Lab 1: Using Testbenches More Effectively

Assigned: Monday 9/9 Due **Monday 9/23** (midnight)

Instructor: James E. Stine, Jr., Rose Thompson

1. Introduction

In Lab 0, you learned how to go through the procedure of designing logic and the methodology in creating digital logic. That methodology is no different than what advanced engineers use at Apple and AMD to create designs. However, one of the differences is that experienced engineers make good and ample use of testbenches.

To ramp our understanding of testbenches, this lab is about using features of the testbenches to get used the process of testbenches. It is also about learning how testbenches work and using them to simulate designs. Fortunately, this laboratory is more about using the simulation software than using the FPGA. But, I want to emphasize that testbenches are the key to making your digital design work correctly. They will also be key to getting your future labs working. Without good testbenches, your implementations on your FPGA will never work. Therefore, I want to emphasize that this laboratory is vitally important for your future labs and implementations and, more importantly, your career.

By the end of this laboratory, you should be familiar with testbenches, how to create them, and what goes into them to create good digital simulation. The laboratory is really meant to be an important process for future labs - therefore, please try to understand everything about testbenches in this lab and how to use them.

1.1 Testbench

Again, simulation is vital to making sure your hardware for any architecture or digital system works. It is often the difference between something that works and a bad product. Therefore, it is vital that for lab that you understand well how to construct a testbench and use it to debug a more complicated design. Engineers will typically spend days verifying their designs through testbenches in hopes they catch all possible issues and anomalies. Perhaps, you have owned a digital device that has behaved oddly compared to what the owner's manual has indicated. This occurred because an engineer did not completely verify its specific operation. Again, we plan on using ModelSim to help us this laboratory.

Although we will not discuss much of the theory here until later in the semester, we will be using a clock in this laboratory. A clock is a synchronization signal usually created by an external device that is accurate and provides an accurate synchronization of data throughout your digital design. In fact, as we will see later in the semester, its one of the more important elements for a digital device. Your computer would be lost without the ability to keep signals synchronized and the clock provides this as well as the ability to move large amounts of data through your digital logic.

One of the things that gets a little confusing is that SystemVerilog [1], in general, is supposed to be for designing hardware. However, they are also used many times for flushing out topological designs that are more behavioral. This means you need some good way of understanding how to get bits back and forth between systems. This is really the job of the testbench.

Although the book has a great chapter on this area (Chapter 4 of our textbook), it is kind of difficult to follow when it comes to testbenches. This is because testbenches are typically best understood by using it and not reading about them. So, we will try to understand more about testbenches and how they work in this laboratory. What is a testbench?

Testbenches are essential to HDL designs and they are not unique to simulators. They are part of the IEEE SystemVerilog standard [1] and are typically written in a behavioral manner. Many testbench designs exist but using a testbench that self-verifies the result is the best approach to create a successful design. Therefore, it is encouraged that you utilize a self-checking style or approach within your testbenches.

Testbenches are written to be run with your simulation. The simulation environment runs a Tool Control Language (Tcl) file called a `DO` file that is basically a batch file for ModelSim that allows the simulation to run regardless of a users set up. A sample `DO` file is given to you, but you should modify to make sure it runs

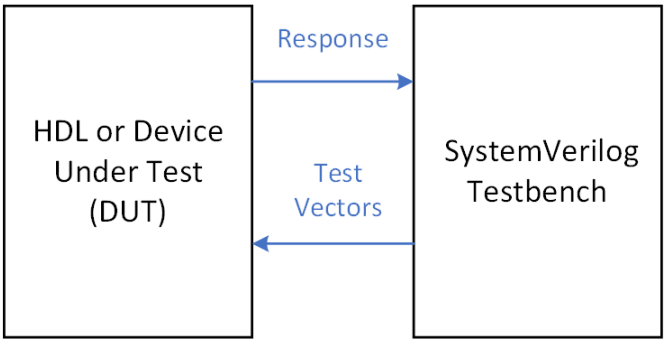

Figure 1: The TestBench Structure

your design and its appropriate testbench. To run ModelSim with a `DO` file, type the following command at a command prompt.

```
vsim -do file.do
```

It is encouraged to consult your TA or the Internet to learn how to get to the terminal in Microsoft Windows, so you can run your DO file correctly.

1.2 Basics of Testbench

The testbench is another SystemVerilog file that sits above, in hierarchy, your Verilog design that you wish to test. The testbench is written behaviorally and is **not** synthesized! It is only used to test your design and see if it works. More importantly, it is typically utilized with good known vectors that comprise a good set of vectors that will assure your design will work. Normally, large digital designs are too difficult to test exhaustively. Therefore, good vectors that are symbolic of what would normally be used in a real application are used to test whether your design will work.

