

Digital Design: A Systems Approach

Lab 1: Introduction to Verilog and the Tool Chain

Introduction/Objectives

Due: To be completed during your assigned lab section

Welcome to Lab 1. This lab is designed to familiarize you with the languages and tools we'll be using in throughout the labs, namely:

- **Verilog** (the hardware description language, or HDL, we'll use in this class to **describe** our design)
- **ModelSim** (a program that takes Verilog descriptions of digital circuits and **simulates** them to test for functional accuracy), and
- **Xilinx ISE** (a program which can translate and compile Verilog descriptions of digital circuits so that they may be downloaded and tested in a real-time way on our field-programmable gate arrays or FPGAs; this is also known as **synthesis**).

The lab is a step-by-step walk-through that will take you through the design process from reviewing the initial specifications of a digital circuit and writing Verilog to implement the desired functionality, as well as using ModelSim for simulation and debugging, and finally synthesizing the code on our FPGAs for real-time testing.

Please do not feel you have to understand everything that is going on in this lab. However, you should make sure you read through everything and get a feeling for what is going on. Remember that you can always ask the TAs for an explanation!

What Is Verilog?

Verilog is a hardware description language (HDL), or in other words, a language that can be used to describe digital circuits. It is different from other programming languages that you may have used previously because in Verilog, 'variables', when the design is fully synthesized, become actual wires and operations on those variables become combinational/sequential logic blocks. This leads to the main principle of Verilog that you should seek to understand from the outset – **All assignments in Verilog will be evaluated CONCURRENTLY**. This is a very important concept to understand and confuses many students who are just starting out, especially if they have previous programming experience.

What do we mean by that? For instance, in C or C++ you may see something like:

```
E = C + D;  
C = A + B;
```

If you saw that in C and C++ you would say, "Ah! I know what's going on: E takes on the sum of variables C and D. Then, a new value of C is computed using A and B." In C or C++, which are

sequential programming languages, you'd be right. If it was Verilog, and it was written in this way:

```
assign E = C + D;  
assign C = A + B;
```

the previous reasoning would be wrong. All `assign` statements in Verilog happen at the same time, so in the above example, C is always taking the value of the sum of A and B. It's this sum that is then added to D to produce E. So the order of the statements doesn't matter because you're describing the behavior of actual wires that are always being evaluated – they have no concept of sequence. (This is for the most part. We'll refine this later when we discuss *testbenches*.)

What is an FPGA?

In this class, we will use a field-programmable gate array (FPGA) to test our designs on a logical level. That is to say, “Does the circuit, as we've specified and coded it, behave as it should?” We won't worry about will the circuit actually work if it were to finally implement as a stand-alone, discrete chip (to do that, one would need to understand issues beyond the scope of this course). We'll just make sure that we've created a design that will function if given ideal hardware, i.e., there are no systematic errors in the design.

Our FPGA is exactly what its name suggests: it is an array of cells (sometimes called *slices*) that are basically programmable gates that can be connected in an arbitrary manner (with certain restrictions) to simulate arbitrary circuits. So, for instance, one cell/slice of the FPGA could ADD two three-bit numbers or it could AND two three-bit numbers. Some cells can also function as little bits of memory or drive input and output pins of the FPGA itself and therefore communicate to the outside world— their behavior depends on how you program them. In fact, typically there are a great number of functions that one cell can implement. As we dive more into the architecture of the FPGA in this lab, it will become clearer on how this “programmable” hardware is realized.

Although it may seem initially that FPGAs serve only research purposes (i.e. for simply testing logic before a tape out), they are sometimes put into marketable products as is because the bring-up time and expense of development is much less than trying to develop an ASIC for that purpose because not only do you need to lay all the transistors, you need to electrically verify functionality, a rather involved and sometimes expensive process. While it can cost well in excess of \$100,000 to tape out an ASIC, FPGAs are freely reprogrammable (allowing for continual updates), and low cost. Additionally, development time is much lower on an FPGA, and development is much simpler. So in many ways, what you will be doing in this class can be directly applicable to industrial applications¹.

¹ In particular, in industry, FPGAs are used mostly for implementing DSP (digital signal processing) functionality in embedded systems for these reasons.

Step 1: Design of a Simple Circuit

Since you haven't been exposed to much digital design yet in the course we're going to start out with a very simple lab. It will use the switches on the FPGA board as two 4-bit inputs (A and B), and perform simple operations on them. We will output the result to 4 LEDs on the FPGA board. We'll start out by implementing two functions: $A \& B$ and $A + B$ (where $+$ means addition). Later in the course, we'll move on to more complicated logic.

Specifying the design

To begin our design, we always start with a specification of the desired behavior. For our example:

- When the Left pushbutton is pressed, the output LEDs should show switch inputs A ANDed with switch inputs B.
- When the Right pushbutton is pressed, the output LEDs should show switch inputs A added to switch inputs B.
- In this lab, pressing both buttons yields the same result as only pressing the right pushbutton.
- If neither button is pressed, the LEDs should not turn on.

Creating A Block Diagram Description

From the description given above, we can define the inputs and outputs to the system. These are the connections that come into the design from the real world and go out to show the results. **Figure 1** describes the top-level block diagram for this lab. Notice that in it, all inputs and outputs are shown, but the block doesn't attempt to describe in any way the inner workings of lab1. We can think of this block diagram as describing a black box that has inputs and outputs.

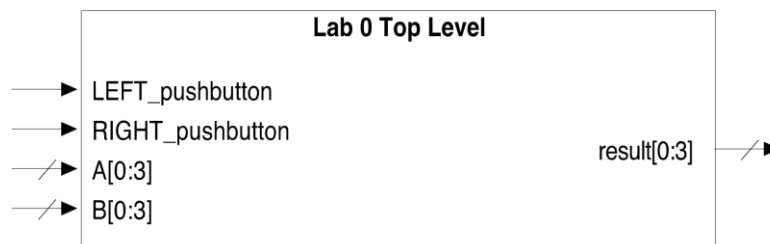


Figure 1: Top level block diagram of lab1.

We describe this block diagram, or module, in Verilog using the following code:

```
module lab1_top (  
    input LEFT_pushbutton,  
    input RIGHT_pushbutton,  
    input [3:0] A,  
    input [3:0] B,  
    output reg [3:0] result  
);  
  
    // Fill in logic here...  
  
endmodule
```

The above is generally referred to as a **module declaration**. Notice we define our block with the reserve words, `module` and `endmodule`. `lab1_top` is the name we chose to give to this particular block. You can think of modules as being the hardware equivalent of software functions and the above code represents that function definition (although this is only to get you going and is a poor relation).

We see we have defined the inputs and outputs. Further notice that we've also decided to declare the output, `result`, as a `reg` (as opposed to a `wire` – we'll get to the difference between the two later). Also note that we've defined it as a 4-bit variable with `[3:0]` which means the most significant bit (MSB), i.e. the left-most bit, is labeled as the third bit and the least-significant bit (LSB) is labeled as the zeroth bit. We could have similarly declared the wire as `[0:3]` which means the exact opposite (MSB is the zeroth bit, LSB is the third bit), but **for this class, we'll always have our LSB at 0**. Regardless, we might think about `result` as being an array of bits, with the first bit having an index of zero. We can operate on the individual bits or the entire variable and we'll see that below. Also note, that if you leave out this bit-depth specification, it is assumed that the signal you're trying to define is only one bit (as `LEFT_pushbutton` and `RIGHT_pushbutton` have been).

Describing the Internal Logic

Now the specification is a simple enough that we can easily figure out what logic goes between the inputs and the outputs. We know we need to calculate two functions: AND and ADD (addition). These are simple functions in Verilog because they are built into the language and thus require only one line each. As such, it's easy to use `wires` and `assign` statements (again – don't worry if you don't understand what all that means, just understand that that's what you're doing). In Verilog you define the meaning of `wires` using the `assign` statement. So anything on the left side of the `=` in an `assign` statement must be defined as `wire` and despite what occurs elsewhere in the circuit, this assignment is being made continuously regardless of *where* it is in the module.

The functionality of this black box, which we'll now start calling a **module** (in line with Verilog vernacular) was specified earlier and is more clearly described in **Table 1**.

Signal	Direction	Width	Purpose
LEFT_pushbutton	Input	1 bit true = pushed false = not pushed	When pushed we show A anded with B
RIGHT_pushbutton	Input	1 bit true = pushed false = not pushed	When pushed we show A plus B
A	Input	4 bits (binary 0-15)	This is our A input from the first 4 switches on the board (1-4)
B	Input	4 bits (binary 0-15)	This is our B input from the second 4 switches on the board (5-8)
Result	Output	4 bits (binary 0-15)	Our output, which is either A & B or A + B according to the button(s) we press

Table 1: Functional description of the inputs and outputs of the “Lab1 Top Level” block.

