as its first value. Note that your register does not have an enable input and will thus write its input on every clock cycle.

As an exercise, I encourage you to think about what would be necessary in order to add an enable input as it's straightforward to do so.

You may only use 8 T flip flops to store your values. You may not use any other module from the Memory library or use more than 8 T flip flops. Doing so will result in a 0.

You may use XOR gates in addition to the AND, OR, and NOT gates you are normally allowed to use. There is no limit on the number of those gates you may use.

3. Parity generator [20]

- Submission file for this part: *3.circ*
- Main circuit name: *paritygen*
- Input pin(s): *inputw* [1], *sysclock* [1]
- Output pin(s): *outputq* [1]

Derive a minimal state table for a Moore model FSM that acts as a three-bit parity generator. For every three bits that are observed on *inputw* during three consecutive clock cycles, the FSM generates the parity bit *outputq = 1* if the number of 1s received in the sequence so far is odd. Thus, this is an even parity generator. Implement the FSM as a circuit in Logisim Evolution.

Note that the FSM outputs a 1 as long as the number of 1s received *so far* in the three-bit sequence is odd. This means that the circuit can output 1s before receiving all three bits. An example of this would be receiving the input 1 at the start; the circuit will output 1 after seeing the 1. Had the FSM received a 0 instead, the circuit would output 0. If the circuit receives a 0 after receiving the 1, the circuit would output 1 again. Note that after receiving the three bits, the circuit resets and starts looking at the next three bit set; thus, this is not a sliding window.

You may use a maximum of 3 flip flops for this problem. If you need to use more than 3 flip flops, your FSM is not minimized. You will lose a significant portion of credit if you have more than 3 flip flops in your circuit.

You may use a maximum of 14 AND gates and 4 OR gates with any number of inputs. You will lose a significant portion of credit if you have more, as it means your combinational logic is not minimized, or you made a mistake on code word assignment. If you violate both this and the previous constraint, you will get a 0.

4. Bit sequence recognizer [30]

- Submission file for this part: *4.circ*
- Main circuit name: *sequencecheck*
- Input pin(s): *inputx* [1], *sysclock* [1]
- Output pin(s): *outputr* [1]

Derive a minimal state table for a Mealy model FSM that acts as a sequence checker. During four consecutive clock cycles, a sequence of four values of the signal *x* is applied, forming a binary number. The oldest value of *x* would become the most significant bit in that binary number. The most recent value of *x* would become the least significant bit.

The FSM will output *outputr = 1* when it detects that the previous 4 bit sequence was either 0100 or 1010. At all other times, including when the previous sequence was not those described previously, *outputr = 0*. Implement the FSM as a circuit in Logisim Evolution.

Note that much like the last problem, this is not a sliding window. After the fourth clock pulse, the circuit resets itself and is ready to take in the next 4 bit sequence.

You will lose a significant portion of credit if your FSM is not minimized.

You will lose a significant portion of credit if your combinational logic is not minimized. If you violate both this and the previous constraint, you will get a 0.