Writing Test Benches

January 10, 2018 • Justin Rajewski

Test benches are used to simulate your design without the need of any physical hardware. The biggest benefit of this is that you can actually inspect every signal that is in your design. This definitely can be a time saver when your alternatives are staring at the code, or loading it onto the FPGA and probing the few signals brought out to the external pins. However, you don't get this all for free. Before you can simulate your design you must first write a **test bench**.

What is a Test Bench

What exactly is a **test bench**? A test bench is actually just another Verilog file! However, the Verilog you write in a test bench is not quite the same as the Verilog you write in your designs. This is because all the Verilog you plan on using in your hardware design **must be** synthesizable, meaning it has a hardware equivalent. The Verilog you write in a test bench does not need to be synthesizable because you will only ever simulate it!

Let us assume we have a module called **basic_and** that looks like this.

```
module basic_and #(parameter WIDTH = 1)(
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] out
  );

  assign out = a & b;

endmodule
```

The functionality of this module should be fairly apparent. It takes two inputs, **a** and **b**, of width **WIDTH** and **ands** them together for the output, **out**.

If we want to now test this module to make sure it is actually doing what we think it is, we can write a test bench! It is best practice to name the test bench associated with a module the same as the module with **_tb** appended. For example, the test bench for **basic_and** is named **basic_and_tb**.

Take a look at what a test bench for **basic_and** could look like.

```
module basic_and_tb();

  reg [3:0] a, b;
  wire [3:0] out;

  basic_and #(.WIDTH(4)) DUT (
    .a(a),
    .b(b),
    .out(out)
  );

  initial begin
    a = 4'b0000;
    b = 4'b0000;
    #20
    a = 4'b1111;
    b = 4'b0101;
    #20
    a = 4'b1100;
    b = 4'b1111;
    #20
    a = 4'b1100;
    b = 4'b0011;
    #20
    a = 4'b1100;
    b = 4'b1010;
    #20
    $finish;
  end

endmodule
```

A test bench starts off with a module declaration, just like any other Verilog file you've seen before. However, it is important to notice the test bench module does not have any inputs or outputs. It is entirely self contained.

After we declare our variables, we instantiate the module we will be testing. In this case it's the **basic_and** module. Notice the name is **DUT**. DUT is a very common name for the module to be tested in a test bench and it stands for **D**evice **U**nder **T**est.

```
initial begin
```

This line has something new, the **initial** block. The inital block is used similarly to an **always** block except that the code in the block will only run once, at the start of the simulation. Because there is no equivalent to an initial block in hardware, initial blocks are not synthesizable and can only be used in test benches.

Lines 13 and 14 simply assign values to **a** and **b**.

```
15  #20
```

This is another new line. The **#** syntax is used to specify a **delay**. In this case this tells the simulator to wait 20 units of time. This is important because without these delays we would have no time to observe how **a** and **b** affect the circuit. Again, there is no hardware equivalent to a delay like this, so these statements are not synthesizable.

It is very important to understand that we are still simulating **hardware**. That means that the delay of 20 units only affects the block the delay is in. The rest of your design will keep chugging away as if there was no delay. This is important because it allows you to specify inputs and wait for the output to be generated.