Figure 2 draws the gate-level diagram of the design we want to implement into the block diagram. (Note: a MUX is a device which selects an input based on the control signals, here LEFT_pushbutton and RIGHT_pushbutton).

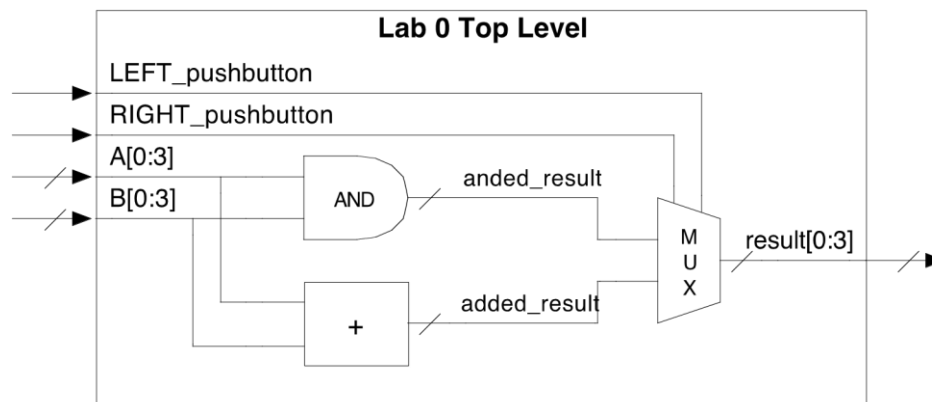


Figure 2: Block diagram of lab1 top level with logic drawn into it. (Note this is for sake of example – in design descriptions you typically don’t draw in the logic to the block diagram.)

Notice that we’ve also labeled all the lines (which represent physical wires that will connect our gates. Remember when we said that ‘variables’ (wires or regs) become actual wires? This is where it happens. Much like other programming languages, **you must first declare a wire/reg before you can use it** (this is simply for the compiler), so we’ll start out by making defining those intermediate wires with the following code:

```

wire [3:0] anded_result;
wire [3:0] added_result;

```

(Note that these are terrible choices for signal names because they differ in only one character.)

How do we actually define the functions? Well, since these are very simple functions and we’ve defined

anded_result and added_result as wires, we can do them directly with assign statements:

```
assign anded_result = A & B;
assign added_result = A + B;
```

Be careful with & and && in Verilog. The single ampersand does a bit-wise operation (i.e., it will AND each bit with the corresponding bit in the other input) the double ampersand is a logical AND which will return a single bit true if both the inputs are true.

Now we've written the logic for the two intermediate results and we just need to make a MUX to select which one we want based on the button presses.

Since this is a bit more complicated we'll use an always block for this. **Verilog requires that any logic defined in an always block have its results stored in regs**, so we need to define the output of our always block as a reg as we've done above. The code below then describes the logical behavior of our circuit (NOTE: the ' in 2'b01 is an apostrophe, not a tick located on the "~" key. Be wary if you plan on cutting and pasting – you might have issues with this).

```
always @* begin
    case( {LEFT_pushbutton, RIGHT_pushbutton} )
        2'b01: result = added_result;
        2'b10: result = anded_result;
        2'b11: result = added_result; // Right push button takes precedence
    endcase
end
```

An always block is one construct in Verilog. As you can see, it's enclosed by a begin and an end and in one sense, it defines a scope where one can assign values to regs. There other blocks that do so as well, but we'll learn about them later, The @* part tells the simulator (ModelSim) when to simulate the logic contained between the begin and end. Often times you may see something like @(LEFT_pushbutton) which tells ModelSim to start evaluating the logic contained in the always block when the LEFT_pushbutton signal changes. The @(LEFT_pushbutton) or @* is known as the *sensitivity list* for this always block.

In the code above, we've used the wildcard * in the sensitivity list to tell the simulator that any time ANY of the signals in this block (including inputs) change, re-evaluate the logic contained in this always block (i.e. between the begin and end). It may seem unnecessary because, why might you ever want NOT to evaluate that logic when any signal changes? If you consider that even in this class, designs can get very large and begin to take incredibly long times to simulate, you may see why it's helpful if the designer can specify when the logic changes the outputs of the module to help the simulator save computational/run time. **Nowadays, since simulators are more advanced and computers are fast, you should *always* use * in your Verilog sensitivity lists.**

We could have placed the assign statements for anded_result and added_result either before or after this always block, again, because the assignment is always happening. Try and keep focusing on the fact that your hooking up pieces of hardware. Logic gates have no idea about order. All they understand is they're given voltages at their inputs which actual transistors use to produce an output.

As for the `case` statement, you can think of that as the hardware equivalent to a `switch` statement in C or C++ and again, it's Verilog shorthand (you could have easily encoded those assignments using logical `&`'s and `|`'s, but it's a lot simpler to specify it that way). Here `{ }` are used to sandwich signals together in Verilog (here, taking the 1-bit variables `LEFT_pushbutton` and `RIGHT_pushbutton` and creating a two-bit variable where `LEFT_pushbutton` is the MSB and `RIGHT_pushbutton` is the LSB). The value of this *concatenated* signal then determines how `result` is computed.

Also note that we've encoded precedence here because our Verilog describes a circuit that will compute the added result of A and B if both pushbuttons are pressed.

Note about “Inferred Latches”

For those of you who paid attention in section the previous code should look a bit problematic. In particular there is something missing which was heavily emphasized during the Verilog introduction. This is intentional, as we want to show you how sneaky inferred latches can be. We will show you the missing part later on in this lab.

Notice that we've defined `result` as both an output and a reg. This means that it is assigned in an `always` block and then is also sent out of our `module`. If we had used an `assign` statement for the result we would have had to define it as a `wire` since `assign` statements always define wires.

For example, we could have done:

```
wire [3:0] result;
assign result = (RIGHT_pushbutton) ? added_result :
               (LEFT_pushbutton) ? anded_result : 4'b0;
```

This does *roughly* the same thing as the above `always` block, but is a lot harder to understand! It uses the ternary operator, which is similar to the one found in C and C++ but it is slightly different, and it's also slightly different that the logic described with our `always` block and `case` statement, but that's intentional as we'll find out what that difference does later in the lab. You should use the `always` block version for this lab.

So that's it. We've got all the code for our lab1 top module, now we just need to fire up ModelSim, write it, and see how it works by simulating it.

Step 2: Entering your design in ModelSim

The following will guide you through a step-by-step procedure to open the design tools we will use. Note that the instructions below assume that you are sitting at a lab station. This doesn't have to be the case as you're more than capable of logging in to CSIL/ECI and running modelsim remotely via an SSH tunnel. **This will be described in another upload on Gauchospace.** What follows is a collection of screenshots from a linux session of modelsim, but all the menus and commands are identical on Windows.

Logging in and setting up your environment

Now, we'll open ModelSim. Open the start menu, and search for "Modelsim." Feel free to create a desktop shortcut, as you will be using it frequently.

Once ModelSim opens, create a new project for lab1 by going to File -> New -> Project...

The project name is "lab1" and the location should be in the lab1 folder you just created. Keep the library name as work (this is the name of the folder inside your lab1 folder where ModelSim will keep all its data) and click OK (**Figure 5**).

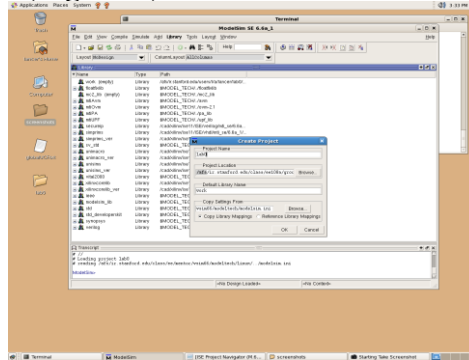


Figure 5

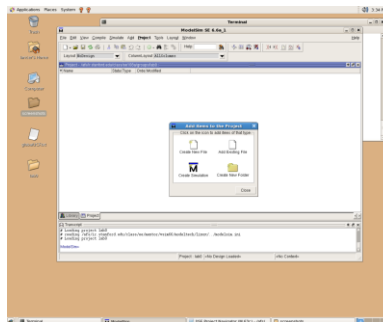


Figure 6

Close the "Add items to the Project" window that appears (**Figure 6**).

You'll now have a ModelSim project window with the Workspace pane on the left set to the Project tab and the Transcript pane on the bottom (**Figure 7**). If you don't see the Workspace pane select it from the "View" menu.

