Lab 2A – Introduction to Testbenches

Introduction

A testbench is an important component in digital design. When developing a new or evolving design, an organization is typically divided into two functional units: a design team and a test/ verification team. The design team works on creating the actual product design. The test/ verification team focuses on: a) how to verify the product design meets the intended specification and b) how to test the product during manufacturing. In this lab, you will be introduced into this verification process.

Interestingly, both teams start with the same specification. In a simple design block, a *complete* specification can be based on an equation or truth table. By *complete*, I mean that the values of all output signals are defined for all possible combinations of input signals. Let's take as an example a three-input AND gate. We know what the output signal should be for all possibilities of the three inputs.

For simple designs, this process is fairly straightforward as will be shown in this lab. We need to apply all possible inputs to our design, and observe the outputs. We could do the output checking by hand, that is, run a simulation that applies these inputs to our design, and check the output for each input possibility. However, this can be tedious and error-prone. A better solution is to use the design specification to store the expected outputs – and compare them automatically. We will then encapsulate or instantiate our design inside of this testbench. This style of testbench is called a *self-checking testbench* because it automatically compares each output signal with an expected value.

The importance of such a testbench is:

- The functionality of the design goes through another *set of eyes*. That is, the verification team does not look at the design team's work, but rather the specification. So one has two independent views of the same design that in the end must match. This reduces functional design errors in large projects.
- As a design evolves (with a fixed specification), the product design team may go through several iterations to minimize power, increase speed, etc. – while the basic functional specification remains unchanged. The team can quickly verify that their changes have not broken the functionality of the product by reusing this *golden* self-checking testbench (or set of testbenches for a complex design).

In this lab, you will be provided a template for developing a self-checking testbench for simple designs that are specified with truth tables. You will then modify it to use for checking your design for a following lab.

Design/Module

Verilog/SystemVerilog is a language that can be used for both design and verification. While it is a *Hardware Description Language* (HDL) not all of the language constructs translate into hardware. In building this testbench we will use several constructs that cannot map into hardware. Remember, the testbench is an *abstract* design to apply stimuli to the inputs of a real design and compare the outputs from this design to their expected values. It is *not* part of the hardware implementation.

The testbench is the top module of a simulation. Think of it as the *main* function in C programming. Because the design modules, testbenches, and supporting text files are related, we will use a consistent file naming style for them. Let's assume we are going to test a design module (***and_3_inputs***) for a three-input AND gate hardware design as given here:

```
module and_3_inputs (output logic f, input logic a, b, c);
   // this is the design to test
   assign f = a & b & c;
endmodule
```

This design will in fact synthesize into a three-input AND gate and can be loaded into the FPGA hardware. To simulate this design, we need to instantiate this design inside of a testbench. For our design style, let's call the top level testbench the same module name, but with a ***tb_*** prefix.

```
module tb_and_3_inputs (); // this is the top level testbench wrapper
…
   // instantiate the unit under test (instance name uut)
   and_3_inputs uut (.f(), .a(), .b(), .c());
…
endmodule
```

Note a) there are no inputs and outputs at the top level for simulation and b) that the input and output connections to the ***uut*** are left empty for now.

Before continuing with the Verilog, let's define the specification for the example three-input AND gate in a truth table using multi-bit notation (start with index 0) for both the input and output signals, even though the output is only one bit wide. (This implementation will be useful when we have multiple output bits.)

data_in[2:0]	**expected_out[0:0]**
3'b000	1'b0
3'b001	1'b0
3'b010	1'b0
3'b011	1'b0
3'b100	1'b0
3'b101	1'b0
3'b110	1'b0
3'b111	1'b1

Now let's put this truth table into a text file that we will later read from the testbench. We will use a ***.txt*** extension for the file and use the same basename as the testbench for the design. So here would be the contents of ***tb_and_3_inputs.txt***:

```
000_0
001_0
010_0
011_0
100_0
101_0
110_0
111_1
```

Note that Verilog allows using the underscore to separate data patterns to make them more readable. Thus, when reading this file, the underscores will be ignored. Note also that it has eight rows (2^3) and four columns.