Finally we get to the following line.

```
28  $finish;
```

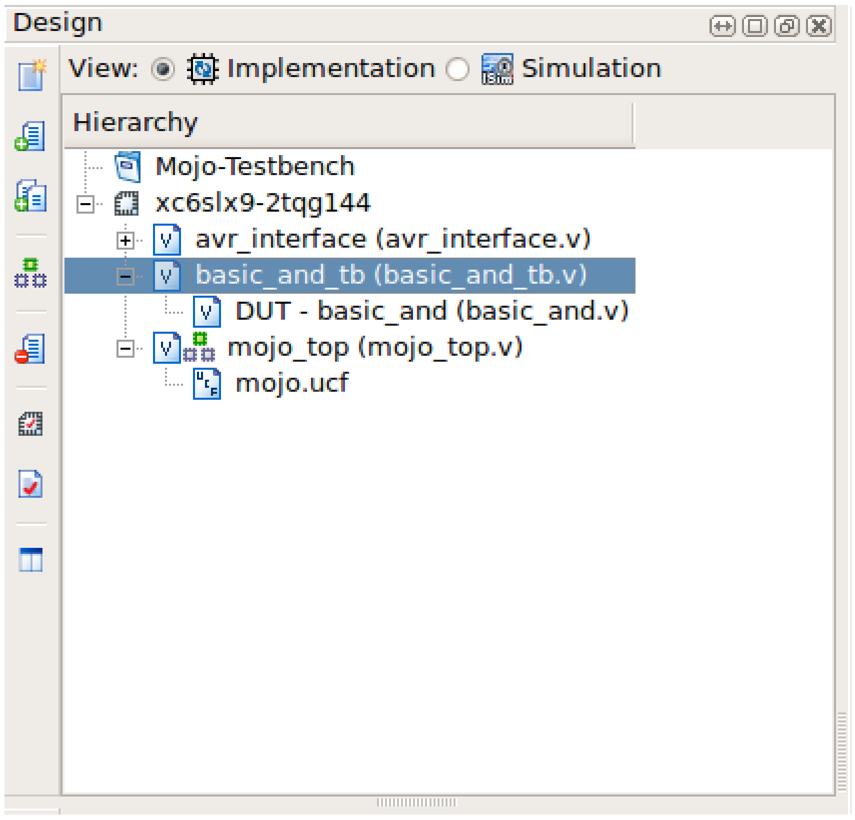
This line tells the simulator we are done simulating.

Running a Test Bench

As with all tutorials, we will start with the Mojo Base Project, so download a fresh copy now.

Open up the project in ISE and add the two source files, **basic_and.v** and **basic_and_tb.v**.

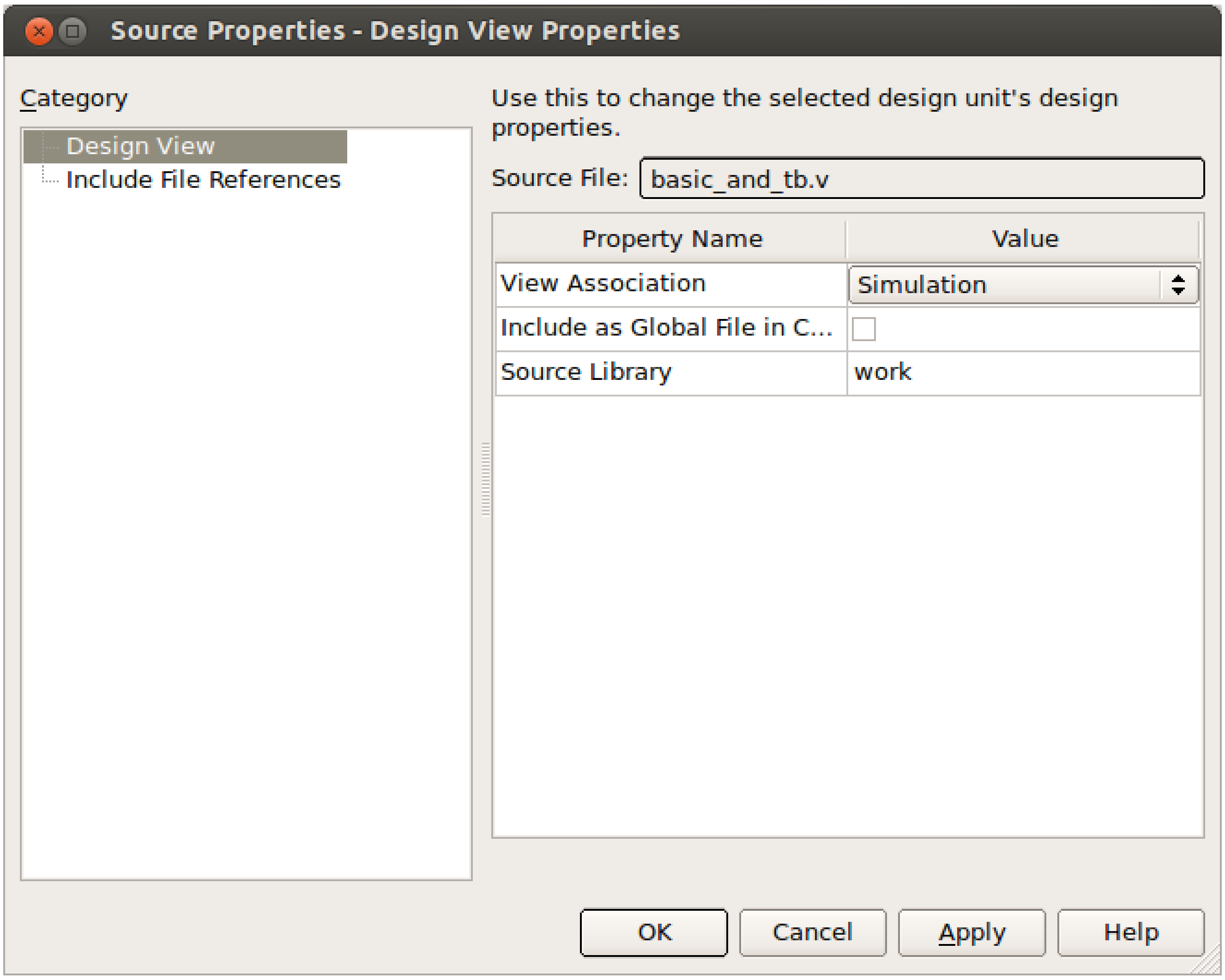
Your project should look something like this.