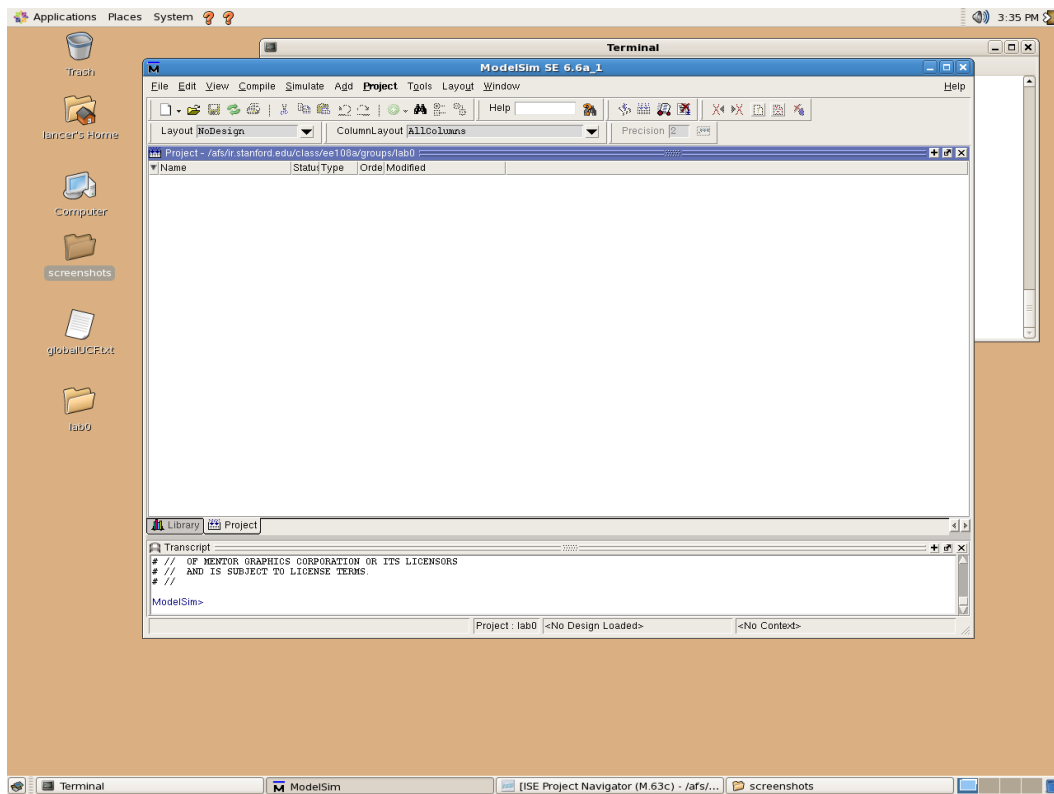


Figure 7

The next step is to create our Verilog files. At the moment we have one module (lab1_top) so we'll only need one file. Make sure the Workspace Pane is set to the Project tab then right click and choose "Add to Project -> New File...".

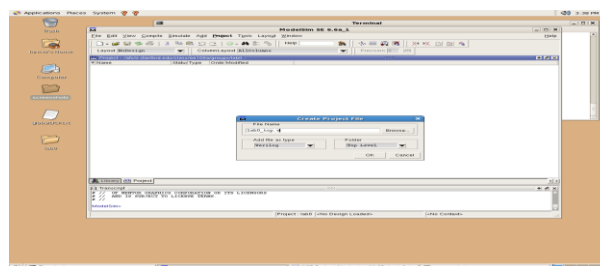


Figure 8

Name the file "lab1_top.v" and make sure you select the type as Verilog and NOT VHDL (**Figure 8**). Your file will now appear in the Project tab with a question mark indicating it hasn't been compiled yet (**Figure 9**).

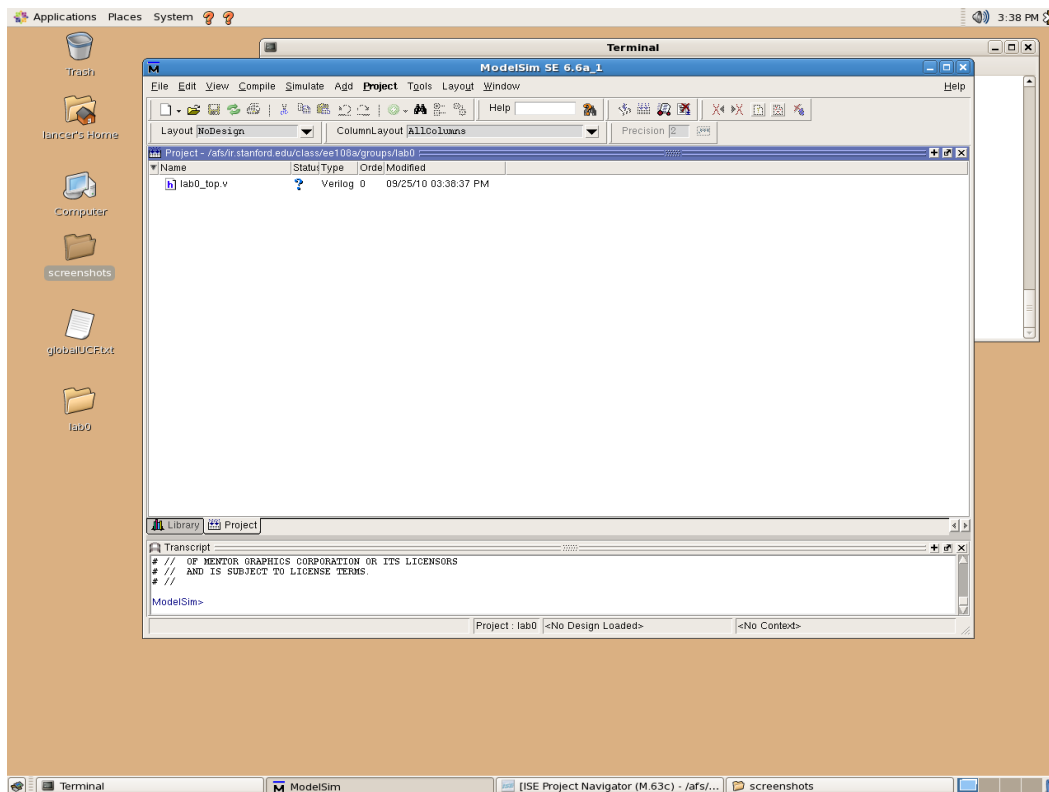


Figure 9

Double click on your file to edit it. It will open in a new editor tab on the right. Now enter the Verilog code for your module from above. Careful if you're cutting and pasting! Often times the ' symbol is mis-translated between a .pdf viewer and ModelSim's text editor.

Save the file and compile it by right-clicking in the Project Pane and choosing Compile -> Compile Out-of-Date (**Figure 10**). This will automatically re-compile any files you have changed since the last compilation.

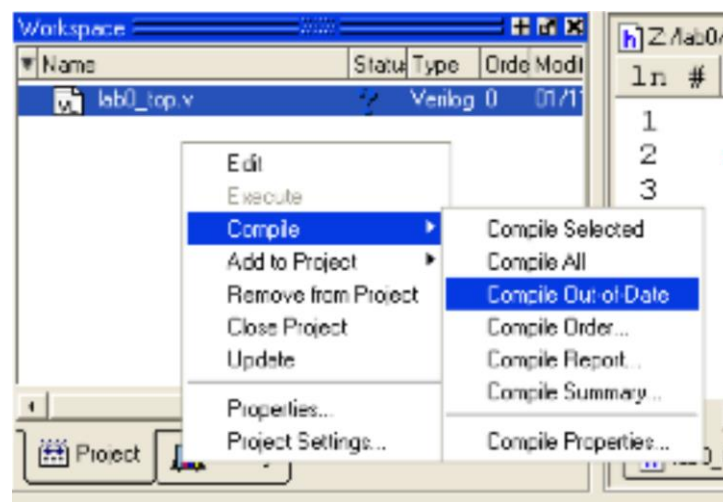


Figure 10

If it's compiled you're all set, if not it will tell you there is an error in the Transcript window

Double-click on the error line and a window will pop-up with usually a fairly useless error message. It will, however, tell you the line number where the error was found and then it's up to you to figure out how to fix it.

If your compile is successful, you will see the following (**Figure 12**):

```
|| # Compile of lab0_top.v was successful.
```

Figure 12

And a pretty green check mark (**Figure 13**):



Name	Status	Type	Order	Modified
 lab0_top.v		Verilog	0	01/11

Figure 13

Now the file is compiled for the simulator, but we can't actually do anything with it.

Writing a Test Bench

The reason we can't use our wonderful new Verilog module is that we haven't specified our inputs anywhere; we've only defined how they are used. In order to simulate our module we need to generate those inputs. This is what a "test bench" is for. A "test bench" is a Verilog module which generates the inputs for another Verilog module so we can test the second module. The module we are testing (in our case lab1_top) is called the "device under test", or dut.

