# ECE 128 – Verilog Tutorial: Practical Coding Style for Writing Testbenches

Created at GWU by William Gibb, SP 2010
Modified by Thomas Farmer, SP 2011

**Objectives:**
- Become familiar with elements which go into Verilog testbenches.
- Write a self-checking testbench

**Assumptions**:
- Student has a coded a full adder module.

**Introduction:**

The ASIC design flow is as follows:

Specification
▼
**RTL Coding and Simulation**
▼
Logic Synthesis
▼
Optimization
▼
Gate Level Simulation
▼
Static Timing Analysis
▼
Place and Route
▼
Static Timing Analysis
▼
Preliminary Netlist Handoff

In this lab, we are at the “RTL Coding and Simulation” stage in the ASIC Flow.

In the previous tutorial we saw how to perform simulations of our verilog models with NCVerilog, using the sim-nc/sim-ncg commands, and viewing waveforms with Simvision. This is a very useful approach to testing digital models, but can become very cumbersome if the amount of signals that you are looking at is more than a dozen, or you are running very long simulations.

Instead of relying solely on visual inspection of waveforms with simvision, your Verilog test benchs can actually do inspection for you - this is called a selfchecking testbench. In order to build a self checking test bench, you need to know what goes into a good testbench. So far examples provided in ECE126 and ECE128 were relatively simple test benches.

**The Basic Testbench**

The most basic test bench is comprised of the following set of items:

```
________________________________________
|                                        |
| Verilog Test Bench                     |
|              _____________________     |
|             |                     |    |
|             |      ________       |    |
|    DUT      |------>  |        |-------> |DUT     |
| Stimulus    |------>  | Verilog|------>  |Monitor|
|             |------>  | DUT    |         |       |
|_____________|         |________|         |_______|
               ^^^ Inputs       ^^^^ Outputs
```

1. A device under test, called a DUT. This is what your testbench is testing.
2. A set of stimulus for your DUT. This can be simple, or complex.
3. A monitor, which captures or analyzes the output of your DUT.
4. You need to connect the inputs of the DUT to the testbench.
5. You need to connect the outputs of the DUT to the testbench.

You can see in the below example, from lab #1, mux_tb.v, the basic requirements for a testbench have been satisfied.

```
// Example Testbench from 128 lab #1: mux_tb.v
//
module mux_tb();

      wire c;
      reg a,b,s;

      mux m1(c, a, b, s) ;

      initial begin
            #0 a=1'b0;
            b=1'b0;
            s=1'b0;
            #5 a=1'b1;
            #5 s=1'b1;
            #5 $finish; // The $finish call ends simulation.
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

There is a DUT, set of stimulus and a waveform capture. However, this testbench doesn't have much structure to it, therefore it is difficult to expand upon. In this lab we will improve this testbench and give it more structure. Then you can use this modified testbench as a model for all future testbenches you create in verilog..

**Structured Verilog Test Benches**

A more complex, self checking test bench may contain some, or all, of the following items:

1. **Parameter definitions**
2. **Preprocessor Directives**
3. **The timescale directive**
4. **Include Statements**
5. **DUT Input regs**
6. **DUT Output wires**
7. **DUT Instantiation**
8. **Initial Conditions**
9. **Generating Test Vectors**
10. **Debug output**
11. **Using Memory in a testbench**
12. **Events in Verilog**

Some explanations for all of these items:

**1) Parameter definitions**
Parameterize items in your test bench - this makes it much easier for you and others to read and understand your testbench. You can also put parameters in your modules (not just test benches). It allows you to customize or tweak your testbench as needed. We recommend that you put these near the top of your testbench for easy manipulation. Commonly parameterized items are as follows.

- Clock period
- finish time
- control words
- data widths

Let’s modify the MUX test bench to have parameters:

```
module mux_tb();

      parameter finishtime= 5 ;

      wire c;
      reg a,b,s;

      mux m1(c, a, b, s) ;

      initial begin
            #0 a=1'b0;
            b=1'b0;
            s=1'b0;
            #5 a=1'b1;
            #5 s=1'b1;
            #finishtime $finish;   // The $finish call ends simulation.
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

