ELE 206/COS 306 Lab 2: Combinational Logic, Modules, and Testing

Due Date: September 30th, 2020
Demonstration Required.

Tutorial Reading: Chapters 1, 2, and 7
Tutorial Review Questions: 1.6, 2.2, 2.4, 2.5, 7.1, 7.2, 7.4, 7.5, 7.6

Estimated Time: 3-5 hours

Introduction

In this lab, we will focus on the two primary ways Verilog can be used to model combinational logic. You will use both of them to build a simple 8-bit adder circuit. Additionally, you will learn about modules and Verilog's hierarchy. Finally, you will write your own test cases to verify that your adder implementation works as expected.

It is expected that you have completed the tutorial reading and tutorial review questions prior to beginning the lab. Your answers to the review questions should be a part of your write-up that is submitted at the end of the lab.

To begin, download the file *lab2.zip* from Blackboard if you have not already done so. Extract the contents of the .zip file. On Linux and Mac OS X, this can be done with a command such as

```
1 unzip lab2.zip
```

On Windows, this can be done by right clicking on the file and selecting "Extract All". This should have created a new directory named *lab2*. Navigate to this *lab2* directory in your command line.

As a final note, adders are very common and useful digital circuits. You'll cover them in great detail later in this course, but for now we'll be using the simplest of all adder designs as the basis for this lab. This design is a simple combinational circuit, and just treat it as you would for any other combinational circuit! You won't need any prior knowledge of how adder circuits work prior to starting this lab.

Part 1: Continuous Assignment – 1-bit Adder

In this section, you will be building a simple 1-bit adder circuit using the first style of combinational logic – continuous assignment. A 1-bit adder circuit has three inputs and two outputs. The inputs are the two values to add (`a` and `b`) and a carry-in value (`ci`). The outputs are the sum (`s`) and a carry-out value (`co`). The truth table for this circuit is shown below.

a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Read the truth table carefully to understand what the circuit does. Essentially, the 1-bit adder adds the three input bits. If the sum is greater than one, the carry-out bit is set. If the sum is one or three, the sum bit is set. Thus, the pair {`co, s`} forms a two-bit number that is the binary sum of the three input bits.

By algebraic reduction, we find that

$$c_o = ab + ac_i + bc_i$$

and

$$s = a \oplus b \oplus c_i$$

where $\oplus$ is the XOR operator. This leads to the circuit shown in Figure 2.1.

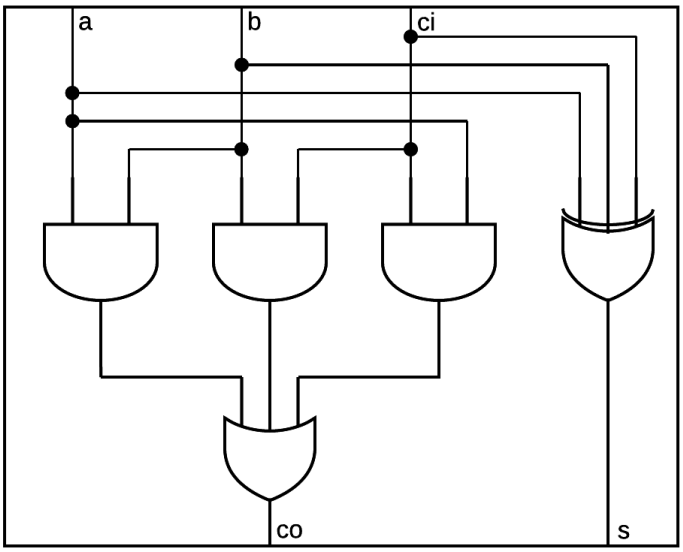

Figure 2.1: One-bit Full Adder Circuit

Your task for this section is to implement this one-bit adder circuit using only continuous assignment and `wire` types. As a reminder, Verilog's continuous assignment statement looks like this:

```
1 wire x, y, z;
2 assign x = y & z; // using assign keyword
3
4 wire w = y | z;   // alternative - assign in declaration
```

Write your code in *FullAdder.v*. The `FullAdder` module has already been declared for you, so just fill in the body to match the circuit pictured in Figure 2.1. One last note – do **not** use the addition (`+`) operator in your code in this section. We want you to implement the circuit exactly as pictured. Don't use built-in logic gate modules either – for now, just stick to bitwise and logical operators and continuous assignments.

Once your circuit has been completed, compile the `FullAdder` module. Open your terminal (OS X or Linux) or `cmd` program (Windows) and change

directories into the */lab2* directory. Then, execute the following command:

```
iverilog -g2005 -Wall -o FullAdderTest FullAdder.t.v
```

This command will compile your *FullAdder.v* and our test module, *Fulladder.t.v*. If there are any warnings or errors, fix them before proceeding.