These test vectors are either generated with another program, a data set of real-life applications, or generated within the testbench. For this laboratory, you will use vectors generated through the testbench to test your design. These *known* test vectors are typically called **golden vectors** as they represent the correct response for a given input. For many applications, specifications written by the Institute for Electronic and Electronics Engineers (IEEE) or other specification/standard committees are used for generating golden vectors. However, for this laboratory we will generate the golden vectors through our testbench.

Again, a test bench comprises a top-level SystemVerilog (SV) file that instantiates the SV file one wishes to test. This instantiated design is called the device under test (DUT) or circuit under test (CUT). This can be represented visually as shown in Figure 1

The basic structure of a testbench involves three distinct regions:

1. Internal declaration of inputs and outputs to the DUT as well as variables used for testing
2. Instantiating the DUT
3. Generating Test Vectors and Producing Output

To create a testbench it is easiest to instantiate the DUT or SV file you wish to test as a basis for your testbench. I typically copy over the SV inputs, outputs, and module name I wish to test over to my testbench to create these three parts to avoid misspellings and other issues.

The test vectors can be generated by an outside program, however, they can be generated internally within testbenches. All good testbenches will have some mechanism to help users test a design and demonstrate that the design is correct. In addition, most testbenches will also have some way of indicating that the SV vector is correct. This is called a self-validating testbench. It makes things easier and avoids issues related to sitting and looking at thousands of vectors visually. For example, suppose an output is called `Y_out` and you have a correct vector as `Y_correct`. Inside your testbench, you can place some behavioral construct

within the SV that indicates the logic is bad or good and is also an easy visual aid for indicating this event. An example would be as follows with a variable called `Error`:

```
assign Error = Y_out != Y_correct;
```

You are going to do a similar item with your design.

As the first laboratory demonstrated, making sure your SV works **before** you go to implementation is vital. As stated earlier in this document many times, it is impossible to create an exhaustive test for your design. You could conceivably make an exhaustive test for a smaller design, as in this laboratory, but normally this is not possible. For our work in this laboratory, we will create a 4-bit ripple-carry adder which is a good simple carry-propagate adder used in many microcontrollers and simple adder structures.

1.3 Testing Designs More Efficiently

In Lab 0, we just used delay statements to test input vectors into our design. Although this worked acceptably for Lab 0, for larger designs, such as in this laboratory, you have to figure an easier way to test more vectors to save time. For this design, we use the snippet of SV found in Figure 2.

In this HDL, a clock is used to test the design. A clock is a periodic signal that uses a repetition of pulses specifically used for synchronization. In Figure 2, the clock in the bottom half of figure. The clock is utilized to synchronize the data : that is, during the positive edge of the clock (i.e., HIGH value), we assign a random signal to A and B and then on the negative edge of the clock (i.e., LOW value), the values are displayed to a value called `desc3`.

The value of `desc3` is used to point to a file as seen by the declaration within Figure 2. All output designated by the `$fdisplay` keyword is sent to the output file. The output file is called `rca.out` in this example, but can be any filename you wish to use. After you run, `vsim -do file.do`, an output file will be created that you can use to see if your design works correctly. As explained earlier, the output also compares the true result to the computed result – i.e., any output at the end of the line which indicates an error.

The top part of Figure 2 just declares variables for use in this HDL snippet. The `integer` designation is just used in testbenches to declare variables that are typically used for `for` or `$fdisplay` statements. The `for` statement is utilized specifically to randomly assign inputs for this test. For this snippet, Figure 2 tests 128 vectors, but can be changed to any value.

The only caveat to the HDL in Figure 2 is the `$finish` statement. This item will end the simulation at the time designated by the time within the delay statement. For this design, it ends at time 1250 ns.

1.4 Test Vehicle: Ripple Carry Adders

Ripple carry adders (RCA) provide one of the simplest types of carry-propagate adder designs. Basically, a RCA is the method we utilize to add two numbers with paper and pencil. It basically adds each bit together and then passes the carry to each subsequent bit. This type of adder is typically called a carry-propagate adder (CPA). A CPA is an adder where the carries are connected together to form a sum of two input operands. A n-bit RCA is formed by concatenating n full adders (FA) similar how I would add several numbers together on paper. You can use the full adder block you used from Lab 0 to help you with this laboratory. The carry out from the k^{th} FA is used as the carry in of the $(k+1)^{th}$ FA, as shown in Figure 3. Unfortunately, since the connections for the carry-out depend on one another, most circuit implementations of RCAs consume a significant amount of delay. On the other hand, because the RCA implementation is straightforward it can be easily implemented and used for simple implementations and, more importantly, in courses where time is of the essence.