The difference between testbenches and regular **synthesizable** Verilog modules is that testbenches often have code which departs from regular Verilog, in that they contain code blocks which are specifically run in the sequence the code was found as opposed to all happening in parallel. As an example, an `initial` block defines code that should be run in sequence. `for`, `repeat`, and `while` blocks also cause the simulator to execute the code in the order it is found. And, as you may notice later, there are other commands that couldn't possibly make sense to implement in an actual chip, like `$display()` for printing output to the screen, or `$stop` which tell the simulator to stop its simulation. These extra commands are almost solely to help you to test your design, and that's really important.

Do you remember how much time it took to compile your lab1_top module just now? Nope? Missed it because it was so fast? Well, if you wanted to actually compile just that module for the real FPGA it would take about 5 minutes every time you changed anything. Time is one of the two reasons we're

going to spend so much time testing our modules before we run them on the FPGAs. The other reason is visibility. When we simulate our modules in ModelSim we can look at any signal at any time during the simulation. When the module is running on the FPGA we can't. If there's a bug somewhere you're a lot more likely to find it if you can watch what your design is doing at a human-readable speed than trying to catch that one error that happens for 10 nanoseconds on the FPGA.

The trick to writing test benches is that there are usually too many possible things to test. Imagine we wanted to test if our add statement is correct. How many possible 4-bit numbers can we add together? Well, 2^4 is 16, and we have two inputs, A and B, so we have $16 \times 16 = 256$ possible additions. Sure we could test 256, but imagine if we had 2 8-bit numbers to add together. Now that's $2^8 = 256$, and we would have two inputs, so we would have $256 \times 256 = 65,536$ possibilities. Again, no problem. Maybe a few minutes of time on the computer. Here's the kicker: the adder in my laptop takes in two 64-bit values. There are 1.84×10^{19} possible choices in a 64-bit value, and with two of them that's 3.4×10^{38} possibilities. If we had a computer that could test 100 billion possibilities a second (roughly 10 times faster than any computer today) that would only, oh, take 1×10^{20} years to finish. (That's roughly 7.8 million times the age of the universe.) So it's fairly unlikely that Intel actually tested all the possibilities.

As an aside: if someone ever tells you that something won't work because an algorithm or problem runs in exponential time, this is exactly what they mean. As soon as you get a tiny bit bigger (64 bits is only 16 times as many as 4) the problem becomes absurdly impossible.

So what do we do? Well, we rely on our understanding of the circuit to figure out what's important to check. In our case we clearly want to check that when we press the buttons we get the AND or the addition of the values, and we should try a few values to make sure they're okay. We could even do a simple loop to test a whole bunch of values, but chances are that if our logic adds 5 and 6 correctly, then it's probably going to add 5 and 7 correctly since it's pretty simple.

If you want to know just how valuable it is to be able to write test benches just go to Intel or AMD's website and search for positions for design-for-test engineers. You'll find that they need to hire a lot more people to make their designs testable than anything else.

So back to testing our design. Let's make our Test Bench.

Right-click in the Projects tab and choose Add to Project -> New File... (**Figure 14**)

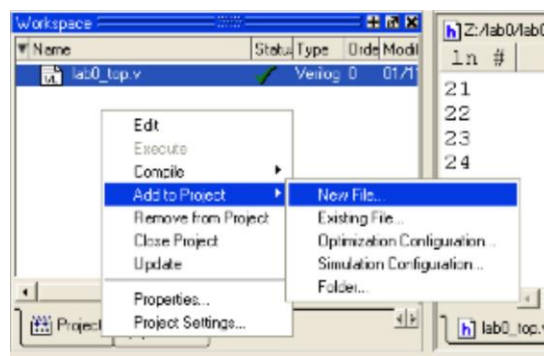


Figure 14

Create a new Verilog file called lab1_top_tb.v. (Remember to make sure you select Verilog as the type.)

In general you'll have one test bench file for every module you write. It's a lot easier to test the individual modules than to try and test a big complicated system, so **we'll force you to start at the bottom and write individual test benches for each module in the labs**. (Trust me: if you do this carefully it will save you an amazing amount of time with lab 3, 4, 5, and the final project.)

Double-click on the file in the Project tab and let's get started.

Just like any other Verilog module, a Test Bench is a module, but chances are it doesn't have any inputs or outputs because all it does is *instantiate* (or tell the simulator to place a copy of) the module you've just written. (You'd need another Test Bench for the Test Bench if it needed inputs itself. Sometimes this happens on much larger projects, though, but don't worry about it now.)

So let's start out by declaring our own testbench module:

```
module lab1_top_tb ();  
    // No inputs or outputs, because it is a testbench  
  
endmodule
```

So far pretty easy – nothing you haven't seen before. Now we need to add/declare the signals we're going to need to send into the device under test, which is our `lab1_top` module.

```
reg sim_LEFT_button;  
reg sim_RIGHT_button;  
reg [3:0] sim_A;  
reg [3:0] sim_B;  
  
wire [3:0] sim_result;
```

Note that the signals that are going to go into the device under test are `regs` since we are going to define them in a combinational block and the signals coming out are `wires`. Signals coming out of modules are always `wires`. You'll get an error if you try to use anything else.

Next up we need to *instantiate* our device under test in our test bench and hook it up. When we instantiate a module inside another module we are telling Verilog to build a copy of the sub-module inside the new module. To do this we need to tell Verilog how to hook up all the inputs and outputs to the sub-module.

So here is our instantiation of a copy of our `lab1_top`, which we are going to call `dut` for Device Under Test. (You can call it anything you want in general, but your names should make sense.)

```
lab1_top dut (
    .LEFT_pushbutton(sim_LEFT_button),
    .RIGHT_pushbutton(sim_RIGHT_button),
    .A(sim_A),
    .B(sim_B),
    .result(sim_result)
);
```

Note how we said that the instantiation is of the module `lab1_top`, and then we named it `dut`. This is because even with the same module, we can instantiate multiple copies of `lab1_top` but each one should have a unique name so that both you and the simulator can tell the separate instances apart.

The next bit of code connects every input or output from the `lab1_top` module. Here, we've used a **dot notation** syntax which we suggest you use for ALL of the labs: first define to what port your going to hook up signal, then define what signal should be hooked up to that port, as in the manner above. We can very clearly see that `sim_LEFT_button` is going into the `LEFT_pushbutton` input of the `lab1_top` instantiation. You could also use the following (again, **NOT suggested**)

```
lab1_top dut (
    sim_LEFT_button,
    sim_RIGHT_button,
    sim_A,
    sim_B,
    result
);
```

In the above, the simulator automatically assumes you've hooked them up in the order found in the module declaration. This is bad because as you could imagine, during debugging you may need to add inputs or outputs and if you use the second syntax, you'll have to hunt through all your code and make sure the order (which you may have changed) is still correct for ALL instantiations. The former syntax saves you all that work.

If you forget to hook up one, you'll get a warning, and that's probably a mistake. (Also watch out for mistakes like confusing “,” and “.”)

I want to re-iterate something here for those of you with programming experience. Instantiation in Verilog means *making a copy* of a module. If you instantiate a module 5 times you will have *5 separate copies* of your logic. This is completely different from Java or C. In either of those languages if you have 5 function calls you run the same code 5 times, one after the other. In Verilog, instantiating 5 copies of a module means you have *5 copies existing in parallel at the same time, taking up five times the space on your FPGA than just one module*.

So now our test bench defines the signals going into our Device Under Test (our `lab1_top` module) and instantiates one copy of it. Next we need to define what those input signals should do so we can see what the output is.

To do this we'll define an `initial` block in Verilog. **initial blocks can only be used in testbenches, that is, they are not synthesizable into logic.** When your simulation starts, Verilog will

process, or sequentially step through all the initial blocks. You can have multiple initial blocks in your test benches, but make sure you don't try to set the same signal from multiple places or you'll get very confused.

So here's what we'll use to start with:

```

initial begin
    // start out by setting our buttons to "not-pushed"
    sim_LEFT_button = 1'b0;
    sim_RIGHT_button = 1'b0;

    // start out with our inputs both being 0s.
    sim_A = 4'b0;
    sim_B = 4'b0;

    // wait five simulation timesteps to allow those changes to happen
    #5;

    // Our first test: try ANDing
    sim_LEFT_button = 1'b1;
    sim_A = 4'b1100;
    sim_B = 4'b1010;

    // again, wait five timesteps to allow changes to occur
    #5;

    // print the current values to the Modelsim command line
    $display("Output is %b, we expected %b", sim_result, (4'b1100 &
4'b1010));
    $stop;
end

```

The `initial` block is another Verilog construct used only in simulation. It departs from normal Verilog behavior in that the lines contained within the `begin` and `end` are evaluated by the simulator not all at once, but rather one at a time and in the order found. So, in the above, `sim_LEFT_button` is assigned the one-bit value of 0 (that's what `1'b0` means) before `sim_RIGHT_button`. Then we assign `sim_A` and `sim_B` values of 0 and in that order (but this time 4-bit of zeros).