Mojo-Testbench
xc6slx9-2tqg144
avr_interface (avr_interface.v)
basic_and_tb (basic_and_tb.v)
DUT - basic_and (basic_and.v)
mojo_top (mojo_top.v)
mojo.ucf

If you created the files outside of ISE then **basic_and_tb** may not show up under the implementation tab. This is because ISE realizes that files that end in **_tb** are test benches and will mark them for use in simulation only.

If your test bench is showing up in the implementation tab, as shown above, **right click** on the test bench and choose **Source Properties...**



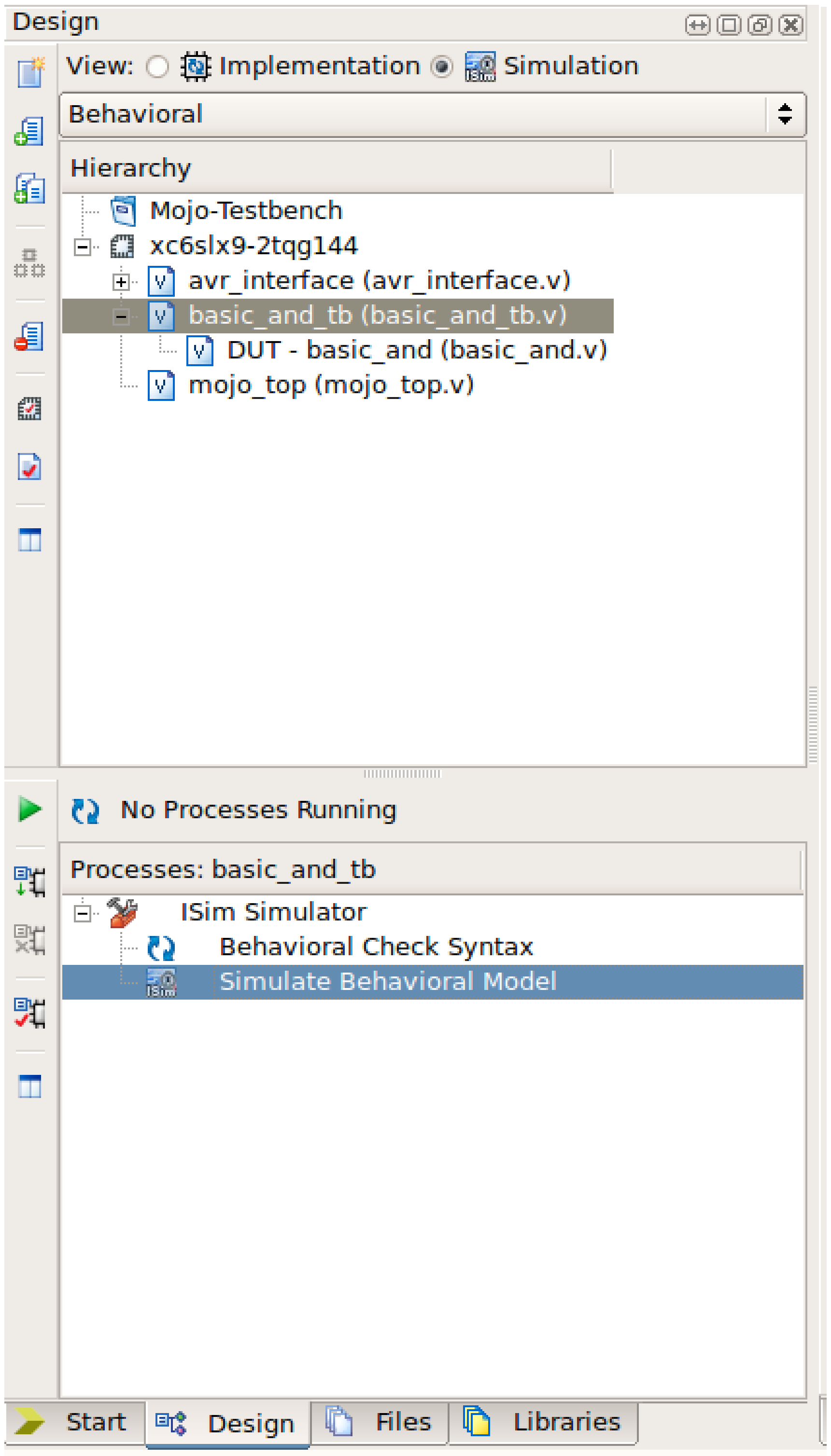
Use this to change the selected design unit's design properties.

