Testing Combinational Implementations

Ben Heard

February 21, 2021

Abstract

To date we've been using the labs to provide detail on how to setup and use the tools as well as to introduce a little Verilog. Please refer back to the previous labs for more details on using the tools. In this lab we'll talk about how to use Verilog to test an implementation. You have already seen testbenches but they have been created by the instructor. In this lab you will create a pair of testbenches for designs that have been provided.

1 Getting the Lab Content

There is a bit of content that I've placed in your repositories. Note, however, that "in your repository" means that I've put it in GitHub. You won't automatically see the new content in your directory (where you cloned the repository for lab 1). You'll need to tell GitHub that you want the new materials.

In the first lab you cloned the repository. I recommended placing it in an ece310 folder and you may or may not have done that. Change into the directory where you cloned your repository and then change into the repository itself. For me that looks like the following. (Note that the $ represents the prompt and is not part of the command that I typed in).

```
$ cd ece310/my_unity_id
```

Now that you are in your repository you can look at the status and the log. It should reflect what you saw that last time you worked with your repository and, in this case, should show that there is nothing to commit and that your last commit was likely something for lab 3.

```
$ git status
$ git log
```

To get the new material from GitHub you need to interact with GitHub and tell it that you want to **pull** any updates that have been made to the repository. This is done with the following pull command.

```
$ git pull
```

1.1 What should have shown up

1. A directory hierarchy containing labs/lab_004
2. A README.md file in that directory (inspect its contents)
3. This PDF file (in a GUI session you could launch acroread)
4. A file called modelsim.ini
5. Some Verilog files down in labs/lab_004
   (a) fa.v
   (b) fa3bit.v

2 Creating a Testbench

2.1 What is a Testbench?

While the implementation of a design is important, an implementation without proof that it behaves as expected doesn't have much value. This is where a verification effort comes in. In many cases more time and resources are expended verifying a design than are spent implementing it. As design complexity increases, so too does the complexity of proving that the design works.

There are many methods of verification but we'll focus on the most basic form, a directed exhaustive testbench. For combinational circuits (those without any sequential elements) a directed exhaustive testbench is one that, over time, stimulates the design will all possible input combinations of 1's and

0's. For the moment we won't worry about the four-value logic that Verilog variables represent, specifically 0, 1, x (unknown), and z (high impedance).

The total number of input vectors needed to exhaustively stimulate a combinational design is 2^n where n is the number of bits in the input. So, for example, if a design accepted 2 bits of input then the the total number of vectors needed to exhaustively test the design is $2^2 = 4$, specifically 2'b00, 2'b01, 2'b10, and 2'b11.

A testbench will instantiate a device (or unit) under test, DUT (UUT), and stimulate it over time with the complete set of input vectors. As the design is stimulated it will produce output that can be compared against the expected output for a design to ensure that the implementation behaves correctly.

2.2 Numeric Representation

As an aside, the representation of an input vector used in the previous paragraph is <number of bits>'<base><value in that base>. E.g. 2'b00 is a 2 bit vector in the binary base (b) with the values 00. Similarly, 4'h3 is a 4 bit vector in base 16 (hexadecimal) with a value of 3. These are all unsigned quantities so any most significant bits that are not represented are interpreted as zero to Verilog. E.g. 16'b10110 is a 16 bit vector in binary with the value $10110_b = 22$. The most significant bits are presumed 0. This is just a shortened way to write 16'b0000000000010110.

Finally, to make it easier to read a long string of numbers, you may insert an underscore at any point in the value. E.g. 16'b0000_0000_0001_0110. With the information in these paragraphs, the follow are equivalent.

```
16'b0000000000010110
16'h0016
16'd22
16'o000026
```

2.3 Verilog Testbench Module

A testbench module in Verilog is similar to any other design module in Verilog. There are a header and terminator, instances and net declarations, but no ports. There are no ports because the testbench doesn't have input or output itself; instead, it internally generates input to the DUT and inspects output from it.

```
module <testbench name>;

  <net/variable declarations>

  <module to test> DUT (
    <connected ports>
  );

  initial begin
    <stimulus>
  end

endmodule
```

2.3.1 Lab 3 Testbench Example

The following listing is from lab 3. Recall that you were to design a small circuit with inputs G, and D, and an output F. This circuit with 2 bits of input requires 4 vectors to fully stimulate the design.

```
module lab_003_partb_tb;

  reg G, D;
  wire F;

  lab_003_partb DUT (
    G, D, F
  );

  initial begin
    $display( $time, ": G D | F" );
    $display( $time, ": ----+--" );
    $monitor( $time, ": %b %b | %b", G, D, F );
  end
  initial begin
         { G, D } = 2'b00;
    #10 { G, D } = 2'b01;
    #10 { G, D } = 2'b10;
    #10 { G, D } = 2'b11;
    #10 $stop();
  end

endmodule
```

The header should be understood. The only difference is that it doesn't have a port list. Generally, you'll see me include _tb at the end of the module name just to visually represent it as a testbench. This is not necessary.

The variable declarations include both wire and reg data types. The most important thing to note about the reg data type is that it is not a register, not a flip flop, just by declaration. It is a data type that holds a value during the operation of a circuit but that doesn't always mean a flip flop. A wire is something you're already familiar with; it acts like a wire in a circuit and carries values from one point to another.

The second point to remember about reg data types is that they are required if you are assigning to a variable within a procedural block. You may not declare these are wires; you will receive an error when compiling. For this style of testbench where you are assigning input stimulus within the initial procedural block, those variables must be declared as type reg.