Now back to the Verilog code. We will assume that we can read all of this data into some variable, and put each row pattern (called a test vector) onto the ***data_in*** and ***expected_out*** signals, which we declare below.

```
module tb_and_3_inputs (); // this is the top level testbench wrapper
…
   logic [2:0] data_in;
   logic [0:0] expected_out;
   logic [0:0] sim_out;     // the simulated output;

   // instantiate the unit under test
   and_3_inputs uut (.f(sim_out), .a(data_in[2]), .b(data_in[1]),
      .c(data_in[0]));
…
endmodule
```

I have also declared signal: ***sim_out*** to hold the simulated output of the unit under test. Even though it is one bit wide in this example, I have used a multi-bit notation so it is easily reusable. I have also connected the ***data_in*** signals to the inputs of the unit under test.

Instantiation of the *unit under test* is a very important concept to understand. Remember, Verilog is a language that *describes hardware*. So instantiation is a parallel operation. Think of the *uut* as a piece of physical hardware – you change the inputs and the outputs change. Because this is parallel, this instantiation does not have any restrictions on where it needs to be placed within the module. However, we must declare our signals at the top of the testbench, so it cannot be placed before these internal signals are declared.

Next let's add some other necessary signals shown below.

- We need a two-dimensional signal called ***test_vectors***. We will read the data from the text file into this array. It will be the number of columns wide by the number of rows deep. Note, Verilog has this particular syntax for a two-dimensional signal that is a little different than other languages.
- We need a variable ***i*** to count the row number as we loop through the rows (also referred to as the index).
- Finally we need a variable ***mm_count*** to count the total number of mismatches when applying all of the patterns to the unit under test.

Note ***test_vectors*** is type ***logic*** (one bit per index); ***i*** and ***mm_count*** are of type ***integer*** (32 or 64 bits).

The resulting declarations for this example AND design would be:

```
logic [3:0] test_vectors [0:7];     // 8 rows by 4 columns
integer i; // variable for loop index
integer mm_count;  // variable to hold the mismatch count
```

Another useful coding style is to make use of the Verilog ***parameter*** statement. This statement defines constant value variables that will be defined in the first pass through the code by the simulator. They are similar to the ***#define*** statement in the C language. They help make the code easier to maintain and reduce errors. A typical coding style for such constant parameters is that they are *upper case*, as a reminder that they are constants, not signals, and should not be reassigned later in the code.

Let's add a few parameters to this design, and with appropriate substitutions to the indices. (Reminder, we count from zero.)

```
parameter ROWS=8, INPUTS=3, OUTPUTS=1;
logic [INPUTS-1:0] data_in;
```

```
   logic [OUTPUTS-1:0] expected_out;
   logic [OUTPUTS-1:0] sim_out;  // the simulated output;
   logic [INPUTS+OUTPUTS-1:0] test_vectors [0:ROWS-1];
   integer i; // variable for loop index
   integer mm_count;  // variable to hold the mismatch count
```

You can see the advantage of using parameters here. When reusing this template one only has to edit this single line for designs with different input/output dimensions.

So now we have declared all of our signals, and instantiated our unit under test. The result so far:

```
module tb_and_3_inputs (); // this is the top level testbench wrapper

   parameter ROWS=8, INPUTS=3, OUTPUTS=1;
   logic [INPUTS-1:0] data_in;
   logic [OUTPUTS-1:0] expected_out;
   logic [OUTPUTS-1:0] sim_out;  // the simulated output;
   logic [INPUTS+OUTPUTS-1:0] test_vectors [0:ROWS-1];
   integer i; // variable for loop index
   integer mm_count;  // variable to hold the mismatch count

   // instantiate the unit under test
   and_3_inputs uut (.f(f), .a(data_in[2]), .b(data_in[1]),
      .c(data_in[0]));

   // insert the test loop below
   …

endmodule
```

Next we have to read the text file and loop through the vectors comparing the outputs from the unit under test to the expected values. Verilog has a construct for doing this that you will learn about in the lectures. It is an ***initial*** block, i.e., a block of statements that will be executed sequentially. This whole block is a parallel construct. But the statements inside of it will be executed sequentially – which allows us to have looping functions, conditional statements, etc.