Once compilation is complete, you can run the testing suite with the following command:

```
vvp FullAdderTest
```

The testing suite should report no errors. If you have any errors, go back and check your logic. The waveforms for the suite will be dumped as *FullAdderTest.vcd*. If you need to use them for debugging, just open them with GTKWave:

```
gtkwave FullAdderTest.vcd
```

Once the testing suite passes, you can move on to the next section.

Part 2: Modules and Hierarchy – 8-bit Adder

Now that your one-bit full adder circuit works correctly, you can chain several of them together to form an adder for multi-bit signals. The simplest form of multi-bit adder is the *carry-ripple adder*, which is simply a linear chain of 1-bit full adders. The carry-out of each full adder is connected to the carry-in of the next full adder in the chain, and as each bit in the final sum is computed, the carry values *ripple* down the chain of adders towards the final carry-out value.

Figure 2.2 shows a diagram of a four-bit carry-ripple adder.

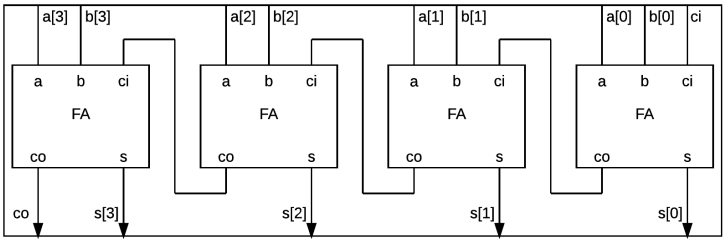

Figure 2.2: Four-bit Carry-Ripple Adder Circuit

Let's walk through how this adder works. Assume that you have two four-bit signals, `a` and `b`, whose bits are labeled from 3 (highest) to 0 (lowest). The right-most adder in Figure 2.2 sums `a[0]`, `b[0]`, and the carry-in bit. The sum is stored in `s[0]`, and the carry-out ripples over to become the carry-in of the next adder. This adder sums `a[1]`, `b[1]`, and the carry-out of the previous adder, writing the sum to `s[1]` and rippling the carry-out down the chain. At its core, this circuit works just like grade school addition – line up the digits of the operands and then work right-to-left, summing matching digits to produce a result and a carry.

Your task for this section is to create an 8-bit carry-ripple adder by instantiating eight copies of your `FullAdder` module from the previous section. You'll want to review the section on modules and the hierarchy from the Verilog tutorial before beginning.

Write your code in *Adder8Hierarchical.v*. The module prototype is provided for you – just fill in the body with your code. Note that *FullAdder.v* has already been included at the top of the file, so you can instantiate `FullAdder` modules without any further hassle.

Feel free to include as many internal wires as you need. If you'd like to use a `generate` loop to make the code simpler, feel free to read the chapter on loops in the Verilog tutorial. However, this is not required, and it is recommended to instantiate the eight full-adder modules explicitly rather than via a loop.

Once you've completed your design, compile it with the following:

```
1 iverilog -g2005 -Wall -o Adder8Test Adder8.t.v
```

This command will compile your *Adder8Hierarchical.v* and our test module, *Adder8.t.v*. If there are any errors, fix them before proceeding.

Note that this testing suite is designed to test both *Adder8Hierarchical.v* and *Adder8Procedural.v*, which you will create next. If you see any warnings regarding *Adder8Procedural.v*, ignore them for now. Run the testing suite with the following command:

```
1 vvp Adder8Test
```

There's only one test in this suite for now. You should see no failures for the hierarchical adder, although there will be a failure for the procedural adder. If you have an error for the hierarchical adder, you can open the waveforms for this test with GTKWave:

```
1 gtkwave Adder8Test.vcd
```

Once you see no errors for the hierarchical adder, you can proceed to the next section.

Part 3: Procedural Assignment – 8-bit Adder

Now that you've learned about continuous assignment and modules, we're going to create another 8-bit adder using procedural assignment. Make sure to read the Verilog tutorial's description of procedural combinational logic before proceeding.

In this section, you will build a full 8-bit adder using only `reg` datatypes and the `always @( * )` procedure. In this section, you **should** use the addition (`+`) operator. As you'll probably notice, your code should be considerably shorter. Write your code in *Adder8Procedural.v*. As usual, the module prototype is provided for you – just fill in the body.

Note that, although the code you write for this task will perform the same function as the code you wrote for your 8-bit hierarchical adder, the actual circuit that will be synthesized from those Verilog files could be radically different. The code you wrote using only bitwise operators will be translated directly down to gates as you specified. However, if you use the addition operator, a Verilog synthesizer is free to use any adder circuit it sees fit during synthesis. This could be a carry-ripple adder, but it could also be a more complex adder type.

It is good to be aware of this trade-off when designing circuits in Verilog. You can express a logical circuit in a wide variety of ways. Using higher-level operators like addition allows for much simpler code, but it gives up some fine-grained control of exactly what the final circuit will look like after synthesis.

Once your circuit is completed (make sure to use a procedural `always` block for your combinational logic), compile it and run it through our testing suite using the same commands from the hierarchical adder's section.

Now, you should see no failures for either the procedural or hierarchical adders. Once neither adder is failing, proceed to the next section.