In order to create a RCA, you will have to use hierarchy similar to what you will do for the testbench. You will have to instantiate four (4) full adder (FA) blocks from Lab 0 to create your 4-bit ripple-carry adder. This will be similar to the following piece of SystemVerilog where the design is coded structurall or how it looks. Since we have more than 1 bit as an input, remember to use your Backus-Naur format (BNF) to efficiently create 4 bits as an input.

```
integer handle3;
integer desc3;
integer i;

initial
  begin
    handle3 = $fopen("rca.out");
    desc3 = handle3;
    #1250 $finish;
  end

initial
  begin
     for (i=0; i < 128; i=i+1)
       begin
          // Put vectors before beginning of clk
          @(posedge clk)
            begin
              A = $random;
              B = $random;
            end
          @(negedge clk)
            begin
            $fdisplay(desc3, "%h %h || %h | %h | %b", A, B, Sum, Sum_correct, (Sum == Sum_corr));
            end
       end // @(negedge clk)
  end
```

Figure 2: Sample HDL snippet to Test More Efficiently

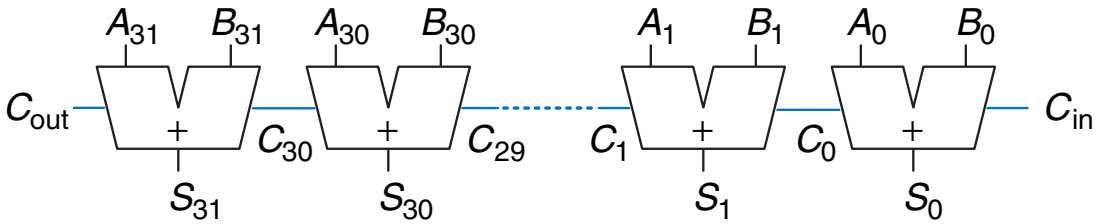

Figure 3: 32-bit Ripple-Carry Adder (RCA) Implementation from our Text [2].

```
module rca4 (input logic [3:0] a, b, input logic cin, output logic [4:0] sum);

  full_adder add1 ( );
  full_adder add2 ( );
  full_adder add3 ( );
  full_adder add4 ( );

endmodule
```

There are more advanced SV features that allow one to automate repeated instantiations (e.g., for-generate blocks), but we will just instantiate our full adder 4 times in this laboratory. It is important note that the sum output is 5-bits as the carry out can be used for the Most Significant Bit (MSB).

To create a good known golden vector for your adder, you will have to use a behavior construct to help you compute the correct sum or result within the testbench. Although behavior constructs in SV have gotten much better over the years, they still have a ways to go for some pieces of digital logic to efficiently synthesize well. Until you get comfortable designing digital logic structurally, it is best to use only the behavior HDL blocks that we recommend. To compute a known sum to compare versus your output, you should use this behavior construct within your SV testbench:

```
assign Sum_correct = A + B + Cin
```

In this example, it is important to note the use of the + operator is **not** an OR statement. Interestingly, these operators are rather awesome in that they actually synthesize very good digital implementations. In conclusion, for this laboratory your job is to use this behavior construct to compare the correct result of your 4-bit RCA is correct.

2. Implementation

To get started there are a couple of downloads that will help you get started. All of these files are found within a zip file on Canvas called `lab1.zip`. The first item is a complete SV file, testbench and DO file from Lab 0 you can use as a template for your test (i.e., the silly SV files). Second, there is a zip file of a demo program that is slightly different than Lab 0. In your new demo program, it contains logic to output a 4-bit value to a 7-segment display. It is important to state that you should not include your testbench in your implementation. The testbench is only used to verify your design works **before** you synthesize and implement it.

Once you design the SV of your 4-bit ripple-carry adder and test it thoroughly, implement it on the FPGA board. This should be fairly easy if you use the demo program (i.e., zipped file) to instantiate the design of your RCA. That is, once inside Vivado, open the unzipped file by navigating through `File->Project->Open` and opening the `Lab1.xpr` project. Once the project has opened, go to the `top_demo.sv` source by double-clicking the `top_demo` design. You can instantiate your RCA in the `top_demo.sv` but may need to add your SV files to your project to get it to synthesize correctly. Please ask your TA if you are unsure how to add files to your `Lab1.xpr` project.

2.1 Seven-Segment Display

If you examine your textbook in Example 2.10 [2], you can see some information on a 7-segment display. A seven-segment display decoder takes a 4-bit data input `D[3:0]` and produces seven outputs to control light-emitting diodes (LEDs) to display a digit. Some seven segment display implementations, such as ours in this laboratory, will display the output from 0 to F. However, some may only output 0 to 9 in applications where users do not know what a hexadecimal output are. The seven outputs are often called segments a through g, or Sa–Sg, as defined in Figure 2.47 in your textbook [2]. The digits are shown in Figure 2.48 in your textbook [2].