A simulator looks for *initial* blocks at the beginning of the simulation and starts executing them at simulation time zero. The simulator steps through the statements logically until it reaches the end of the block. This should become clearer as we work through this example. Let's set up this ***initial*** block:

```
   initial begin
      $dumpfile("tb_and_3_inputs.vcd");
      $dumpvars();

      mm_count = 0;   // zero mismatch count
      // read all of the test vectors from a file into
      // array: test_vectors
      $readmemb("tb_and_3_inputs.txt", test_vectors);

      // loop over all of the rows using a for loop
      for (i=0; i<ROWS; i=i+1) begin
      …
      end
   end
```

The first two lines of code in the initial block use Verilog functions: ***$dumpfile*** and ***$dumpvars***. These tell the simulator to write all of the signals from this simulation to a *vcd* file (*Value Change Dump*) called: **tb_and_3_inputs.vcd**. After the simulation is complete, these signals can be viewed with a waveform viewer to debug the design if necessary. For large designs and/ or long simulations this file can get quite large. There are ways to limit the number of signals stored by adding arguments to ***$dumpvars***. We will discuss this in later labs as necessary.

After the mismatch count variable is initialized to zero, you see a new Verilog function: ***$readmemb.*** It reads the data from the text file into the test vector variable – both the file name and variable defined in ***$readmemb's*** arguments. Note this reads the *whole file* in one function call – not line by line. That is why the array has to be large enough to handle all of the columns and rows.

I'm sure you recognize the *for loop* structure which is similar to that of other languages.

Now let's fill in the details of the *for* loop, remembering that in each loop we want to process one test vector (line of data in the text file).

Here are the details inside the *for* loop:

```
      for (i=0; i<ROWS; i=i+1) begin
         // read each vector (row) into the input data and
         // expected output variables - at this time the
         // data_in is applied to the unit under test
         {data_in, expected_out} = test_vectors [i];

         #10;    // artificial wait 10 ns for inputs to settle
         // now compare the output from uut to expected value
         if (sim_out !== expected_out) begin
            // display mismatch
            $display("Mismatch--loop index i: %d; input: %b, expected: 
%b, received: %b",
              i, data_in, expected_out, sim_out);

            mm_count = mm_count + 1;        // increment mismatch count

         end
         #10;    // add 10 ns for symmetry
      end
```

Reminder: the ***{…}*** is a concatenation operator. Thus, ***{data_in, expected_out}*** has a bit width equal to the number of columns – which is the width of ***test_vectors***. You can also see how each column bit is mapped to each input and output bit with this statement.

I put in an artificial delay of 10 ns to imagine a natural gate delay from the inputs being applied to the *uut*, until the output results would settle. The ***#10;*** actually means delay 10 simulation units. What is a simulation unit? This is typically defined in the first line of the file:

```
`timescale 1ns / 1ps
```

These two numbers are the time units and precision of the simulation, respectively.

Then the output from the *uut* is compared to the expected value (because of the parallel operation of the instantiation previously discussed). If there is not a perfect match the statements inside the ***if*** statement are executed.

You can see that the Verilog ***$display*** function is somewhat similar to a C language ***printf*** statement.

The mismatch count variable is incremented, another 10 ns delay is added for symmetry, and then the loop continues.

We just need a few more statements to finish up the testbench as shown below.

```
   // tell designer we're done with the simulation
   if (mm_count == 0) begin
      $display("Simulation complete - no mismatches!!!");
   end else begin
      $display("Simulation complete - %d mismatches!!!",
         mm_count);
   end
   $finish;
```

After completing the loop through all of the test vectors, the mismatch count variable is tested. An appropriate message is sent based on this condition. Then the simulation is formally completed by calling the ***$finish*** function. This basically cleans things up before exiting.