***Note, wherever the word: “finishtime” is encountered, it is replaced with the number 5.***

**2) Preprocessor Directives**

A preprocessor directive works in a very similar fashion to a parameter. It is essentially a variable that gets replaced when encountered. Below, we have included **`define DELAY** at the top of the test bench. This is a preprocessor directive. Whenever the word "`DELAY" is encountered in the code, it is replaced by the number 5. Read the following example:

```
`define DELAY 5
module mux_tb();

      parameter finishtime= 5 ;

      wire c;
      reg a,b,s;

      mux m1(c, a, b, s) ;

      initial begin
            #0 a=1'b0;
            b=1'b0;
            s=1'b0;
            #`DELAY a=1'b1;
            #`DELAY s=1'b1;
            #finishtime $finish; // The $finish call ends simulation.
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

**So, what is the difference between a parameter and a preprocessor directive?**

**A preprocessor directive:**

-comes before the 'module' statement
-is typically used for variables you want GLOBAL to all your modules
-we typically use UPPERCASE for a preprocessor directive
-a preprocessor directive can be SET at runtime, from the compile line like this:

```
sim-nc mux_tb.v +define+DELAY=10
```

By running the command above, the value of DELAY will now be worth 10 instead of 5

**A parameter:**

-can come after the 'module' statement
-is typically used for variables LOCAL to just the module you are working on
   -for instance, in our test bench, FINISHTIME is only important for a our TB, not any other modules
-we typically use LOWERCASE for a parameter's name
-a parameter cannot be changed or set at runtime on the compiler line

**3) The timescale directive**

If a timescale statement is included at the top of a module, as follows:

```
`timescale 1ns/10ps
`define DELAY 5
module mux_tb();

      parameter finishtime= 5 ;

      wire c;
      reg a,b,s;

      mux m1(c, a, b, s) ;

      initial begin
            #0 a=1'b0;
            b=1'b0;
            s=1'b0;
            #`DELAY a=1'b1;
            #`DELAY s=1'b1;
            #finishtime $finish; // The $finish call ends simulation.
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

The value of “time” in the simulator is given a unit. So for `timescale 10ns/10ps, the first number represents the value of one time unit. The second number represents the precision that is kept by the simulator. Now when a # statement is encountered, it has a unit of time:

```
`timescale 10ns/10ps
...
#5 a=1’b;
```

Now the #5 means the simulator will wait “5 nanoseconds” before preceding to the next line

For this example:

```
#5.01 a=1’b;
```

the simulator will wait “5 nanoseconds” and “10 picoseconds” before preceding to the next line. However no further precision can be indicated: 5.001 will just be considered a wait of 5 nanoseconds.

**4) Include Statements**

Include statements are similar to C style include statements. They allow a another file to be a part of the current file. They are commonly used to include a file with the timescale directive. They can also be used outside of testbenches, often for global constants.

Let's separate our testbench into two files:

---

```
// filename: globals.vh
//
`timescale 1ns/10ps
`define DELAY 5
```

---

```
// filename: mux.v
//
`include "globals.vh"
module mux_tb();

      parameter finishtime= 5 ;

      wire c;
      reg a,b,s;

      mux m1(c, a, b, s) ;

      initial begin
            #0 a=1'b0;
            b=1'b0;
            s=1'b0;
            #`DELAY a=1'b1;
            #`DELAY s=1'b1;
            #finishtime $finish; // The $finish call ends simulation.
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

---

We can use the 'globals.vh' later when it is needed and we do not need to include when we compile it:

```
sim-nc mux.v
```

(it will find globals.vh if it is in the same directory)

**5) DUT Input regs**

Because you are typically using procedural style verilog to create your testbench, any variable you 'assign' data to, you must use a REG type to do so. This is why the inputs to your DUT, will be REGs. In part #1 of this lab, notice variables a,b, and s are regs.

**6) DUT Output wires**

The output from the DUT needs to be connected to something, for this you'll want to use wires. Instance your output wires together. Since you are not assigning the data to the outputs inside the procedural block, (your module is), the outputs can be assigned to wires and not registers (REGs).

**7) DUT Instantiation**

You'll have to instance your device under test. Be sure that you get in the practice of using named ports when you instantiate modules. Named ports do not depend on the port order, only on port name. It's a more portable way of instantiating modules.

Never use this style to instantiate a module:

```
mux m1(c, a, b, s);
```

*Use this style:*

```
mux m1(.c(c), .a(a), .b(b), .select(s));
```

Now the local variable s, is connected to the port 'select' inside the module. But notice, thie named port style allows you to change the order of the instance:

```
mux m1(.c(c), .a(a), .select(s), .b(b) );
```

Even with the order switched this code will work just fine.

**8) Initial Conditions**
Your first initial block should start setting the initial conditions for your test bench. You should set all of your TB registers to an initial value, including your clock(s). Reset signals may be set to a normal, non-reset state. This is so that you can perform any device specific resets as the beginning of your device stimulus.

It is a good idea to have a block for ending the simulation at a specified time included here. You could add a #FINISHTIME $finish; line in your initial block, or add a separate initial block, just for causing the simulation to end. This is to prevent your TB from running forever, and should be longer than your expected time of test, unless your purposely running only a portion of the testbench.

View the portions of the testbench in bold print to see the changes:

```
`include "globals.vh"
module mux_tb();

      parameter finishtime= 5 ;

      wire c;
      reg a,b,s;

      mux m1(.c(c), .a(a), .b(b), .select(s));

      initial begin     // initialize all variable in a separate initial block
            a=1'b0;
            b=1'b0;
            s=1'b0;
      end

      initial begin
            #`DELAY a=1'b1;
            #`DELAY s=1'b1;
            #finishtime // everything below will printout after "finishtime"
                        // expires
            $display ("Finishing simulation due to simulation constraint.");
            $display ("Time is - %d",$time);
            $finish;
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

**9) Generating Test Vectors**

Instead of using separate variables for our inputs to the MUX, (a,b,s), we could use a single 3-bit register to hold the data. Then we can use it to create all 2^3=8 combinations possible as follows:

```
`include "globals.vh"
module mux_tb();

      parameter finishtime= 5 ;

      integer N;
      wire c;
      reg [2:0] test_vectors; // 3-bit wide test vector

      mux m1(.c(c), .a(test_vectors[2]), .b(test_vectors[1]), .select(test_vectors[0]));

      initial begin     // initialize all variables
            test_vectors = 3'b000;
      end

      initial begin
            for(N=0; N<7; N=N+1)
                #`DELAY test_vectors = test_vectors + 1;

            #finishtime // everything below will printout after "finishtime"
            expires
            $display ("Finishing simulation due to simulation constraint.");
            $display ("Time is - %d",$time);
            $finish;
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

Now, all the data is stored in "test_vectors." The most significant bit, is assigned to "A", the next to "B" and the last to "S" or the select.

In the first initial block, the register is initialized to all 0's.

In the second initial block, we generate all the test vectors, 000 through 111, in a for loop. Notice that the #`DELAY waits 5 time units before going to the next test vector.

## 10) Debug output

Often times, we have complicated waveforms for the modules we are testing. We do not always want to open simvision to verify the waveforms are correct. We can add some TEXT output to our test bench using two verilog commands: $display and $monitor. Look at the updated test bench:

```
`include "globals.vh"
module mux_tb();

      parameter finishtime= 5 ;

      integer N;
      wire c;
      reg [2:0] test_vectors; // 3-bit wide test vector

      mux m1(.c(c), .a(test_vectors[2]), .b(test_vectors[1]), .select(test_vectors[0]));

      initial begin     // initialize all variables
            $display ("-----------------------------------------------------");
            $display ("                                            ABS");
            $display ("-----------------------------------------------------");
            $monitor ("TIME = %d, test_vectors= %b, c= %b", $time, test_vectors, c);
            test_vectors = 3'b000;
      end

      initial begin
            for(N=0; N<7; N=N+1)
                #`DELAY test_vectors = test_vectors + 1;

            #finishtime // everything below will printout after "finishtime"
            expires
            $display ("Finishing simulation due to simulation constraint.");
            $display ("Time is - %d",$time);
            $finish;
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

*The* ***`$display`*** *statement will print only 1 time. The* ***`$monitor`*** *statement will printout every time any of the variables listed in its list are changed. The output of this testbench will be:*

```
-----------------------------------------------------
                                            ABS
-----------------------------------------------------
TIME =                    0, test_vectors= 000, c= 0
TIME =                    5, test_vectors= 001, c= 0
TIME =                   10, test_vectors= 010, c= 1
TIME =                   15, test_vectors= 011, c= 0
TIME =                   20, test_vectors= 100, c= 0
TIME =                   25, test_vectors= 101, c= 1
TIME =                   30, test_vectors= 110, c= 1
TIME =                   35, test_vectors= 111, c= 1
```