Source File: basic_and_tb.v

Property Name	Value
View Association	Simulation
Include as Global File in C...	☐
Source Library	work

Change the **View Association** to **Simulation**.

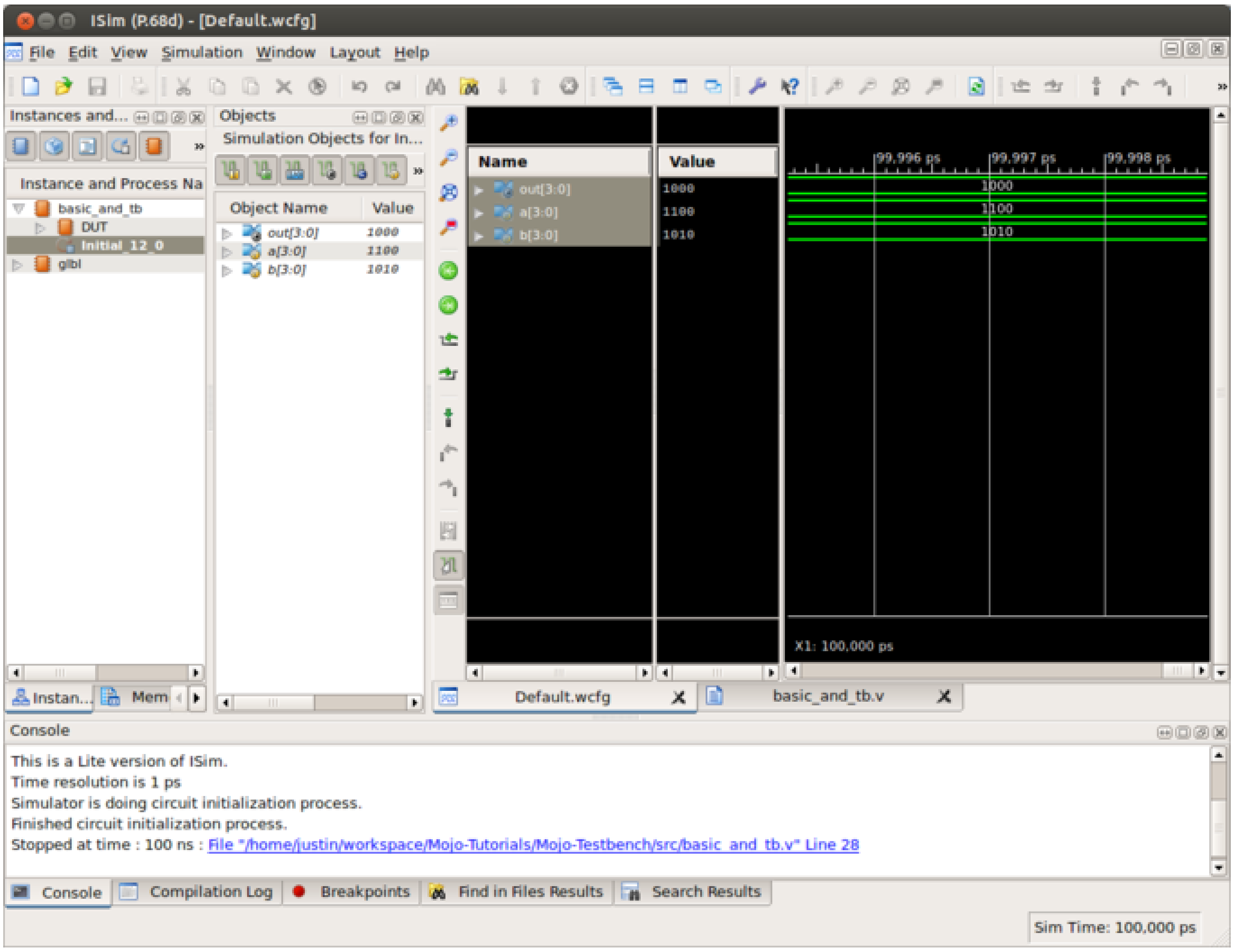
Back in ISE, click on the **Simulation** tab.