For this implementation, the 7-segment display implementation is already set up for you. You just have to send the 4-bit value to that digital instantiated part and it will display correctly on your DSDB board.

```
// 7-segment display
 segment_driver driver(
 .clk(smol_clk),
 .rst(btn[3]),
 .digit0(sw[3:0]),
 .digit1(4'b0111),
 .digit2(sw[7:4]),
 .digit3(4'b1111),
 .decimals({1'b0, btn[2:0]}),
 .segment_cathodes({sseg_dp, sseg_cg, sseg_cf, sseg_ce, sseg_cd, sseg_cc, sseg_cb, sseg_ca}),
 .digit_anodes(sseg_an)
 );
```

Figure 4: Instantiation of 7-segment display in `Lab1.zip` Demo program

For this laboratory, you want to send both A and B (i.e., both input operands) to your 7-segment display as well the sum. Since there are four 7-segment displays on your DSDB board, please use two 7-segment displays for your 5-bit sum (i.e., since c_{out} will be the `sum[5]`). To use the 7-segment display, just assign a value to one of the inputs within the `segment_driver` instantiation for `digit0` through `digit3`. You should make sure that any value given to this instantiation is a 4-bit quantity. You can also experiment with it to see known outputs - for example, you can see one of the values in Figure 4 is the value `0xF` or `4'b1111` given to `digit3`.

2.2 Power, Performance and Area (PPA)

As with the first laboratory, you should discuss your Power, Performance and Area (PPA) analysis. You should document in your report how much PPA is utilized for your final design. Similar to the previous laboratory, please report the total number of "slices" used for your design. We will expand your PPA for power and performance in later labs.

3. What to design?

Similar to Lab 0, you will be using the DSDB board. For this laboratory, you will also use the slide switches (i.e, `btn[7:0]`) to select the inputs. You might need to use the push buttons (i.e., `sw[3:0]` to indicate a carry in signal), the LEDs on board to help in debugging, and of course the 7-segment display explained in the previous Section.

The following are items will be needed to complete this laboratory:

1. Using the full adder from Lab 0, design a 4-bit structural version of a ripple carry adder to add two 4-bit input operands and a 1-bit Carry In signal. You can read more about the ripple-carry adders in Chapter 5 of your textbook. Although we did not discuss the ripple-carry adder (RCA) in depth, the idea should be easy to understand from Figure 3 above and Figure 5.5 in your text. What you need to do is instantiate 4 full adders and connect them appropriately.

2. Create a testbench from the `silly_tb.sv` testbench from Lab 0 to test your RCA.

3. Integrate a self-validating structure into your testbench so that it checks the answer automatically. Test, at least, 175 different vectors and demonstrate that the sum is indeed correct. That way, it should be easy to see if a signal is asserted when an error occurs. It should also be relatively easy to test all vectors and relax knowing that you have exhausitively verified your design too.

4. Once your RCA works, implement your design on the RCA using the demo program from `Lab1.zip` in the Vivado directory of your repository. Your design should display the input operands (i.e., A and B) and the final sum on the 7-segment display.

3.1 Importance

Although this laboratory is somewhat minimal compared to previous and future labs, the topics are of extreme importance for digital design. Verification dominates many products today and having good testbenches is vital, even in mixed-signal designs (i.e., analog and digital designs). I have heard hearsay from some companies that have told me they spend more than 70% of their design time on verification. Therefore, I implore you to learn as much as you can about testbenches and ask questions when possible. It will also pay **huge** dividends and may even shorten your development time in later labs if you can understand verification with testbenches well.

3.2 Handin

You should electronically hand in your HDL (all files that you want us to see) into Canvas. You should also take a printout of your waveform from your ModelSim simulation. Only one of your team members should upload the files and/or lab report. Please contact the instructor James Stine (james.stine@okstate.edu) for more help. Your code should be readable and well-documented. In addition, please turn in additional test cases or any other added item that you used. Please also remember to document everything in your Lab Report using the information found in the Grading Rubric.

3.3 Extra Credit

You will use the four 7-segment displays on your DSDB board to help highlight the input operands and your sum. However, the current 7-segment display that is instantiated in Figure 4 is written to display hexadecimal numbers. You could add another instantiated design between your RCA and the `segment_driver` to convert the output into decimal. This is not very hard, but might be a little tricky. Early computers did this with something called packed Binary-Coded Decimal (BCD) and I encourage you to look up this format to help you (see Exercise 1.64 in [2] as well as the Wikipedia entry for BCD [1]). There is also an excellent journal that discusses information on this too and is included in your repository [3]. Although the implementation is a little tricky it is not terribly difficult to implement.

References

[1] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE STD 1800-2009*, pp. 1–1285, 2009.

[2] S. Harris and D. Harris, *Digital Design and Computer Architecture: RISC-V Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2021.

[3] R. Kenney and M. Schulte, "High-speed multioperand decimal adders," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 953–963, 2005.

[1] `https://en.wikipedia.org/wiki/Binary-coded_decimal`