Lab 1: Simple Combinational Logic

Daniel Simone

29 September 2021

1 Introduction

1.1 Class Information

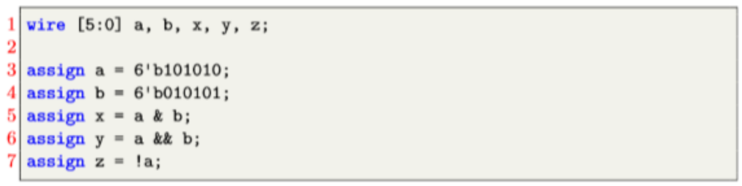
Course: ECE 206
Demo Lab section: B02
Lecture Instructor: Professor Sharad Malik

2 Write-Up

2.1 Tutorial Review Questions

2.1.1 Question 1.6

Read and understand the following Verilog code:



```
wire [5:0] a, b, x, y, z;

assign a = 6'b101010;
assign b = 6'b010101;
assign x = a & b;
assign y = a && b;
assign z = !a;
```

a *Will the values for x and y be the same? Why?*

The values for X and Y are different: X returns the binary number 6'b000000, while Y returns the binary number 1'b0. These numbers are both zero, but have different amounts of bits in them.

b *What type of operator is &? What type of operator is &&? What is the difference between the two types of operators?*

X uses the bitwise AND (&), which compares each bit in the same place (1s, 2s, 4s, 8s, etc. place) in the two numbers and returns the AND operation of each comparison as a binary number. Y uses the logical AND (&&), which compares the entire numbers together as whole units, and returns a single bit: 0 or 1.

c *Assume the programmer was expecting z to equal 6'b010101 using the code above. What is the value of z? Is this the desired result? If not, how could you fix it without using b?*

The value of Z is 0. This is not the desired result. It can be fixed by replaced the "!" (logical NOT operator) with "~" (bitwise NOT operator).

2.1.2 Question 2.2

What is a port? What is the difference between a module input and output port?

A port is a connection used to send data to and receive data from the module. An input port is a wire within the module that is read-only, and receives data from outside the module. An output port is also a wire by default, but can be a register if specified explicitly, and sends data to sources outside the module.

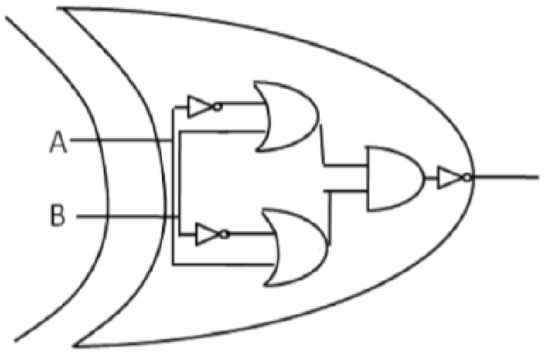
2.1.3 Question 2.4

How do you reference another file in Verilog? Is instantiating a module from another file different from instantiating a module from within the same file?

To reference another file, which contains a module, in Verilog, you can use the "‘include" statement with the file's path. Then, you can instantiate the module by stating the name of the module and the name of the instance, as well as attaching input and output ports to wires in the original file, using the dot operator to name the port. This is the same process as instantiating a module from within the same file.

2.1.4 Question 2.5

The following is an implementation of an XOR gate:



a *How could a programmer abstract this design into modules?*

A programmer could make each component gate (NOT, OR, and AND) into modules, and then make the entire XOR gate its own module.

b *What is a top-level module in Verilog? What are they usually used for?*

A top-level module is a module with no ports that cannot be instantiated, and only serves as the starting point for the full description of a circuit. It is the root of the hierarchy of child-parent module relationships. These modules serve as bases for testing the circuit by simulation.

2.1.5 Question 7.1

For software developers, testing is an important part of the development process, but even if bugs are later found in the software, developers are able to release an update or patch. Is this the same case for hardware developers? How could not rigorously testing a Verilog circuit design be financially costly, especially for hardware developers like Intel?

While it is possible to re-print an engineering sample of a circuit during the testing phase, it takes significantly more time and cost than simply pushing an update to a GitHub repository. The situation is even more dire if you or your company has started to ship the faulty circuits to resellers or consumers, and has to issue a recall to scrap all of the circuits, then replace them with correct ones. This would take months of lost time when you could have been selling inventory, and great quantities of money would be lost on printing the faulty circuits (as well as on the months of time that you could have been earning money). Furthermore, disposing of the faulty circuits would also be timely and expensive.