Notice the **Simulate Behavioral Model** option now. This is how you launch the simulation. Make sure that **basic_and_tb** is selected and double click **Simulate Behavioral Model**.

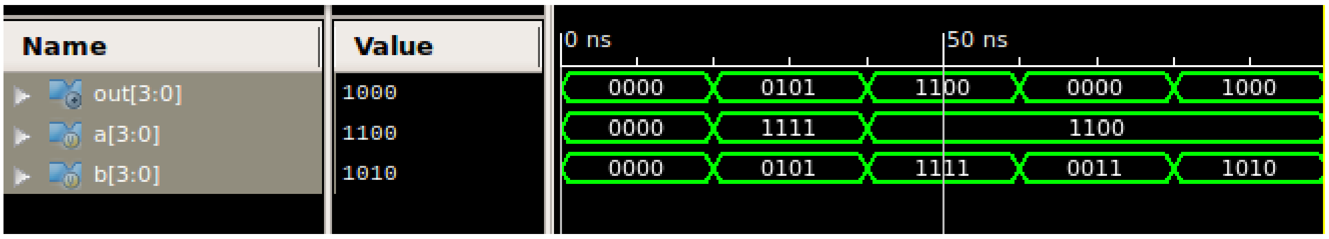
That will launch **ISim**.

It will show your test bench code and show where it hit **$finish;**. Go ahead and click on the tab that says **Default.wcfg**.



```
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Stopped at time : 100 ns : File "/home/justin/workspace/Mojo-Tutorials/Mojo-Testbench/src/basic_and_tb.v" Line 28
```

Click the magnifying glass that looks like it has an X in it. That will fit the simulation to the display. You should then see the following.



Notice how **a** and **b** change. Their values match the ones in our test bench and you can see that they change every 20 time units (ns in this case).

The power of the test bench is now you can look at **out**. If you compare **out** with the corresponding **a** and **b** inputs, you will notice that the module is infact **anding a** and **b** together (1111 **and** 0101 = 0101!).

Sequential Test Benches

The previous example is a great way to get you feet wet with test benches, but the circuit we tested didn't have any sequential logic in it. How do you go about testing a circuit that requires a clock signal?