Write-up Question 1:
What are the two different types of assignment in Verilog? Which datatypes does each style use?

Write-up Question 2:
In the last lab, we discussed two different styles of writing combinational logic in Verilog. The hierarchical 8-bit adder is best described as which of the two? How about the procedural 8-bit adder? Justify your answers.

Write-up Question 3:
In the hierarchical 8-bit adder, it can be seen that the circuit's gate-level structure is explicitly described. On the other hand, the procedural 8-bit adder does not describe much of the gate-level structure. What are the advantages and disadvantages of each implementation approach (hierarchical and procedural)? Furthermore, what are the advantages and disadvantages of using higher-level operators in Verilog, such as using arithmetic operators instead of bitwise operators?

Part 4: Writing Your Own Tests

Now that both of your 8-bit adders are completed, you will add to our test cases. Read the Verilog tutorial's chapter on testbenches before starting (you can ignore the details about clocks for now).

Once you're ready, open up our tests for the 8-bit adder, which are contained in *Adder8.t.v*. We have one basic test case already laid out (commented as "Test Case 0"). For this section, your task is to add at least five more test cases. Some may be random, but be sure to select a few carefully to test corner cases of the adders. For example, make sure to test a case where the carry-out for the adder is actually used!

For each test case, make sure to test both the hierarchical and procedural adder. Feel free to copy liberally from our example test case – the point of this exercise is to make you comfortable writing tests and coming up with your own test cases, so it's okay if you just re-use our code and change values.

Once you finish writing your test cases, re-compile *Adder8.t.v* and re-run `Adder8Test`. Ensure that both of adders pass all of your tests.

Write-up Question 4:
Describe the test cases that you chose for your adder. Explain why you chose each and how each contributes to testing your adder's correctness. It's okay if a few of your choices are random, but at least a few should have been chosen intentionally to test specific behaviors.

Write-up Question 5:
Attach a screenshot of GTKWave's waveforms for *Adder8Test.vcd*. Add the signals `ah`, `bh`, `cih`, `sh`, and `coh` from the `Adder8Test` module, and drag the cursor to hover over the provided test case (10 + 15 + 1 = 26 + 0). Explain how these waveforms show that the adder works for the given test case. Please note that GTKWave may represent multi-bit signals by default in hex, depending on the bit width – you can convert to binary or decimal by left-clicking (on some systems, use Ctrl+click or Cmd+click) on a multi-bit signal and then selecting from the "Data Format" menu. You may also find it by left-clicking the signal and then right click anywhere in the wire panel and then go to "Data Format" (it's also listed under the "Edit" menu).

Write-up Question 6:
Please leave a bit of feedback regarding this lab. How much time did it take, how difficult was it, did you get stuck anywhere? There are no wrong answers here – we'll be using the feedback to adjust this lab for the future.

Demonstration, Write-Up, and Submission

Demonstration

For your demonstration this week, visit one of the lab help sessions on Zoom and show a graduate TA your test cases for the 8-bit adder. Explain at least one test case, and demonstrate that your adders produce the correct results for that test case by using either `$display` statements in the testbench or by viewing waveform outputs in GTKWave.

Write-Up

For your write-up, please write your solutions to the tutorial review questions assigned at the beginning of the lab (1.6, 2.2, 2.4, 2.5, 7.1, 7.2, 7.4, 7.5, 7.6) and the 6 numbered Write-up questions above (15 questions in total). Please make sure your answers are concise. Feel free to create the write-up in any program of your choice, but save it as a PDF. Name your PDF *ELE206_lab2_netid.pdf*, with *netid* replaced with your Princeton NetID.

Submission

Delete *Adder8Test*, *FullAdderTest*, and any *.vcd* waveform files from the */lab2* directory. You can do this via your file browser or with one of the following commands.

On OS X and Linux:

```
rm -f *.vcd Adder8Test FullAdderTest
```

On Windows:

```
del *.vcd Adder8Test FullAdderTest
```

Then add your PDF explanation file to that directory and create a *.zip* archive of that directory. Make sure to name it as *ELE206_lab2_netid.zip*, with *netid* replaced with your Princeton NetID.

On OS X and Linux, simply change directories in your terminal to the the directory that contains the */lab2* directory and then execute this command:

```
zip -r ELE206_lab2_netid.zip ./lab2
```

On Windows, simply open up File Explorer and locate the */lab2* folder. From Command Prompt, you can open your current directory's parent in File Explorer by issuing this command:

```
explorer.exe ..
```

Right-click on the */lab2* folder and select "Send to", then "Compressed (zipped) folder". This will create a *.zip* file.

For reference, here is a checklist of files that must be in your *.zip*:

- *ELE206_lab2_netid.pdf*
- *FullAdder.v*
- *Adder8Hierarchical.v*
- *Adder8Procedural.v*
- *Adder8.t.v*

Finally, submit the following to Gradescope:

- *ELE206_lab2_netid.zip*