We'll talk about procedural modeling in future labs. For now just know that the initial block is a special type of procedural block. Within a procedural block statements are evaluated top to bottom. That doesn't mean, necessarily, that the execute one after the other. In the case of the first initial block, the display statements do evaluate in order; they work like printf to display strings during the simulation.

The initial block only evaluates once so the display statements are only evaluated a single time, printing out the heading of a table. The monitor statement is similar. However, it will evaluate (and print) any time that any of its arguments change. Using %b formats the output as binary. Since the values for G, D, and F are only single bits, the monitor prints out a continuous table any time any of those bits change.

The second initial block evaluates beginning at time 0 for the simulation, just as the first initial block does. Similarly, its statements are evaluated top to bottom. In this case, however, there are blocking assignments, assignments made with a single equals sign (=). This blocking assigment tells the simulation to evaluate the expression and perform the assignment before moving on to evaluate the next statement.

Also, there are delay statements (the #10) that tell the simulation to wait 10 units of simulation time before evaluating the next expression. So, at the beginning of simulation the values for G and D are set to 0, each. The blocking assignment informs the simulator to make the assignment before moving to the next statement. The next statement is a delay of 10 units of time. The simulator allows 10 units of time to elapse before evaluating the next statement. That next statement is another assignment to G and D. The process continues until the simulator reaches the $stop() statement.

As a final point, the assignment statements leverage the Verilog concatenation construct, {}. Think about each variable as a number of bits. When they are concatenated, the result represent the combined variables. So, if you concatenate 2'b00 and 2'b01 you get a value that is 4'b0001. This is done as {2'b00, 2'b01}. The concatenation may also be used with variables so {G,D} is the concatenation of the variables G and D. When used on the left hand side (LHS) of an assignment the value on the right hand side (RHS) is distributed across the concatenation according to the variable sizes.

For example, take the following.

```
wire [3:0] M;
wire [1:0] N;

assign {M, N} = 8'b0100_1101;
```

After the assignment, M will have the value 4'b0011 and N will have the value 2'b01. The two most significant bits (MSb) will be lost since there is not enough space in the concatenation to hold them. Returning back to the example, the concatenation is how we're able to effectively count on the RHS making assignments through every possible state.

3 Lab Assignment

3.1 Full Adder Testbench

Produce a testbench for the full adder provided in your repository. The full adder is in the file fa.v. Ensure that your testbench stimulates the DUT with an exhaustive set of inputs.

Display Hints The following are a couple of hints about displaying your output.

1. Use the $time directive in your display and monitor statements to see the time units when the prints occur. This will help you better understand what is going on throughout the simulation.
2. There are 3 inputs bit; A, B, and Cin. There are two outputs, Sum and Cout. Each one is a single bit so using the %b format is sufficient. For example, "%b %b %b | %b %b" would make a good table row.
3. Similarly, the three inputs may be assigned as a concatenation like {A, B, Cin}.

3.2 3 Bit Full Adder Testbench

Produce a testbench for the 3-bit full adder provided in your repository. The 3-bit full adder is in the file fa3bit.v. Open the implementation and take note that there is no Cin to the 3-bit adder. Instead, it has been tied to zero meaning that this would be used as the least significant adder if more than one were chained together. As a result there are only 6 bits of input; the vectors for A and B are 3 bits each.

Ensure that your testbench stimulates the DUT with every combination of input to exhaustively test the circuit.

Concatenation Reminder Recall that you can concatenate variables that are wider than a single bit. Follow the guidance from earlier in the lab manual to set up your assignment statements.

Display Hints Now that the inputs and outputs are more than single bits the display can become a little tricky. My recommendation is to represent the values in the monitor as decimal and control the spacing within the format specifier. For example.

```
$monitor( $time, ": %1d + %1d = %2d", A, B, F );
```

The %1d formats the output in that place as a single digit decimal value. Since the inputs are 3 bits each they are only able to take the values 0-7 so a single decimal value is sufficient. The concatenation of Cout and Sum makes is handled in the module itself, providing only the 4 bit output. The result as a 4 bit quantity that could take the values 0-15 but will only ever see 0-14 since the maximum addition will be 7+7. Then, the %2d will format the decimal output padded to 2 digits.

4 What to Submit ... and How

You will need to submit both of the testbenches that you created.

```
$ git status
```

The files that you created for the testbenches should appear as untracked files. This means that, while you've created them, they are not yet part of your Git database. You need to add them to the repository.

```
$ git add <file for fa testbench>
$ git add <file for fa3bit testbench>
$ git status
```

Your status should now reflect that the files have been added and are called "new" files. Commit them to the repository.

```
$ git commit -m "Your comment here"
```

Note that there will be other files that are untracked. For example, the work library and a transcript file from your testbench runs. These are generated and temporary files and don't need to be part of you repository. The modelsim.ini file may have also changed. It's not necessary that it be committed back to the repository but you are welcome to do so.

Now that you've committed your changes you can push those changes to GitHub.

```
$ git push
```

Now that it's been committed and pushed to GitHub you may record the commit SHA in the lab 3 quiz. Remember that you may get the SHA value from either the log or the rev-parse command.

```
$ git log -1
$ git rev-parse HEAD
```

As a last step, I recommend that you remove any files that are untracked. This just keeps your working area clean and reduces clutter that you'll see in any git status. I find that the easiest way to remove untracked files is to issue the command git clean. If you do this with the recursive options then it will remove untracked files from where you are and down the directory hierarchy. You can be at the top of the repository and issue it and it will remove all untracked files.

```
$ git clean -fd
```