To show to do this, we will be testing the pwm module from the pulse-width modulation tutorial.

```
module pwm #(parameter CTR_LEN = 8) (
    input clk,
    input rst,
    input [CTR_LEN - 1 : 0] compare,
    output pwm
  );

  reg pwm_d, pwm_q;
  reg [CTR_LEN - 1: 0] ctr_d, ctr_q;

  assign pwm = pwm_q;

  always @(*) begin
    ctr_d = ctr_q + 1'b1;

    if (compare > ctr_q)
    pwm_d = 1'b1;
    else
    pwm_d = 1'b0;
  end

  always @(posedge clk) begin
    if (rst) begin
      ctr_q <= 1'b0;
    end else begin
      ctr_q <= ctr_d;
    end

    pwm_q <= pwm_d;
  end

endmodule
```

To test this we again will need to write a test bench. An example of what that test bench could look like is shown below.

```
module pwm_tb ();

  reg clk, rst;
  reg [7:0] compare;
  wire pwm;

  pwm #(.CTR_LEN(8)) DUT (
    .clk(clk),
    .rst(rst),
    .compare(compare),
    .pwm(pwm)
  );

  initial begin
    clk = 1'b0;
    rst = 1'b1;
    repeat(4) #10 clk = ~clk;
    rst = 1'b0;
    forever #10 clk = ~clk; // generate a clock
  end

  initial begin
    compare = 8'd0; // initial value
    @(negedge rst); // wait for reset
    compare = 8'd128;
    repeat(256) @(posedge clk);
    compare = 8'd30;
    repeat(256) @(posedge clk);
    $finish;
  end

endmodule
```

Again notice there are no inputs or outputs of our test bench.

In this test bench we have to have two initial blocks.

```
initial begin
  clk = 1'b0;
  rst = 1'b1;
  repeat(4) #10 clk = ~clk;
  rst = 1'b0;
  forever #10 clk = ~clk; // generate a clock
end
```

This first block generates the clock and reset signals. You will use basically this exact same initial block for any test bench that is testing a sequential circuit.

The **clk** and **rst** signals are initialized to 0 and 1 respectively.

The next line uses the **repeat** statement.

```
repeat(4) #10 clk = ~clk;
```

This is used to toggle the clock four times with 10 units of time between each toggle.

After the clock is toggled a bit, **rst** is brought low to allow the circuit to resume normal operation.

It is very important to always reset your circuit before expecting any meaningful output. Flip-flops do not have a default value and will output **x** in simulations when they are not reset properly.

```
forever #10 clk = ~clk; // generate a clock
```

The last line of this initial block will generate a clock signal forever! The **forever** keyword is used to create a loop that lasts, you guessed it, forever.

The second initial block is where we do the actual testing.

```
initial begin
  compare = 8'd0; // initial value
  @(negedge rst); // wait for reset
  compare = 8'd128;
  repeat(256) @(posedge clk);
  compare = 8'd30;
  repeat(256) @(posedge clk);
  $finish;
end
```

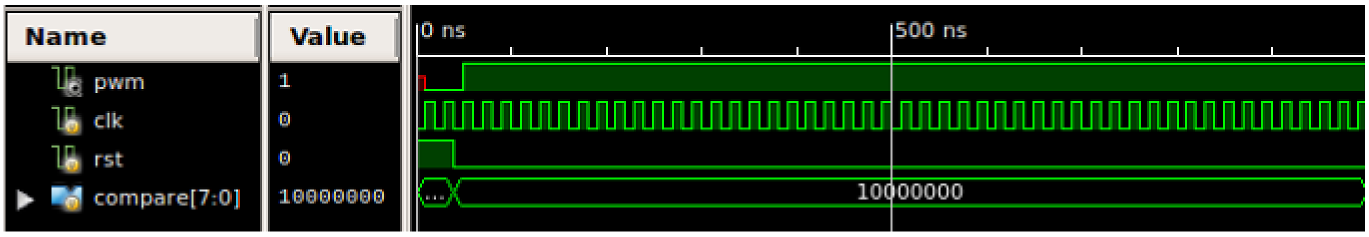
The first part of this block waits for **rst** to fall. This is accomplished with the **@(negedge signal)**; syntax. You can use **negedge** or **posedge** to make your initial block wait for the corresponding edge of that signal.