Then we delay and change our inputs. The delay (`#`) tells the simulator to run for some number of time steps before the next statement. The `$stop` command tells the simulator to stop, and the `$display()` tells the simulator to print out some text to the command line. In this case we're going to print out what we get (`result`) and what we expect (`4'b1100 & 4'b1010`).

Let's try running the test bench.

Right-click in the project tab and choose Compile -> Compile Out-of-Date. (**Figure 15**)

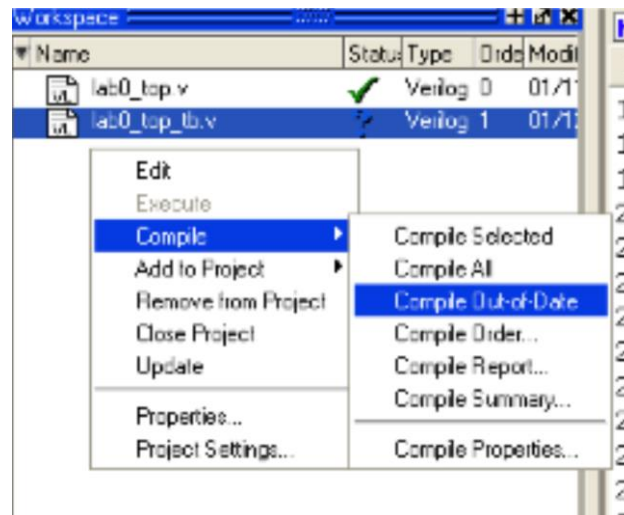


Figure 15

If everything compiles, we now want to run our simulation and view the results. Choose Simulate -> Start Simulation, then under the Design tab expand “work” (remember when we created the ModelSim project we told it to use “work”) and select your lab1_top_tb as the top module (**Figure 16**). Also **uncheck the Enable optimization option!** If you forget to uncheck this, ModelSim will make your design run faster by collapsing all your logic, which will mean you won’t be able to see what any of it is doing. Click OK.

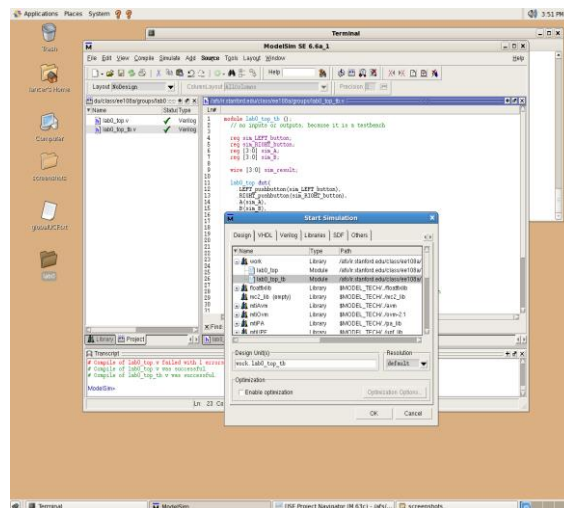


Figure 16

Note that the Project tab has been replaced with the sim tab. The sim tab (**Figure 17**) shows you the hierarchy of your design. Here we just see that we have one module in lab1_top_tb, and that it is called “dut”.

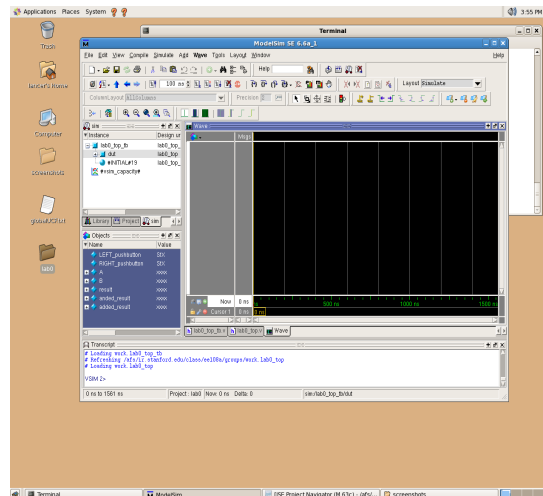


Figure 17

From the View menu choose View -> Wave. I like to arrange my screen with the Objects pane below the Workspace pane as you can see (**Figure 18**). Note, if you have a processes pane open, you can go ahead and click the “x” to remove that – we won’t be using it here.

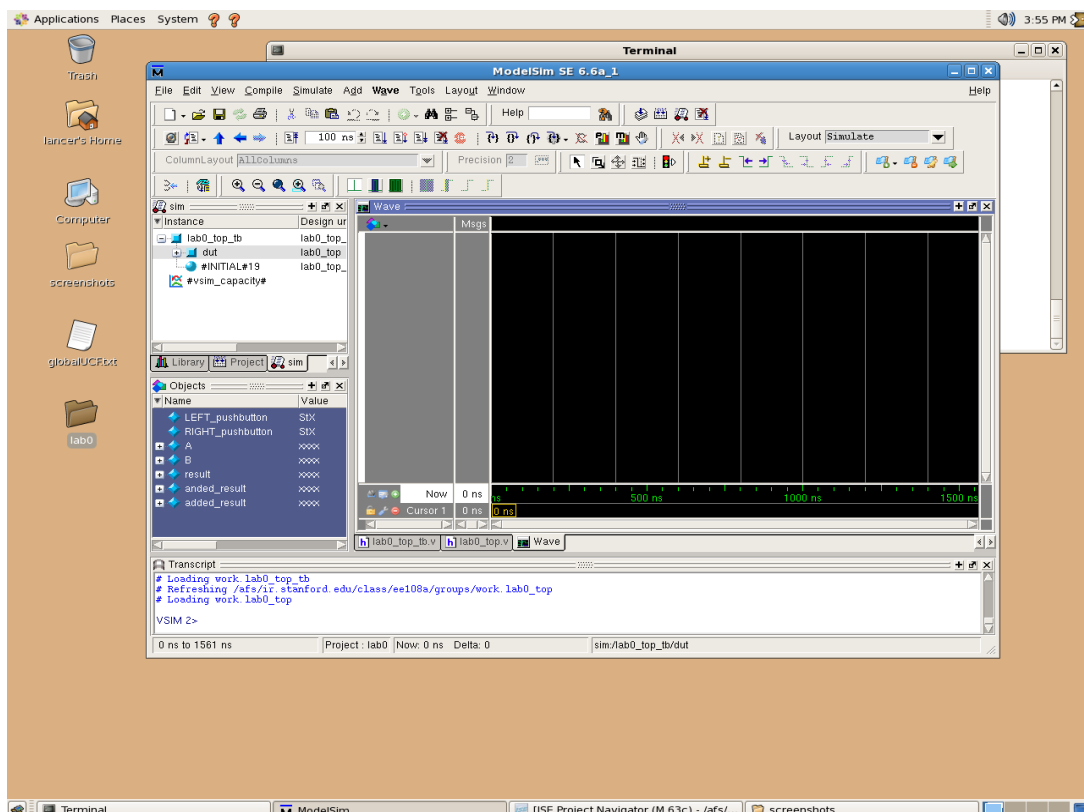


Figure 18

Now we want to watch what’s going on in our simulation, so we will add objects from the Objects pane to the Wave pane.

Note that the Objects pane shows the signals in the module selected in the sim pane. I.e., if I have the top level selected I see the signals at the top level (**Figure 19**). If I have the `dut` module selected I see the signals inside the `dut` module (**Figure 20**).