Putting it all together, we have:

```
`timescale 1ns / 1ps
module tb_and_3_inputs (); // this is the top level testbench wrapper

   parameter ROWS=8, INPUTS=3, OUTPUTS=1;
   logic [INPUTS-1:0] data_in;
   logic [OUTPUTS-1:0] expected_out;
   logic [OUTPUTS-1:0] sim_out;  // the simulated output;
   logic [INPUTS+OUTPUTS-1:0] test_vectors [0:ROWS-1];
   integer i; // variable for loop index
   integer mm_count;  // variable to hold the mismatch count

   // instantiate the unit under test
   and_3_inputs uut (.f(sim_out), .a(data_in[2]), .b(data_in[1]),
      .c(data_in[0]));

   initial begin
      $dumpfile("tb_and_3_inputs.vcd");
      $dumpvars();

      mm_count = 0;   // zero mismatch count

      // read all of the test vectors from a file into
      // array: test_vectors
      $readmemb("tb_and_3_inputs.txt", test_vectors);

      for (i=0; i<ROWS; i=i+1) begin
         // read each vector (row) into the input data and
         // expected output variables - at this time the
         // data_in is applied to the unit under test
         {data_in, expected_out} = test_vectors [i];

         #10;    // artificial wait 10 ns for inputs to settle
         // now compare the output from uut to expected value
         if (sim_out !== expected_out) begin
            // display mismatch
            $display("Mismatch--loop index i: %d; input: %b, expected:
%b, received: %b",
               i, data_in, expected_out, sim_out);

            mm_count = mm_count + 1;        // increment mismatch count

         end   // end of if
         #10;    // add 10 ns for symmetry
      end   // end of for loop

      // tell designer we're done with the simulation
      if (mm_count == 0) begin
         $display("Simulation complete - no mismatches!!!");
      end else begin
         $display("Simulation complete - %d mismatches!!!",
            mm_count);
      end
      $finish;
   end   // end of initial block

endmodule
```

The highlighted areas are edits that you would need to change for connecting to a similar design. Otherwise, the rest of the code can be unchanged.

Simulation/Verification

You will be given a testbench template (***tb_template.sv***) that follows the approach described above. You need to copy and modify it. The truth table specification and top level module definition for Lab 2 are provided below. The details of the design are provided in that write up.

d[4:1]	**e[7:1]**
4'b0000	7'b0000000
4'b0001	7'b0000111
4'b0010	7'b0011001
4'b0011	7'b0011110
4'b0100	7'b0101010
4'b0101	7'b0101101
4'b0110	7'b0110011
4'b0111	7'b0110100
4'b1000	7'b1001011
4'b1001	7'b1001100
4'b1010	7'b1010010
4'b1011	7'b1010101
4'b1100	7'b1100001
4'b1101	7'b1100110
4'b1110	7'b1111000
4'b1111	7'b1111111

```
module hamming7_4_encode(output logic [7:1] e, input logic [4:1] d);
…
endmodule
```

Create the testbench module for this design (**tb_hamming7_4_encode.sv**) based on **tb_template.sv**. Here are the steps:

- Open **tb_template.sv** in one window.
- Open a new file in another window: **tb_hamming7_4_encode.sv**
- Copy the template code into the hamming code and edit the appropriate areas as discussed previously in this document.
- You can test the syntax of your entry by running **tb_hamming7_4_encode.m_sim**.

You will also need to create a text file (**tb_hamming7_4_encode.txt**) that contains the data from the truth table above.

When you have these two files completed, you test your simulation operation with this testbench. Right-click on **hamming7_4_encode.m_sim** and select *Run*. The simulation should start with the testbench applying the stimuli (16 different test vectors from your *txt* file) to the Hamming encoder design. However, since you will be doing this design in next week's lab, the design is empty and you should see the following mismatches (the received data is unknown denoted by x's):

```
Mismatch--loop index i:           0; input: 0000, expected: 0000000, received: xxxxxxx
Mismatch--loop index i:           1; input: 0001, expected: 0000111, received: xxxxxxx
Mismatch--loop index i:           2; input: 0010, expected: 0011001, received: xxxxxxx
Mismatch--loop index i:           3; input: 0011, expected: 0011110, received: xxxxxxx
Mismatch--loop index i:           4; input: 0100, expected: 0101010, received: xxxxxxx
Mismatch--loop index i:           5; input: 0101, expected: 0101101, received: xxxxxxx
Mismatch--loop index i:           6; input: 0110, expected: 0110011, received: xxxxxxx
Mismatch--loop index i:           7; input: 0111, expected: 0110100, received: xxxxxxx
Mismatch--loop index i:           8; input: 1000, expected: 1001011, received: xxxxxxx
Mismatch--loop index i:           9; input: 1001, expected: 1001100, received: xxxxxxx
Mismatch--loop index i:          10; input: 1010, expected: 1010010, received: xxxxxxx
Mismatch--loop index i:          11; input: 1011, expected: 1010101, received: xxxxxxx
Mismatch--loop index i:          12; input: 1100, expected: 1100001, received: xxxxxxx
Mismatch--loop index i:          13; input: 1101, expected: 1100110, received: xxxxxxx
Mismatch--loop index i:          14; input: 1110, expected: 1111000, received: xxxxxxx
Mismatch--loop index i:          15; input: 1111, expected: 1111111, received: xxxxxxx
Simulation complete -          16 mismatches!!!
```

You now have completed the design of the testbench for the Hamming encoder. You will use this next week to test that design.

