ELE 206/COS 306 Lab 1: Simple Combinational Logic

Due Date: September 23rd, 2020
Demonstration Required.

Tutorial Reading: Chapters 1 and 2
Tutorial Review Questions: 1.1, 1.2, 1.8, 1.11, 2.1

Estimated Time: 3-5 hours

Introduction

In this lab, you will write and debug Verilog to simulate simple combinational circuits. It is expected that you have completed the tutorial reading and 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 *lab1.zip* from Blackboard and save it as *lab1.zip* on your computer. Extract the contents of the .zip file. On Linux and Mac OS X, this can be done with a command such as

```
1 unzip lab1.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 *lab1*. Navigate to this *lab1* directory in your command line.

Part 1: Number of 1's counter

The number of 1's counter is a circuit discussed in lecture. It is also described in the textbook in Section 2.7, Example 2.25. The number of 1's in a binary

string is also called the population count. Computing the population count of a binary number is a useful operation for applications in information theory, cryptography, and data structures.

You will be implementing the number of 1's counter with a three-bit input and a two-bit output, as described in lecture and the book. You will implement the counter once in **structural Verilog** and once in **behavioral Verilog**. As discussed in the Verilog tutorial, structural and behavioral are terms used to describe levels of abstraction in Verilog. In this lab, you will begin these two models. Structural models are closely associated with continuous assignment and `wire` datatypes. Behavioral models are closely associated with always blocks (i.e. procedural assignment) and `reg` datatypes. Structural code is written with gate-level design and implementation in mind. Behavioral code, instead, focuses on the end behavior of the circuit, not necessarily on the gate implementation; in this case, the gate implementation may be left fully up to the synthesizer.

In summary, structural takes a gate-level, bottom-up approach to design, and behavioral typically takes a top-down approach without explicitly describing gate-level structure. Be careful not to confuse these two implementation approaches with types of assignment in Verilog. Continuous and procedural assignment in Verilog affects the way code is executed and does not directly impact gate-level circuit design. This will be further explored in Lab 2.

Structural Verilog

Generally, the style of writing Verilog using `wire` datatypes is known as **structural Verilog**. `wire` datatypes can only be assigned values either through continuous assignment, or by being connected to another module. The `SimpleCircuit` module contained in *SimpleCircuit.v* in Lab 0 is an example of a Verilog module written in structural Verilog.

First, you will implement the number of 1's counter using only wires and continuous assignment, i.e. structural Verilog. As a reminder, declaring a wire and assigning it a value using continuous assignment looks like this:

```
1 wire a;
2
3 assign a = b && c;
```

Open the file *Popcount_structural.v*. You will notice that the module `Popcount` has a three-bit input named `bitstring`, and a two-bit output named `popcount`. Bits with a higher index are the more significant (a.k.a. higher order) bits, so `bitstring[0]` is the one's place of `bitstring`, while `bitstring[1]` is the two's place, and `bitstring[2]` is the four's place.

Fill in the combinational logic such that each bit of `popcount` will be assigned the correct values based on the values of the bits of `bitstring`. You may refer to the discussion of the number of 1's counter from lecture and the textbook. You may declare additional `wire` datatypes if you would like. You should not use any `reg` datatypes or any `always` blocks in your solution. Additionally, instantiating other modules, such as Verilog's built-in logic gates, would still be structural Verilog, but, for this assignment, please avoid using other modules inside of your `Popcount` module. More specifically, please try to only use bitwise (`&`, |, etc.) and logical (`&&`, ||, etc.) operators.

After saving your modified version of *Popcount_structural.v*, execute the following commands to compile and run a simulation, then view the simulation results in GTKWave:

```
1 iverilog -Wall -o Popcount_structural_test Popcount_structural.v Popcount.t.v
2 vvp Popcount_structural_test
3 gtkwave Popcount_test.vcd
```

The first command compiles the code, and should complete without generating any warning messages and generate a file named *Popcount_structural_test* in the same directory. If it generates any warning or error messages, address them and run the command again before proceeding. The second step should run the simulation and generate a file named *Popcount_test.vcd*, which the third step will open in GTKWave. Note that both the compilation step and the simulation step will overwrite their output files if they already exist. Therefore, you do not need to delete these generated files before recompiling or resimulating.

In the "SST" pane, expand the `PopcountTest` module to view `circuit_under_test`, which is an instance of the `Popcount` module you just wrote. Selecting `circuit_under_test` will bring up the available signals inside the `Popcount` module in the lower portion of the "SST" pane. Add both `bitstring` and `popcount` from the SST pane to the "Signals" pane. Since these signals comprise multiple bits, GTKWave interprets these signals as binary numbers and initially displays a compact representation for the of all the bits in the "Waves" pane, rather than an actual waveform. Depending on the number of bits in a signal, GTKWave may choose to represent the signal in binary or hexadecimal by default, but you can change the representation by right clicking on a signal in the "Signals" pane and selecting from the "Data Format" menu. On some systems, you may need to hold Ctrl or Cmd and click on a signal instead of right-clicking.

Double click on both signals in the "Signals" pane to expand them to show the waveforms for the individual bits in each of these vectors. Zoom out if

necessary so you can see the values of all signals over the course of the entire simulation. Make sure the values of the outputs are correct.

Write-up Question 1:
Place the marker at a time in the simulation such that the "Signals" pane shows values when `bitstring` held a binary representation of the number 5. Leave `circuit_under_test` selected in the "SST" pane so that all the signals are visible and it displays that they are all `wire` datatypes. Take a screenshot showing the waveforms for all the bits within `bitstring` and `popcount`, as well as any other signals you feel like adding, over the course of the simulation. Do the waveforms for the individual bits convey any additional information than the values shown for the multi-bit signal if it were not expanded?

Behavioral Verilog

The style of Verilog that uses `reg` datatypes and procedural assignment and its associated conditional structures is generally referred to as **behavioral Verilog**. In the case of combinational circuits, procedural assignment can be advantageous when building more complex circuits. A procedure allows for the use of powerful programming constructs such as if-then-else and case statements; this can be very useful to a programmer trying to simulate a complex circuit or trying to implement a circuit without much regard to gate-level implementation.

Next, you will implement the same number of 1's counter functionality using `reg` datatypes and procedural assignment. Although you will be using the `reg` datatype, it is important to note that these registers will be behave and be implemented as wires when compiled. As a reminder, declaring a `reg` and assigning it a value using procedural assignment looks like this:

```
1 reg a;
2
3 always @( * ) begin
4   a = b && c;
5 end
```

Open the file *Popcount_behavioral.v*. You will notice that the output `popcount` has already been declared to be a `reg`, so that you may assign values to it procedurally. You may declare and use additional `reg`s if they help. You may not declare any `wire`s,

Fill in the combinational logic such that each bit of `popcount` will be assigned the correct values based on the values of the bits of `bitstring`. You

may declare additional `reg`s if desired. You should not declare any new `wire`s or write any `assign` statements in your solution. So, the only wire in this module should be the input `bitstring`, which is implicitly a `wire`. Similar to the structural implementation, please avoid using other modules; try to only use bitwise (`&`, |, etc.) and logical (`&&`, ||, etc.) operators.

After saving your completed version of *Popcount_behavioral.v*, execute the following commands to compile and run a simulation, then view the simulation results in GTKWave:

```
1 iverilog -Wall -o Popcount_behavioral_test Popcount_behavioral.v Popcount.t.v
2 vvp Popcount_behavioral_test
3 gtkwave Popcount_test.vcd
```

The first command should complete without generating any warning messages and generate a file named *Popcount_behavioral_test*. If it generates any warning or error messages, address them and run the command again before proceeding. The second step should run the simulation and generate a file named *Popcount_test.vcd*, which the third step will open in GTKWave.

In the "SST" pane, expand the `PopcountTest` module to view `circuit_under_test`, which is an instance of the `Popcount` module you just wrote. Selecting `circuit_under_test` will bring up the available signals inside the `Popcount` module in the lower portion of the "SST" pane. Add both `bitstring` and `popcount` from the SST pane to the "Signals" pane. Double click on both of these in the "Signals" pane to expand them to show the waveforms for the individual bits in each of these vectors. Zoom out if necessary so you can see the values of all signals over the course of the entire simulation. Make sure that all values are correct, and the same as in your simulation of your structural Verilog implementation.

Write-up Question 2:
Place the marker at a time in the simulation such that the "Signals" pane shows values when `bitstring` held a binary representation of the number 2. Leave `circuit_under_test` selected in the "SST" pane so that all the signals are visible and that it displays that they are all `reg`s, except for `bitstring`. Take a screenshot showing the values of the vectors `bitstring` and `popcount`, as well as any other signals you feel like adding, over the course of the simulation. How do the values shown in the "Waves" pane for the multi-bit vectors show that the circuit is computing the correct value at all points in time?

Write-up Question 3:
Why might you prefer to write this module in structural or behavioral Verilog? What are the benefits of each choice, if any?

Part 2: Buggy Decoder

Decoders are one class of circuit components discussed in lecture. They are also discussed in Section 2.9 of the textbook, beginning on page 84.

Open the file *Decoder3x8.v*. This contains a module `Decoder3x8` within which someone attempted to write a behavioral Verilog implementation of a 3-to-8 decoder. There are syntax errors that will prevent the code from compiling. Attempt to compile the buggy code with the command

```
1 iverilog -Wall -o Decoder3x8_test Decoder3x8.v Decoder3x8.t.v
```

First, find and fix only the syntax errors you find, making minimal changes to get the code to compile successfully, with no error or warning messages. Note that every warning or error message the compiler generates will usually include an approximate line number.

Write-up Question 4:
Describe the warning or error messages the Icarus Verilog compiler generated. What were the errors that caused those messages?

Besides the syntactic errors, there are logic errors that cause the code to specify the incorrect behavior. Nonetheless, now that code compiles, we can run the simulation using the following commands:

```
1 vvp Decoder3x8_test
2 gtkwave Decoder3x8_test.vcd
```

The simulation tests the decoder by testing all possible inputs twice. If you fixed the syntatic errors without fixing the logical errors, the resulting waveforms in GTKWave should look like this:

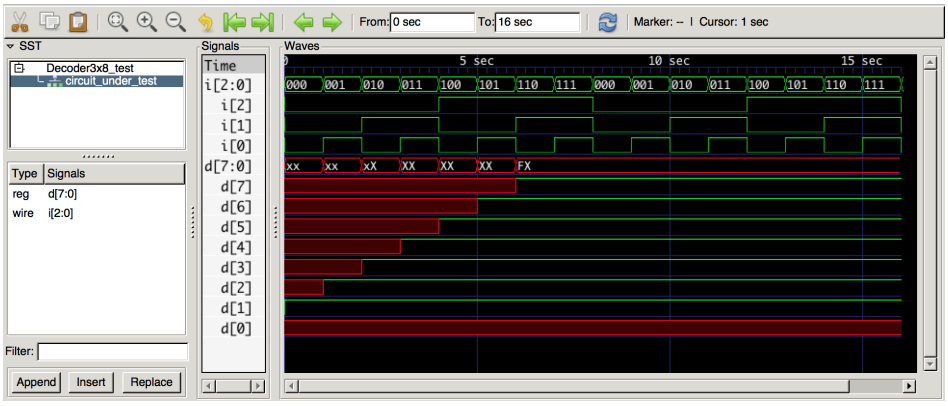

Figure 1.1: Result of simulating decoder with logic errors.

Write-up Question 5:
For a combinational circuit, it is possible to write a Boolean formula for each output bit that specifies its value in terms of the input. Does the buggy code, as shown in Figure 1.1, successfully implement the decoder combinational circuit? If so, support your answer by writing a Boolean formula for each output bit. If not, justify your claim using the behavior shown in GTKWave.

One of the logical errors is that the procedural assignments in the `always` block does not fully specify the values of all outputs in all cases. As discussed in the tutorial, this causes **implicit latching**, a problem that can be solved by adding a single default assignment at the start of the `always` block's body. Fix all the logical errors so that the decoder functions correctly according to the behavior for decoders described in lecture and in the textbook. Once finished, save your fixed version of *Decoder3x8.v*, recompile and simulate the circuit, and verify that the behavior is correct in GTKWave.

```
1 iverilog -Wall -o Decoder3x8_test Decoder3x8.v Decoder3x8.t.v
2 vvp Decoder3x8_test
3 gtkwave Decoder3x8_test.vcd
```

Write-up Question 6:
In GTKWave, place the marker at a time in the simulation such that the "Signals" pane shows values when `i` held a binary representation of the number 6. Take a screenshot showing the waveforms for all the bits within `i` and `d`, as well as any other signals you feel like adding, over the course of the simulation.

This decoder could also be written using structural Verilog. Open the file *Decoder3x8_structural.v*. In this file, the output `d` should not be declared as a `reg`. Write the logic of a 3-to-8 decoder using structural Verilog, i.e. using only wires and continuous assignment. You can test your logic using the same testbench as the behavioral Verilog `Decoder3x8` implementation.

```
iverilog -Wall -o Decoder3x8_test Decoder3x8_structural.v Decoder3x8.t.v
vvp Decoder3x8_test
gtkwave Decoder3x8_test.vcd
```

Write-up Question 7:
Why might you prefer to write this module in structural or behavioral Verilog? What are the benefits of each choice, if any?

Write-up Question 8:
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 and show a graduate TA your code for the structural and behavioral implementation of `Decoder3x8`. Demonstrate that your structural 3-to-8 decoder behaves correctly by discussing the waveforms in GTKWave.

Write-Up

For your write-up, please write your solutions to the tutorial review questions assigned at the beginning of the lab (questions: 1.1, 1.2, 1.8, 1.11, 2.1) and the 8 numbered Write-up questions above (13 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_lab1_netid.pdf*, with *netid* replaced with your Princeton NetID.

Submission

Delete *Popcount_behavioral_test*, *Popcount_structural_test*, *Decoder3x8_test*, and any *.vcd* waveform files from the */lab1* directory. You can do this via your file browser or with one of the following commands.

On OS X and Linux:

```
rm -f *.vcd Popcount_behavioral_test Popcount_structural_test Decoder3x8_test
```

On Windows:

```
del *.vcd Popcount_behavioral_test Popcount_structural_test Decoder3x8_test
```

Then add your PDF explanation file to that directory and create a *.zip* archive of that directory. Make sure to name it as *ELE206_lab1_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 */lab1* directory and then execute this command:

```
zip -r ELE206_lab1_netid.zip ./lab1
```

On Windows, simply open up File Explorer and locate the */lab1* 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 */lab1* 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 *ELE206_lab1_netid.zip*:

- *ELE206_lab1_netid.pdf*
- *Popcount_structural.v*
- *Popcount_behavioral.v*
- *Decoder3x8.v*
- *Decoder3x8_structural.v*

Finally, submit the following to Gradescope:

- *ELE206_lab1_netid.zip*