2.1.6 Question 7.2

Read and understand the following Verilog code:

```
`timescale 10ns/10ps
```

a *How do you specify a delay in Verilog?*

A time delay is specified using hte ”‘timescale” directive. After the directive, two numbers, separated by a forward-slash, are written: the time scale and precision. Both must be either 1, 10, or 100 and include units (like ns for nanoseconds).

b *What is the timescale directive used for? In the above example, what is the time scale and what is the precision?*

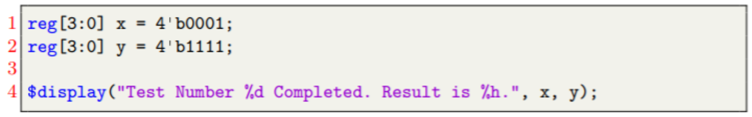
The timescale directive is used for adding delays into circuits. For instance, with the above code, delays will be in units of 10 nanoseconds (this is the time scale of the directive), with a smallest unit of 0.001 nanoseconds, which is 10 picoseconds (this is the precision of the directive).

c *Why are delays useful in Verilog testbenches?*

Delays are useful for adding time between setting the inputs and checking the outputs of a circuit, since the results of processing the inputs may take some time to propagate throughout the circuit. Delays may also be used to add a time gap between separate tests.

2.1.7 Question 7.4

Read and understand the following Verilog code:



```
reg[3:0] x = 4'b0001;
reg[3:0] y = 4'b1111;

$display("Test Number %d Completed. Result is %h.", x, y);
```

a *What would be the final text printed to the terminal?*

The final printed text should be: “Test number 1 Completed. Result is F.”

b *How would you change the code to print the test number as an octal value?*

To print the test number as an octal value, line 4 can be changed to:

```
$display("Test Number %o Completed. Result is %h.", x, y)
    ;
```

Listing 1: Test Number in Octal Digits

2.1.8 Question 7.5

How do you properly end a simulation? How does the simulation behave if you fail to add this directive?

To end a simulation, one should type "$finish" just before the "end" line of the "initial begin" statement which is commonly used in testbenches. Otherwise, the simulation will not exit, and the code will have to be stopped manually.

2.1.9 Question 7.6

Why should you use !== and === when checking equality in testbenches?

The "!==" and "===" operators are useful in testbenches because, unlike the logical comparison operators, they will check for a bitwise exact match, including "X" and "Z" values. The logical operators may wrongly pass conditions when the variable being compared evaluates to "X" or "Z" since they return "Z" if the relation they are testing is ambiguous, but these special operators will not.

2.2 Write-Up Questions

2.2.1 Question 1

What are the two different types of assignment in Verilog? Which datatypes does each style use?

The two different types of assignment in Verilog are continuous and procedural assignment. Continuous assignment uses the wire datatype, while procedural assignment uses the register datatype.

2.2.2 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.

The hierarchical 8-bit adder is best described as structural Verilog, while the procedural 8-bit adder is best described as behavioral Verilog. The hierarchical adder code is written very similar to how the hardware circuit would be built, which is seen through the use of wires, specifying exactly what kind of adder it is, and coding the exact gates to be used in the circuit, which is in line with structural Verilog. On the other hand, the procedural adder uses registers and the arithmetic addition operator, which is more high-level coding and not directly synthesizable into hardware - the synthesizer may translate the code to more complex adder circuits than a carry-ripple adder - which is more in line with behavioral Verilog.

2.2.3 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?

The advantage of the hierarchical approach is direct control over the exact circuit structure used in the final hardware implementation, down to the wires and gates used. This allows the programmer to predict time delays and optimize the circuit to be as fast or power-efficient as the specific application requires. However, the procedural approach may be preferred in the interest of saving time in coding and producing more human-readable code. The downside of this approach is that higher-level operators common to procedural code, like the arithmetic addition operator, are not directly synthesizable into hardware - the exact implementation is up to the synthesizer, which may translate addition to more complex adder circuits than expected.

2.2.4 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.

The five test cases I chose for my adder were the following:
Format: a, b, ci →s, co

1. 0, 0, 0 →0, 0

   This test case tests the edge case that all values are zero. It would find whether the adder is adding any random values at any point.

2. 10, 0, 0 →10, 0

   This test case tests the edge case that only one value (a) is inputted. It would find whether the adder's manipulations on "a" morphed it to a different value.

3. 0, 10, 0 →10, 0

   This test case tests the edge case that only one value (b) is inputted. It would find whether the adder's manipulations on "b" morphed it to a different value.