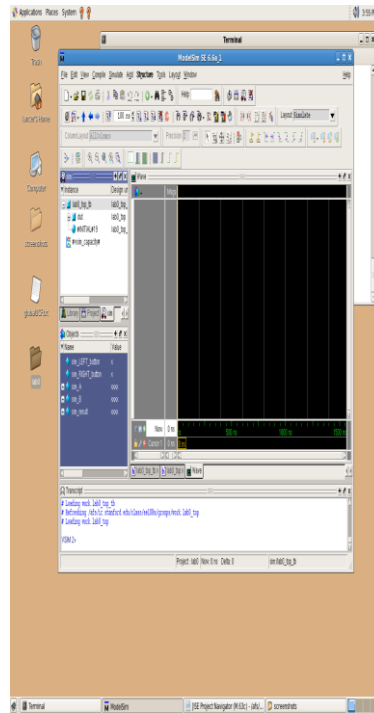


Figure 19

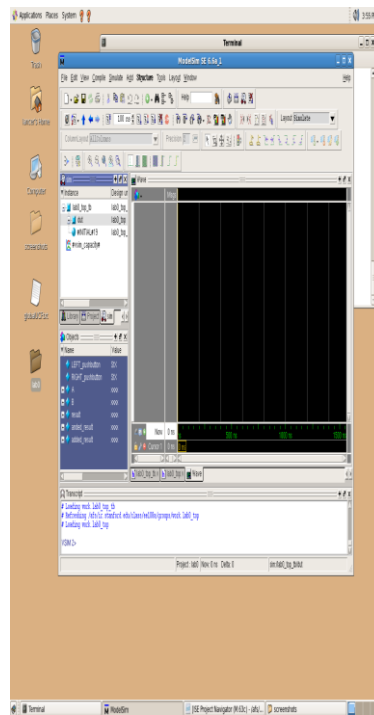
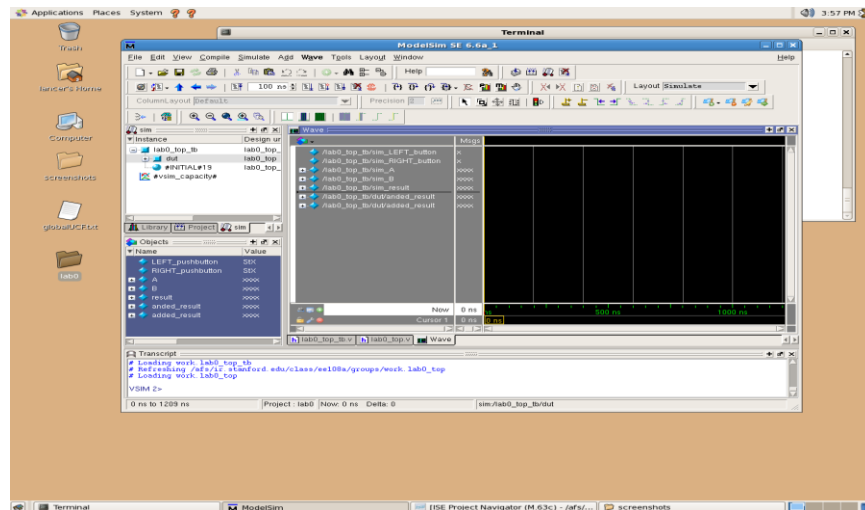


Figure 20

For this simulation we want to look at all the signals in the top-level `lab1_top_tb` module, and the intermediate signals `anded_result` and `added_result` in the `dut` module. Select those signals and drag them to the Wave pane to add them.

They should now show up in your Wave viewer (**Figure 21**).

**Figure 21**

Note that the wave viewer shows a graph of signals vs. time. As we simulate it will fill in the signals. Our test bench will only simulate for 10 time steps (we have two delays of 5 timesteps each), but later on we'll get much longer test benches!

So now let's run the simulation. Choose **Simulate → Run → Run-all**

Your simulation will run until it gets to the `$stop` we inserted, and you should see the output (**Figure 22**):

```
VSIM 3> run -all
# Output is 1000, we expected 1000
# Break in Module lab0_top_tb at Z:/lab0/lab0_top_tb.v line 42
```

Figure 22

Go to the Waves pane and let's see what we got (**Figure 23**):

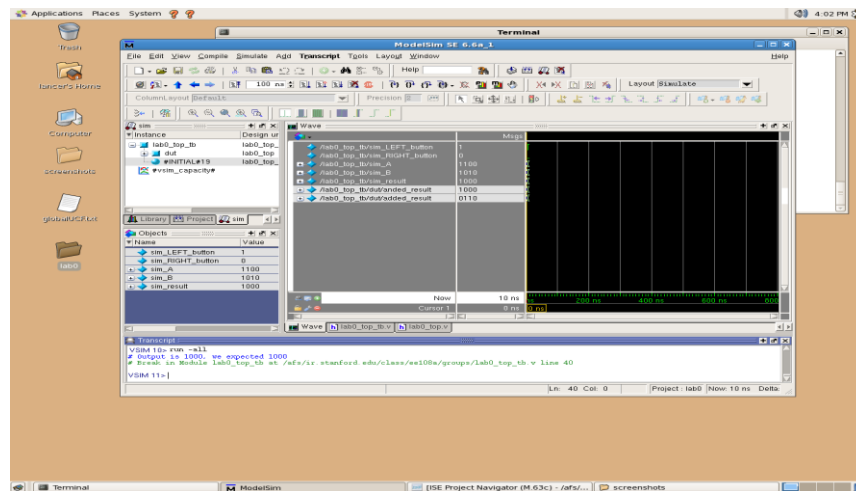


Figure 23

Kind of hard to see, hm? Well, just zoom in on it by clicking in the wave pane, and typing + (using the shift key as well) (**Figure 24**). Or, sometimes I prefer to use “Zoom Full” and then “Zoom In on Active Cursor” to magnify the details.

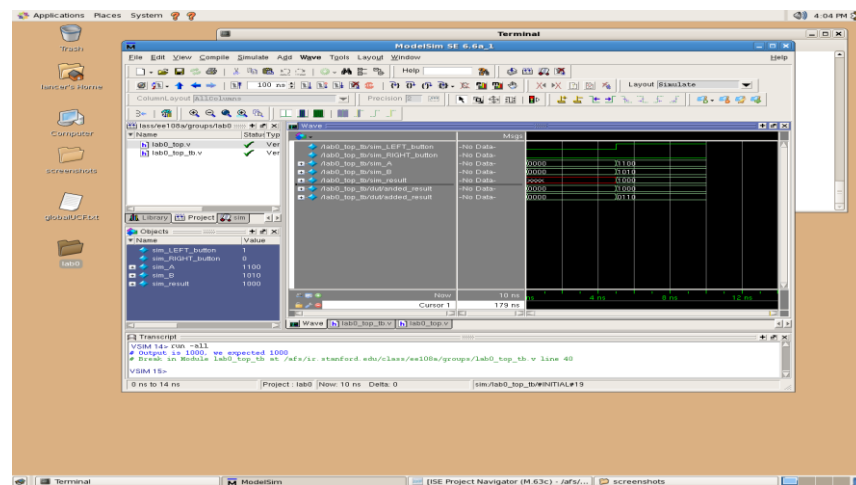


Figure 24

So that looks pretty good! We pushed the LEFT_button and our result changed as we expected. Note that you can see that we are calculating both `anded_result` and `added_result` the whole time, but our output is just the one we want.

Let's make our test bench a bit more interesting by adding the following:

```
// Try adding
sim_LEFT_button = 1'b0;
sim_RIGHT_button = 1'b1;
sim_A = 4'b1100;
sim_B = 4'b1010;
#5

$display("Output is %b, we expected %b", sim_result, (4'b1100 + 4'b1010));

// Try changing our inputs, note that we're still adding!
sim_A = 4'b0001;
sim_B = 4'b0011;
#5

$display("Output is %b, we expected %b", sim_result, (4'b0001 + 4'b0011));

// Let's go back to ANDing
sim_LEFT_button = 1'b1;
sim_RIGHT_button = 1'b0;
#5

$display("Output is %b, we expected %b", sim_result, (4'b0001 & 4'b0011));
```

Remember to add this before the \$stop or we'll never get there.

Now here's a great HINT:

We need to re-compile this file since we changed it, but we don't need to stop simulating to do that. So click on the Project tab and right-click and choose Compile -> Compile Out-of-Date. As long as you didn't have any errors you can now restart and re-run your simulation. To do this choose Simulate -> Run -> Restart and click Restart.

Note what appeared in the Transcript pane when you did this:

```
VSIM 4> restart -f
```

That is, the text command for choosing that menu was "restart -f", so you could just type restart -f instead of choosing the menu. That's not much faster, but what is faster is that you will be making lots of changes and recompiling so what you want is a fast way to execute: "**restart -f; run -all**". In fact, if you type that in it will do the restart and then run your simulation. Even better is if you use the up-arrow key in the Transcript window it will bring you back to the previous command you entered. So, instead of choosing Simulate → Run → Restart, click Restart, then Simulate → Run → Run-all, you can type in "restart -f; run -all" once, and then just press up-arrow, return to do both commands. You'll really appreciate this on later labs!

So run your design with run -all (either by typing it and pressing return, typing up-arrow until you get to your previous command and pressing return, or using the slow-old-fashioned menu).