As a reference, the ***and_3_inputs*** files are included in the lab2 directory. If you like you can run this simulation also (**and_3_inputs.m_sim**).

Viewing the waveform

As described earlier, the ***vcd*** file contains the signal data from the simulation. This is very useful if you need to debug your design. You can view this signal data (i.e., timing diagram) using the ***WaveTrace*** extension in VS Code that you downloaded during Lab 0. Your lab instructor will demonstrate using WaveTrace. WaveTrace has a very similar interface to the waveform viewer explained below.

[Optional]

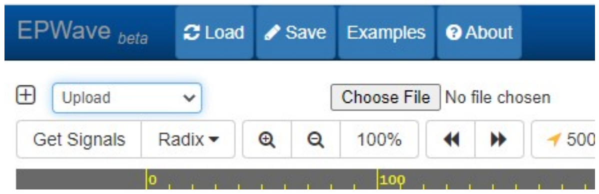
If WaveTrace is not setup, there are alternative options. I outline the steps below for using the online waveform viewer: ***EPWave***. There are other waveform viewers, a popular freeware one is ***gtkwave***.

Download the vcd file to your local PC

Locate the *vcd* file of interest. In this example, it would be **tb_hamming7_4_encode.vcd**. Do ***NOT*** open this file. For some designs, it will be large and may lock up your system as it tries to read and display it into a window. Right-click on the file and select: *Download*.

Open EPWave and upload the file from your local PC

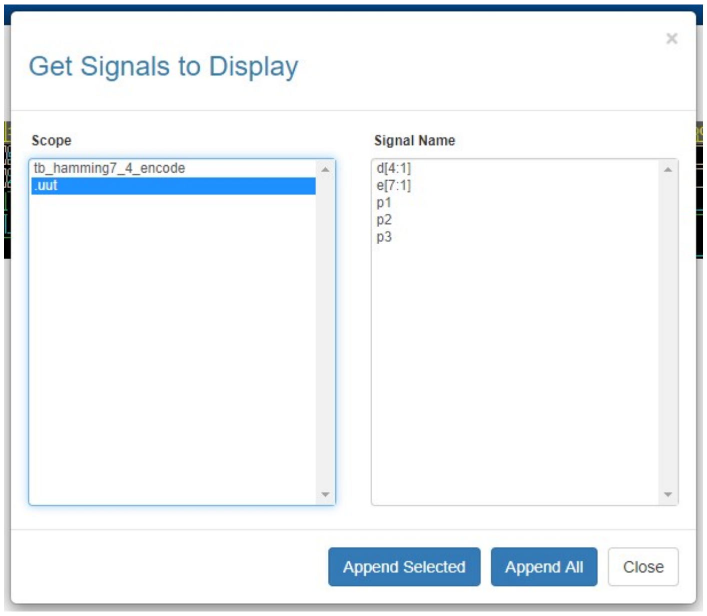
Go to: https://www.edaplayground.com/w/home (you may want to bookmark this location). You may need to create a free account with the sponsoring company (Doulos).



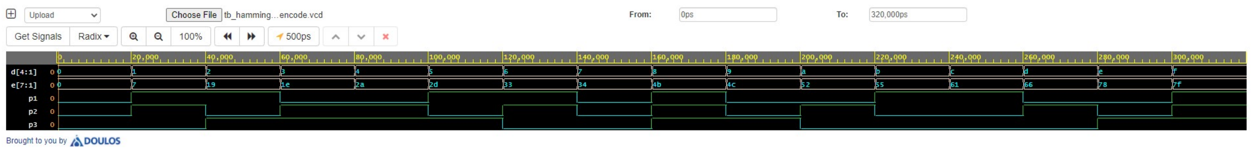
Select *Upload* from the dropdown box in the upper left. Then select the *Choose File* button to get your local *vcd* file.