4. 255, 1, 0 →0, 1

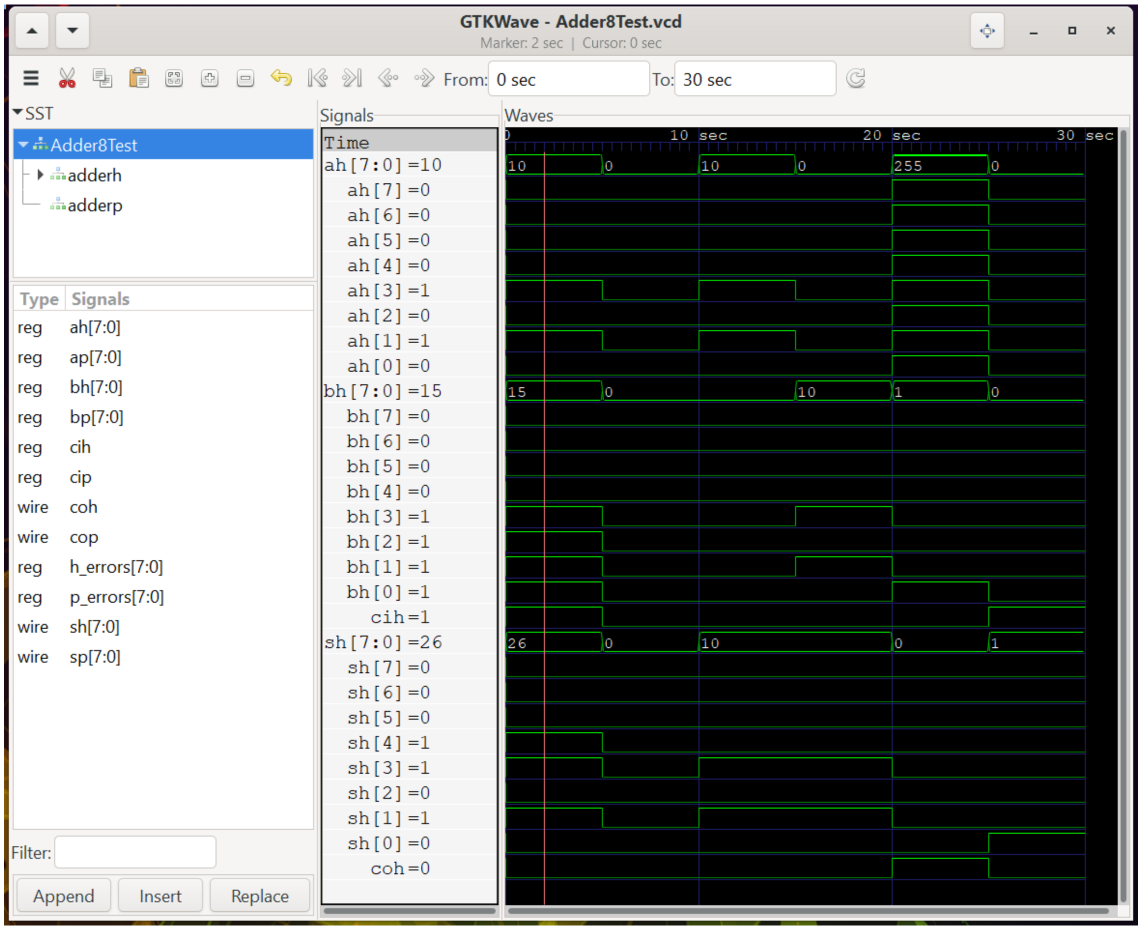
   This test case tests the edge case that the result of adding is greater than 8 bits. It would find whether the carry-out value is evaluated properly, signifying that the result of addition was over 8 bits.

5. 0, 0, 1 →1, 0

   This test case tests the edge case that only one value (ci) is inputted. It would find whether the adder's manipulations on "ci" morphed it to a different value.

2.2.5 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).



Type	Signals
reg	ah[7:0]
reg	ap[7:0]
reg	bh[7:0]
reg	bp[7:0]
reg	cih
reg	cip
wire	coh
wire	cop
reg	h_errors[7:0]
reg	p_errors[7:0]
wire	sh[7:0]
wire	sp[7:0]

```
Time
ah[7:0]=10
  ah[7]=0
  ah[6]=0
  ah[5]=0
  ah[4]=0
  ah[3]=1
  ah[2]=0
  ah[1]=1
  ah[0]=0
bh[7:0]=15
  bh[7]=0
  bh[6]=0
  bh[5]=0
  bh[4]=0
  bh[3]=1
  bh[2]=1
  bh[1]=1
  bh[0]=1
     cih=1
sh[7:0]=26
  sh[7]=0
  sh[6]=0
  sh[5]=0
  sh[4]=1
  sh[3]=1
  sh[2]=0
  sh[1]=1
  sh[0]=0
     coh=0
```

Figure 2.1: Test Case Waveforms

These waveforms show that the adder works for the given test case since the circuit returns the correct value (10 + 15 + 1 = 26). This is visible in the signals side panel, where it is clear that, for the hierarchical circuit, the inputs are:

- a = 10,
- b = 15,
- ci = 1

whereas the outputs are:

- s = 26,
- co = 0

which is correct. This is also visible when the signals are expanded, such that each individual input and output signal that makes up a, b, and s is visible in binary. Here, the inputs are:

- a = 00001010,
- b = 00001111,
- ci = 1

whereas the outputs are:

- s = 00011010,
- co = 0

which is also correct. This gives us two views to confirming that the calculations are correct, down to the bit level.

2.2.6 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.

This lab took me about 6 hours. It was not very difficult, but I did get stuck somewhat on the procedural adder, since I was not sure how to use the arithmetic addition operator - I thought that perhaps there was some kind of syntax to it. Also, assigning the carry-out bit to the correct value gave me some trouble, since two registers were involved in representing the same number.

This paper represents my own work in accordance with University regulations.

Daniel Simone