## 10) Self-Checking

The above test bench (from step 9) is better, we can view the output on the screen, but it would be better if the test-bench could check itself and let us know if any test case has failed. At the top level testbench this is always preferred so that the waveforms do not have to be viewed. This can be done many different ways, but for our MUX example, view the following code:

```
`include "globals.vh"
module mux_tb();

      parameter finishtime= 5 ;

      integer N;
      wire c;
      reg [2:0] test_vectors; // 3-bit wide test vector

      mux m1(.c(c), .a(test_vectors[2]), .b(test_vectors[1]), .select(test_vectors[0]));

      initial begin     // initialize all variables
            $display ("----------------------------------------------------");
            $display ("                                           ABS");
            $display ("----------------------------------------------------");
            $monitor ("TIME = %d, test_vectors= %b, c= %b", $time, test_vectors, c);
            test_vectors = 3'b000;
      end

      initial begin
            for(N=0; N<7; N=N+1)begin
                  #`DELAY test_vectors = test_vectors + 1;
                  #(`DELAY/5)
                  if      (c==test_vectors[2] && test_vectors[0]==1) $display ("PASS");
                  else if (c==test_vectors[1] && test_vectors[0]==0) $display ("PASS");
                  else $display ("FAIL");
            end

            #finishtime // everything below will printout after "finishtime"
            expires
            $display ("Finishing simulation due to simulation constraint.");
            $display ("Time is - %d",$time);
            $finish;
      end

      initial begin
            // Open a db file for saving simulation data
            $shm_open ("mux_tb.db");
            // Collect all signals (hierarchically) from the module "mux_tb"
            $shm_probe (mux_tb,"AS");
      end

endmodule
```

Now the testbench will test the output of the MUX as it runs through each test vector. Notice the testbench waits for 1ns after setting the input to account for any propagation delay in the MUX. It is very important to wait for the outputs to stabilize before checking their values.

## 11) Using Memory in a testbench

The next two sections will make more sense to you when you encounter the CPU project. You will need to create RAM for your CPU. This example covers how to instantiate RAM and then load it with data.

To fill arrays with data use a loop to generate the data or use the $readmemh or $readmemb directives to load data from a file.

```
parameter WIDTH=8; //multiple of two
parameter DEPTH=16;
parameter PATTERN = {WIDTH/2{2'b10}};

reg [WIDTH-1:0] test_memory [DEPTH-1:0];

initial
for(N=0; N<DEPTH; N=N+1)
      test_memory[N] = PATTERN + N;

end example

example:

// This memory model is an ram block meant for simulation. It can be used with a
// microprocessor to provide a memory file. It would be instantiated in a test
// bench and connected to the microprocessor.  It contains an example of using the
// $readmem directives. These can be used in top level test benches to load arrays
// with test vectors.

module exmem #(parameter WIDTH = 8, RAM_ADDR_BITS = 8)
( input clk, en, memwrite, input [7:0] adr,  input [7:0] writedata,  output reg
[7:0] memdata  );

integer i;
integer k;
reg [7:0] mips_ram [0:256];

// The following $readmemh statement initializes the RAM contents via an external
file (use $readmemb for binary data). The fib.dat file is a list of bytes, one per
line, starting at address 0.

initial begin
      for(i=0; i<256; i=i+1)
            mips_ram[i]=8'b0;
      $display("memory scrubbed");

      $readmemh("fib.dat", mips_ram);   //or $readmemb("fib.dat", mips_ram);

      $display ("File loaded.");
      $display("Contents of Mem after reading data file:");
      for (k=0; k<256; k=k+1)
            $display("%d:%h",k,mips_ram[k]);
end

// The behavioral description of the RAM - note clocked behavior
always @(negedge clk)
if (en) begin
      if (memwrite)
            mips_ram[adr] <= writedata;
            memdata <= mips_ram[adr];
      end endmodule
```

**12) Events in Verilog**

One can setup an event or something similar to a “jump/GOTO” in verilog.  This can be very handy in a self-checking test bench that needs to have a more detailed set of cases when tests pass or fail.  View the following example:

Events are triggers for always@ blocks of the following form:

always @(event_name) begin....end blocks

These blocks enter when the event event_name is triggered. These always blocks can have procedural code in them, so they can easily be used to define a sequence of actions. These events can be both stimuli and DUT checking.

```
//
// do stuff
//
-> event_name;
always@(event_name)
begin
//
// do stuff
//
end
```

Their use can lead to very robust test benches. See the example below

example:

```
//initialize test pattern and start simulation
initial
begin
for(N=0; N<DEPTH; N=N+1)
       test_memory[N] = PATTERN + N;
       #10 -> reset_fifo_a;
End

//event definitions
always @(reset_fifo_a)
begin
       $display ("Aysnc Reset");
       #10 mrst_n=0;
       #10 ;
       for(N=0; N<DEPTH; N=N+1)
              if(DUT.memory[N]!={WIDTH{1'b0}})
              begin
                     $display("Error in async reset");
                     $display("Time %d, DUT.memoryt=%d",$time, DUT.memory[N]);
                     $finish;
              end
       #10 mrst_n=1;
       $display ("Async Reset completed");
       -> write_till_full;
End

always @(write_till_full)
begin
       countIn=0;
       $display("write_full start");
       #10 en=1;
       for(N=0; N<DEPTH; N=N+1)
              if(!full)
```

```
            fork
                  #10 din=test_memory[N];
                  wr=1;
                  #20 clk=1;
                  #20 countIn = countIn + 1;
                  #30 clk=0;
            join
      if(countIn != DEPTH-1)
      begin
            $display("Error in write full");
            $display("Time %d, CountIn = %d", $time, countIn);
            $finish;
      end
      #10 en=0;
      wr=0;
      $display("Time %d, Count = %d", $time, countIn);
      $display ("Write Full ended");
      #10 -> read_out;
End

always @(read_out)
begin
      countOut=0;
      $display("read out start");
      #10 en=1;
      #10 rd=1;
      for(N=0; N<DEPTH-1; N=N+1)//fifo only fills to depth-1
            begin
            //current location is always there
            if(dout!=test_memory[N])
            begin
                  $display("Error in read out");
                  $display("Time %d, CountOut = %d", $time, countOut);
                  $display("Read Value %d, Memory Value
                  %d",dout,test_memory[N]);
                  $finish;
            end
            else
            begin
                  $display("Read Value %d, Memory Value
                  %d",dout,test_memory[N]);
            end
            #10 countOut=countOut+1;
            if(!empty) //clock out the next value;
            begin
                  #10 clk=1;
                  #10 clk=0;
            end
      end
      #10 rd=0;
      #10 en=0;
      $display("read out end");
      #10 ;
      if(second_pass)
            -> fin;
      else
            -> reset_fifo_s;
end
.
. more events
.
```

**Tutorial Assignment**

Using the existing test benches you've developed for your full adder circuit expand upon them. It is recommended that you perform this work in a new directory, in order to keep your individual labs separate.

1) Update your test bench so all values are parameters or define macros (delays, finishtime, etc). Separate the global variables into a separate include file. (5 points)

2) Update your test bench to test all possible input & output combinations for the full adder circuit from lab 2 using loops as discussed in this tutorial. (5 points)

3) Make your testbench properly structured, for readability. Look at the testbench tutorial and example test bench. Be sure to have separate initial blocks to initialize variables and to generate the test vectors. (5 points)

4) Make your testbench self-checking. Whenever the adder inputs change check the output sum and carry to make sure that they are correct. Be sure to give yourself some time between updating inputs and checking the output, in the event that you reuse this testbench in a timing simulation. (5 points)

5) Write a 1-2 (or more) paragraphs describing your testing strategy, how your testbench decides if the DUT is successful and what you learned in this lab. (5 points)

In your report, be sure to include the following:

- Verilog Testbench
- Verilog Code for any modules instantiated
- Simulation log file
- Waveform showing the correct operation of your adder.

You can look at ECE126 Lab 9C, step 3 for an additional reference of how to do self-checking.

6) Repeat steps 1-5, but now update your test bench for the DFF you created as part of HW assignment #1.

This is an in-lab assignment to be completed before you leave lab today.