You should see (**Figure 25**):

```

VSIM 5> restart -f; run -all
# Output is 1000, we expected 1000
# Output is 0110, we expected 0110
# Output is 0100, we expected 0100
# Output is 0001, we expected 0001
# Break in Module lab0_top_tb at Z:/lab0/lab0_top_tb.v line 66

```

Figure 25

And your Wave pane should now have (Figure 26):



Figure 26

Here's an important comment: How easy is it for you to understand the wave diagram above vs. the text? That's right, unless you've been staring at it for a while, the wave diagram is just confusing. This is the same for TAs grading your labs. So when you submit a wave diagram for a lab report you **MUST** annotate it with what's going on so we know what to look at, i.e. **Figure 27**:

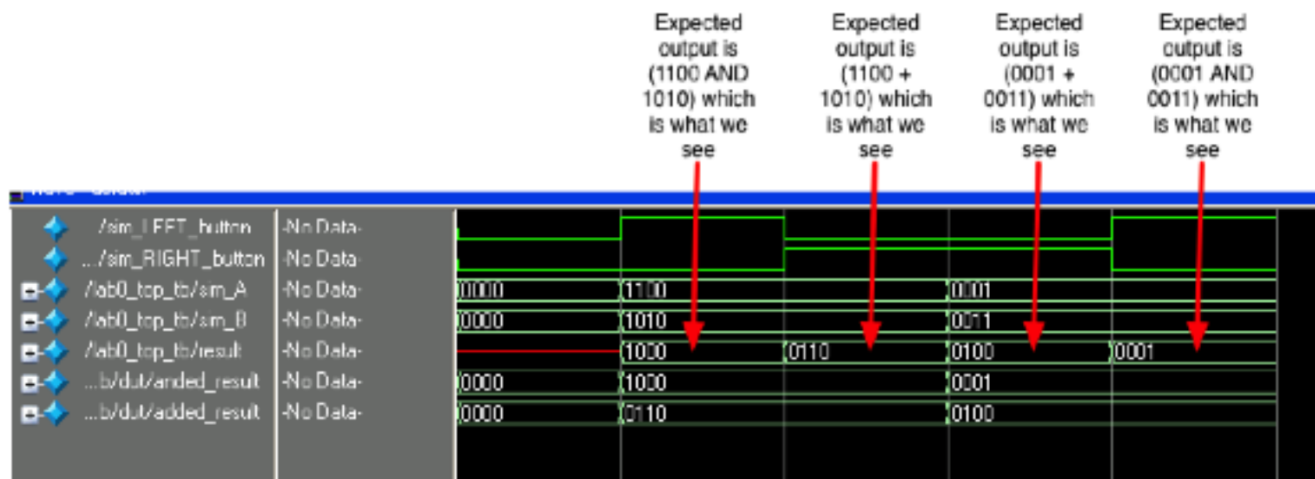


Figure 27

If you don't do this we **will not even read** your waveform diagrams. It doesn't have to be anything fancy, but you **must** annotate your waveforms.

Step 3: Synthesizing your design

So at this point it looks like the design is working. Now we want to actually synthesize it for the FPGA. We'll summarize the crucial steps, but you should still read up on the documentation you can find here: <https://www.ece.ucsb.edu/courses/ECE152/digilab-fpga/>

Open Quartus II and create a new project. Save it where you will be able to easily access the folder. (If you use the X:\ drive, you can access the files from any Windows computer on the network). On page 3 of 5 of the setup dialog, make sure the configuration window looks like this (Figure 28): Then, continue through the setup.

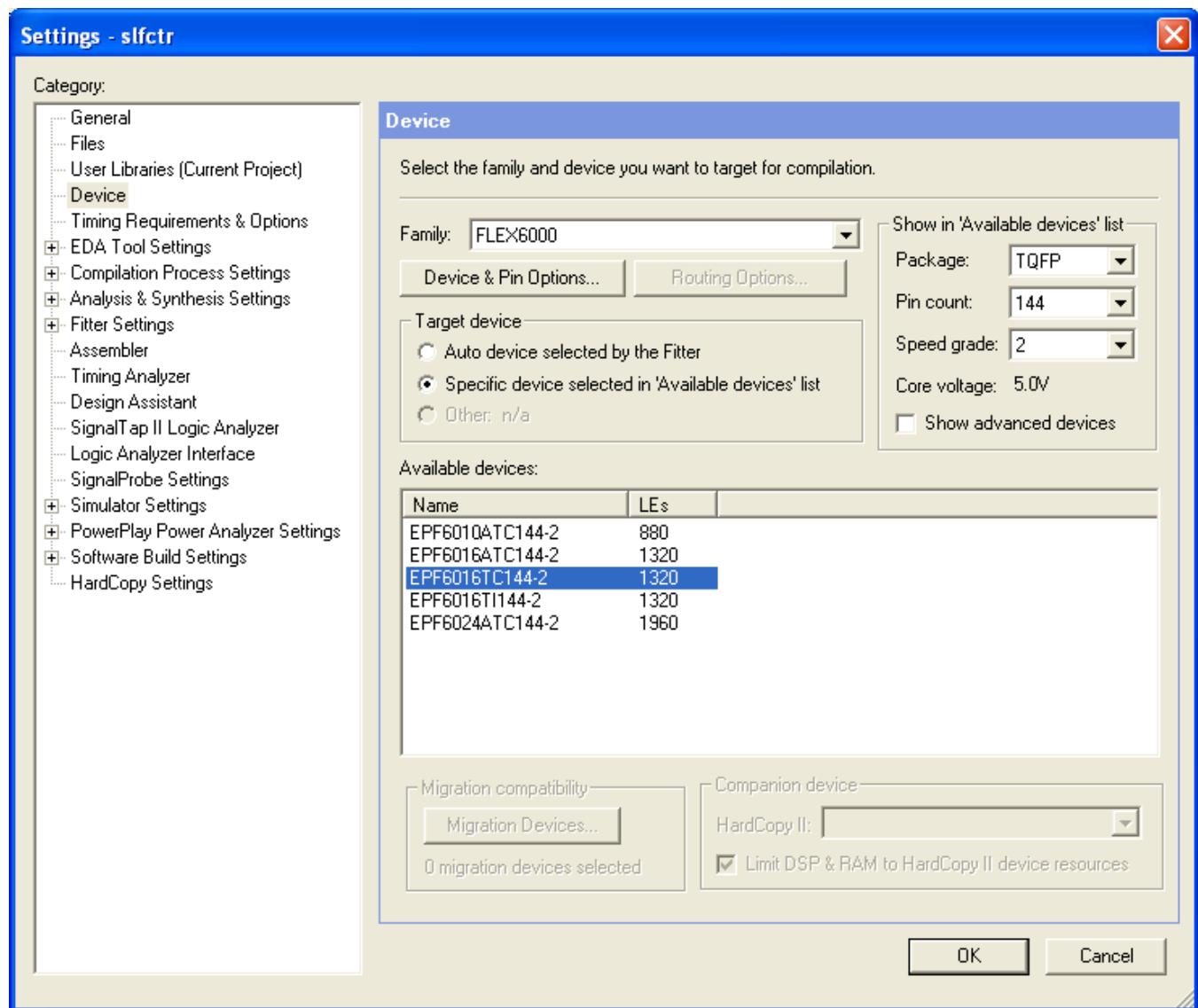


Figure 28

Next, we need to take care of some configuration to make sure everything works. Go to **Assignments->Device**. Click the **Device and Pin Options** button. Under the **General** tab, enable the **Enable JTAG BST Support** option (Figure 29). Under the **Configuration** tab, you must select **Passive Serial** for the Configuration Scheme. Be sure to leave the **Use configuration device** option unselected. Under the **Programming Files** tab, enable the **Raw Binary File (.rbf)** option. Under the **Unused Pins** tab, we recommend you select the option "As input tristated" for the behavior of all unused pins. It is OK to go ahead and familiarize yourself with some of the other choices but the remaining tabs of this dialog either control options that are not applicable to the Flex-6000 device or the settings you may choose should not matter in the application. At this point we should be ready to build the project.