The signal **compare** is then set to 128 to make sure we get a 50% duty cycle from the **pwm** module. Since each cycle of the module takes 256 clock cycles (because we set **CTR_LEN** = 8), we need to wait for 256 positive edges of the clock.

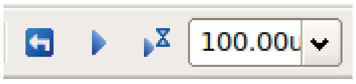
After we wait long enough we can change **compare** to a new value to test the **pwm** module under a different case.

In this example we are only testing two cases, but in most test benches you should try more, especially the edge cases.

If you now add these two files to your project and run the simulation you should see the following.



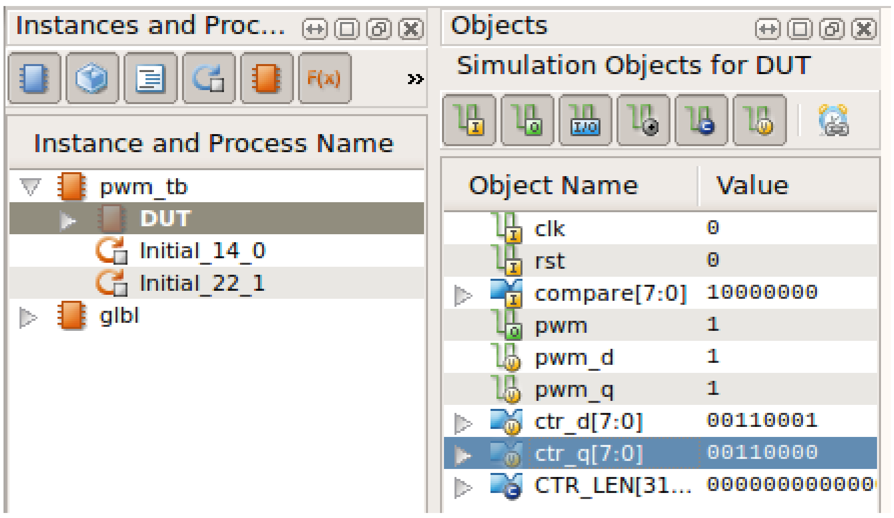
Now you may notice that the simulation stopped after 1us. This is the default value in ISim. To change it set the new value in the toolbar like this.



Set it to 100us just to be safe. Since we have the **$finish;** statement in our test bench it doesn't hurt to set it longer than we need because it will end when it hits that line.

If you now click the arrow with the hour glass it will run the rest of the simulation.

Before looking at the signal we should show the **ctr_q** signal from the **pwm** module. To do this take a look af the signals shown on the left of the display.

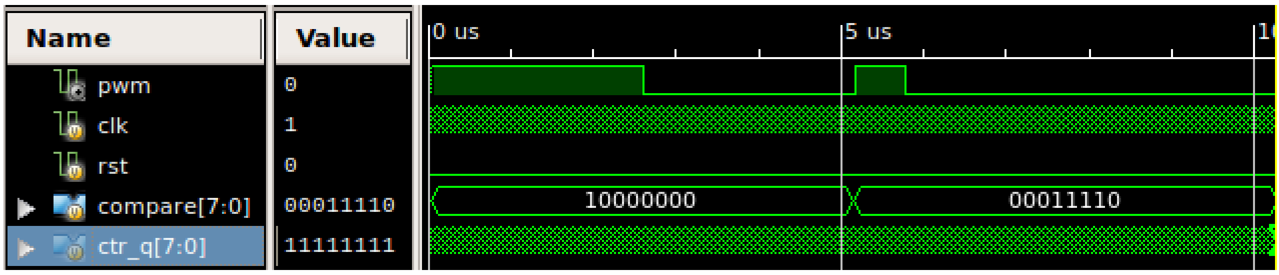


Instance and Process Name
pwm_tb
DUT
Initial_14_0
Initial_22_1
glbl

Object Name	Value
clk	0
rst	0
compare[7:0]	10000000
pwm	1
pwm_d	1
pwm_q	1
ctr_d[7:0]	00110001
ctr_q[7:0]	00110000
CTR_LEN[31...	000000000000

Expand the pwm_tb and select DUT. You can then right click **ctr_q[7:0]** and click **Add to Wave Window**.