Select the signals to view

Select the *Get Signals* under the *Upload* box. You will be presented with a window that describes the hierarchy of your design.

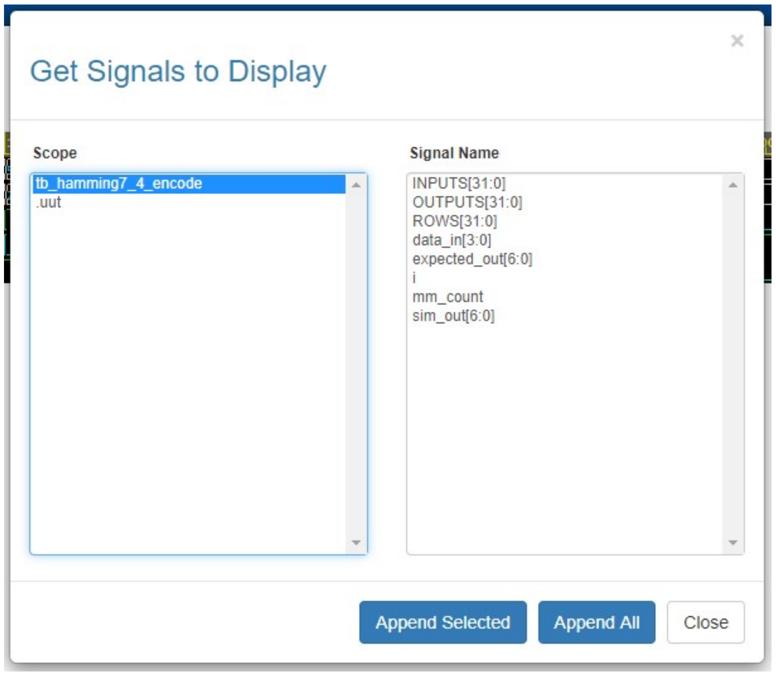


By highlighting ***.uut*** (instance name of the *unit under test – hamming7_4_encode*), the *Signal Name* box is populated with the signals inside that instance. Since this is a small design, you can select *Append All* and they will appear in the wave viewer.



The default for displaying a multi-bit signal is hexadecimal format, you can change this to binary by selecting the *Radix* dropdown. The above figure is a waveform of the completed design, which you will have done next week. Note a few things here. All of the signals are changing every 20 ns. This is exactly what was defined inside the testbench: two 10 ns delays inside the *for* loop. There are 16 different values for the signal ***d*** inside the Hamming design. And you see all of them defined here. The signal ***e*** is the output of the completed Hamming design. You can see that it matches the expected values in the truth table above. The other ***p*** signals are internal parity signals.

If you wanted to, you could add the internal signals from the top-level testbench (*Get Signals* button). This does not have an instance name, but you can see from the image below that all of the internal signals are available. These signals are typically not as informative in debugging the design. However, if you have mismatches, looking at when ***mm_count*** changes will point you to the failing pattern.


Get Signals to Display

Scope
tb_hamming7_4_encode
.uut

Signal Name
INPUTS[31:0]
OUTPUTS[31:0]
ROWS[31:0]
data_in[3:0]
expected_out[6:0]
i
mm_count
sim_out[6:0]

Synthesis

There is no synthesis for this lab.

I emphasize here the other important aspect of creating a self-checking simulation testbench. If your design passes simulation, it is very, very likely that your synthesized design (in hardware) will operate to the specification. I sometimes hear from students having trouble debugging their designs. When asked if their design passed simulation: “No it had mismatches, but I thought I would try and check it in the hardware anyway.” I can guarantee you that it will not work in the hardware. While you may be able to load your flawed design into the hardware – it will behave with the same design errors that it revealed in the mismatches of the simulation.

So follow this rule: If your design does not pass simulation, don’t waste your time synthesizing it and loading it into the hardware. The mismatches in the specification will *not* get magically corrected from this step.

Finally, if you follow this path and think you can debug the problem in hardware, it will be very difficult. The main reason is that you have limited visibility into the FPGA. So it is much easier to debug your design errors during this simulation process where you have complete visibility of any signal in the design at any time in the simulation. So debugging is much easier with a self-checking testbench – that basically points you to the problem patterns. For our purposes in this course, *a passing simulation almost guarantees that your design will operate correctly in hardware*.