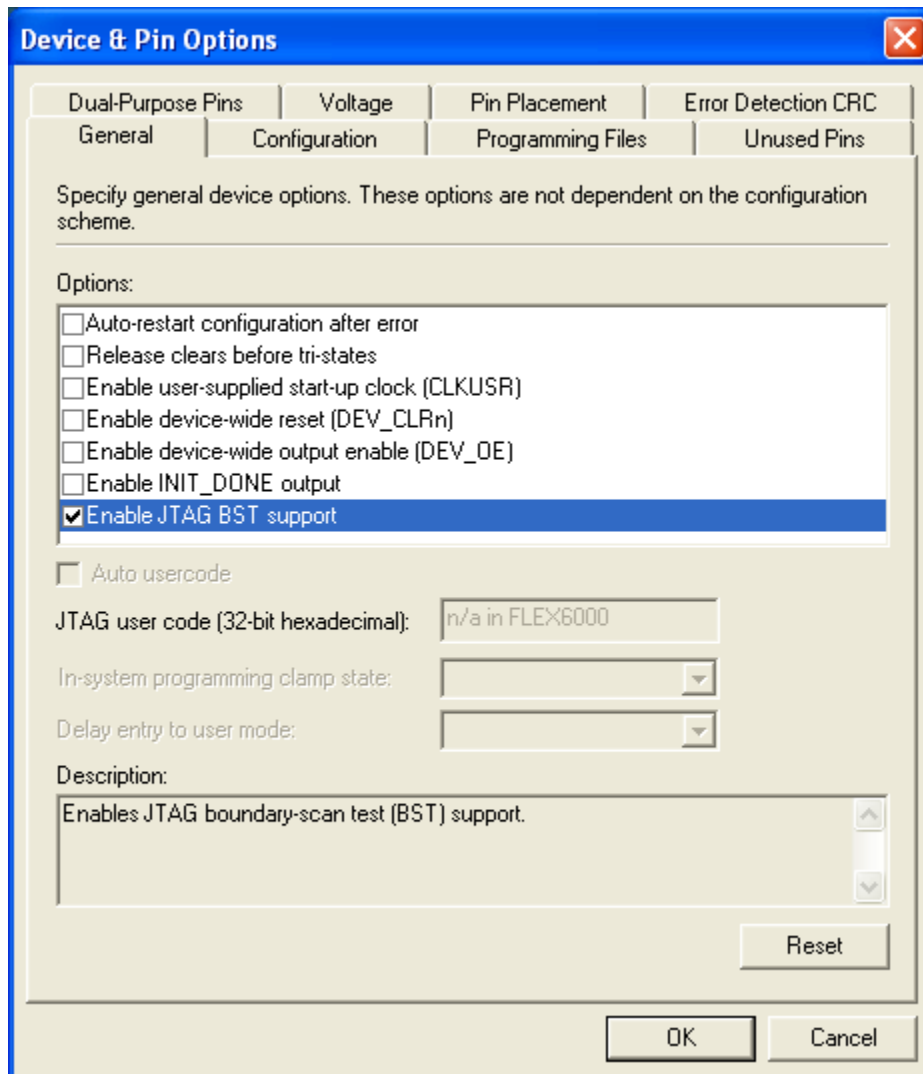


Figure 29

Next, we need to add our file(s). Within the project create a new verilog file for each module you have implemented by going to **File->New->Verilog HDL File**. We have only one synthesizable file, lab1_top.v. Remember that our test bench uses an `initial` statement which is not synthesizable.

5. Create a new file called “lab1_top.v” Paste the code from Modelsim into the new file. Note that you could also add the existing file to your project so your code doesn’t exist in two places. Once the code is in a file that Quartus can see, we need to designate it as the top level entry. Go to the Files tab of the Project Navigator window in the upper left corner. Right click on the module, and then select “Set as top-level entry.”

Now let’s synthesize our design.

Use the top menu to navigate to **Processing->Start Compilation**. Even for our small module, this will take a bit of time. Notice how the console window at the bottom logs the compiling process. When it’s finished, you’ll see a window with errors and warnings if any occurred. There shouldn’t be any errors at this point. If there are, make sure your code looks correct, and then double check all of the configuration settings. If the errors persist, talk to your TA. What you should see are a few warnings. Scroll up through the log and see if you can find them.

When compiling designs, you want to look out for **inferred latches**. If you have inferred latches it means that the logic you designed doesn’t define the outputs for all the cases. If you do this I guarantee you that you will get almost correct, but very hard-to-debug behavior. For example:

```
input button;
output out;
reg out;

always @* begin
    if (button == 1'b1)
        out = 1'b1;
end
```

Let’s take a look at this code. If button is true then out will be true. What is the value of out if button is not true? Well, it’s undefined, so Verilog cleverly says, “Ah, I know how to solve this! I’ll just infer a latch to store the previous value of out and use it whenever the current value isn’t defined!” Unfortunately that is not what you want. In fact, if you do this we will deduct points from your labs. (All storage has to be explicitly instantiated as flip flops, but we’ll talk about that later in class.)

So how do you avoid this? Simple: for every **if** you need an **else** and for every **case** you need a **default**, and every reg that is set in **one** part of the if/else or case must be set in **all parts** of the if/else and case. If you remember those rules you will make sure that you define every output in every case.

So to fix the above code we would simply add:

```

input button;
output out;
reg out;

always @* begin
    if (button == 1'b1)
        out = 1'b1;
    else
        out = 1'b0;
end

```

Now it is explicit what the value of `out` is for all possibilities and no latches will be inferred.

What you will find in the compilation log is an inferred latch error. We need to fix it! Open the Verilog file and find the logic where we define the `result` signal.

```

reg [3:0] result;
always @* begin
    case( {LEFT_pushbutton, RIGHT_pushbutton} )
        2'b01: result = added_result;
        2'b10: result = anded_result;
        2'b11: result = added_result; // right takes precedence
    endcase
end

```

Can you see where the problem is? What is the value of `result` if neither of the buttons are pressed? Well, it's undefined, so Verilog very politely infers a latch to store the previous value and use that.

Let's fix it by adding an `else` at the end which defines the value of `result` when no button is pressed, and recompile.

```

reg [3:0] result;
always @* begin
    case( {LEFT_pushbutton, RIGHT_pushbutton} )
        2'b01: result = added_result;
        2'b10: result = anded_result;
        2'b11: result = added_result; // right takes precedence
        default: result = 4'b0;
    endcase
end

```

This is important: Why didn't we see this error when we simulated our design? Wasn't that the whole point of simulation? Well, the answer is that we did see it. Look at the undefined line for `result` before the `LEFT_button` goes high. The output is undefined because we didn't tell it what the output should be when neither button is pressed. Any time you see undefined signals (x values or red lines) in ModelSim you should make sure you understand what's going on. Almost all of the time they are the result of errors in your design and you need to fix them!

This time you should have succeeded in synthesizing your design with no errors and you're now ready to download it to the FPGA board and test your design. Before moving on, we need to assign pins to the signals in the circuit. We can do this by going to **Assignments->Pins**. This will bring up a new window with all the inputs and outputs defined in the top level module. Each signal can be assigned via a drop down menu in the **Location** column. Once all the pins are assigned, we can close the window and compile the code again. If you followed all the instructions, you'll find a file in the Quartus directory you defined when you created the project called "lab1_top.rbf" Each time you change the code or pin assignments of your design, you will need to regenerate this file by recompiling.

Next, navigate to <https://www.ece.ucsb.edu/courses/ECE152/digilab-fpga/> and download FPGAtool (from the top right). Make sure your FPGA is connected to the DC jack at your station and connected to the computer via USB. Before you flash on the code, you want to make sure all the pins you have assigned are functional. *This is very important!* Real hardware breaks, and we want to make sure we don't waste time troubleshooting issues in the code that are actually coming from the hardware. There are a few ways to test the pins. One way is to click on "Board Test" in the FPGA Tool window. Then, you can drive all the pins high or low. You can probe the outputs with the multimeters at your lab stations to verify they work. You can also drive the pins you want to test as inputs, and select "Capture Pins" in the Test Mode. If you do this, **make sure you aren't driving the pins when you hook up external inputs! This can break the pins.**

It's very important that you take input voltages from the power supply pins on the FPGA rather than from the voltage source at your testbench. Can you explain why using another power supply is a bad idea?

Once we have verified that the pins are working, we can hook up the circuit. We will use LED's to indicate the output. You can purchase these from the shop, but look up the part number before you get there! Hook up the positive terminal of the LED to the output pins you assigned and hook the negative terminal to ground. I find this is easiest by tying the rails of a breadboard to the power and ground pins of the FPGA so I can put the LEDs on the same row as the output pins. Then, tie the A and B input pins directly to power and ground, copying the values you set in the modelsim testbench. Finally, hook up your push-button inputs, either by directly tying the pins to a supply, or purchase some pushbuttons and current limiting resistors from the shop and build a dynamic input circuit. If you choose to do this, make sure you don't exceed the power rating of the resistors. The power supply on the FPGA is more than capable of burning out resistors, creating a foul smell that everyone in the lab will resent you for.

Finally, use the "Browse" button on the FPGAtool to select the .rbf file you compiled. Click "download," and test out your circuit. If it doesn't work, do some troubleshooting and fix it. When it does work, grab your TA and demonstrate that your circuit works. Be prepared to answer a few questions about the topics we've covered here.