To get the waveform updated you need to click the icon in the toolbar that is a blue box with a white arrow pointing to the left (left most icon shown two pictures above). This will rerun the simulation from the beginning.



Name	Value
pwm	0
clk	1
rst	0
compare[7:0]	00011110
ctr_q[7:0]	11111111

Here is what the final simulation should look like. You can see the pulses are exactly as we expected them to be. If you zoom in to any point you will be able to see the value of **ctr_q** and the rising and falling edges of **clk**.

Hopefully this gives you a good starting point for writing your own test benches. Test benches are incredibly important for verifying that your modules are written correctly and everything is working as it should.

Verilog

Like 2 Share Tweet G+ Save

Previous

Next

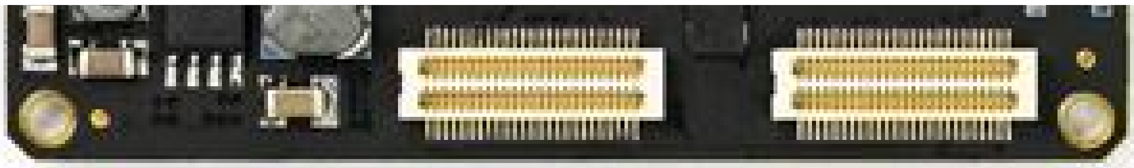
Search

Newsletter

Enter your email address below to join our mailing list and have our latest news and member-only deals delivered straight to your inbox.

Email →

Top Product



Alchitry Au

Alchitry

$109.99 USD

Add to Cart

Links

Home

Products

Blog

Tutorials

Projects

Forum

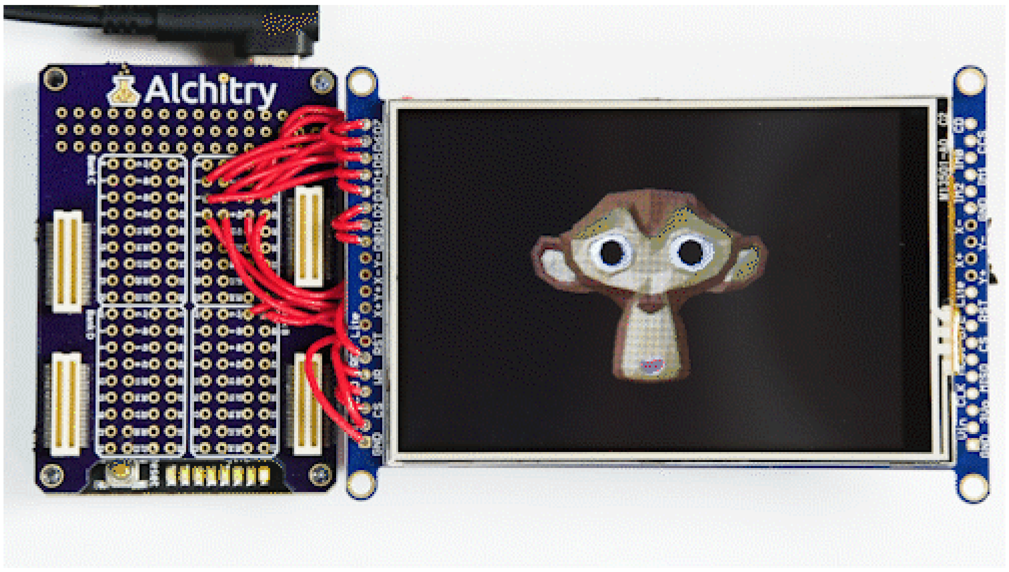
Links

Contact Us

Blog

GPU Demo Project

August 27, 2018 • Justin Rajewski



Learn to make your own GPU! This demo project is running on the new Alchitry Au that is still available from our Kickstarter campaign!

Newsletter

Enter your email address below to join our mailing list and have our latest news and member-only deals delivered straight to your inbox.

Email →

Social Links

Search /