Introduction

Welcome to EE108a Lab 0. This lab is designed to familiarize you with Verilog and the tools we'll be using in EE108a. The lab is a step-by-step walkthrough which will take you from the initial specifications through writing the Verilog, testing and debugging in simulation, synthesizing your design for the FPGA hardware, and finally running your design for real.

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 TA for an explanation!

Design

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 to take in two 4-bit values A and B, and then do simple operations on them. The result will be displayed on 4 LEDs on the board. We'll start out implementing two functions: (A AND B) and (A + B), and then we'll move on to some more complicated logic.

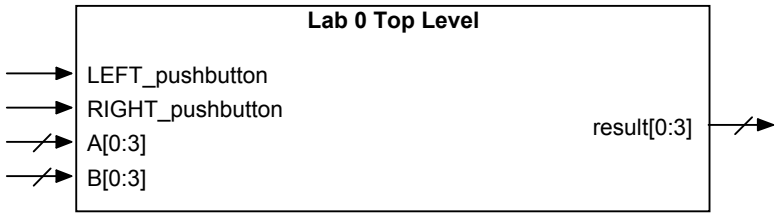
Specifying the design

To begin our design we'll specify the desired behavior:

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

From that description we can define the top-level inputs and outputs are. These are the connections that come into the design from the real world and go out to show the results.



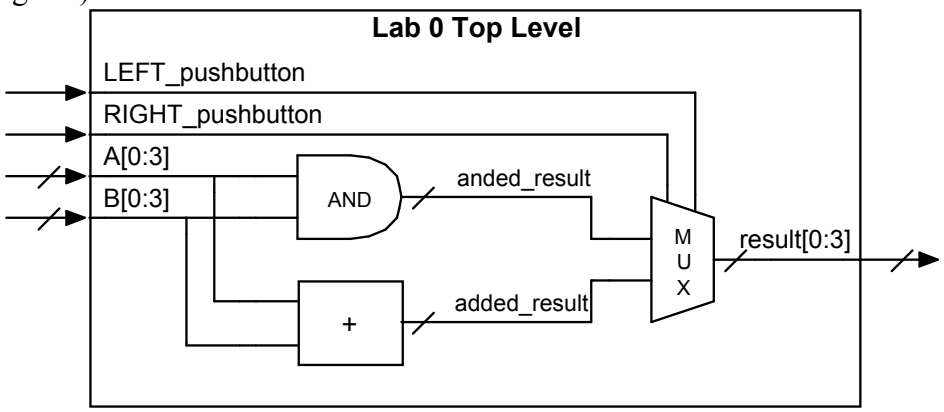
Signal	Direction	Width/Details	Purpose
LEFT_pushbutton	Input	1 bits true = pushed false = not pushed	When pushed we show A anded with B
RIGHT_pushbutton	Input	1 bit true = pushed false = not pushed	When pushed we show A plus B
A	Input	4 bits (binary 0-15)	This is our A input from the first 4 switches 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 and B or A + B according to the button we press

From this definition we can even write the Verilog for our top level module's inputs and outputs:

```
module lab0_top (
      LEFT_pushbutton,
      RIGHT_pushbutton,
      A,
      B,
      result
);

      input LEFT_pushbutton;
      input RIGHT_pushbutton;
      input [3:0] A;
      input [3:0] B;

      output [3:0] result;

      // Fill in logic here...
endmodule
```

Now this is a simple enough module 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 addition. Since these are simple functions (only one line each) we'll use wires and assign statements. In Verilog assign statements always have to define the output of wires. So anything on the left side of the = in an assign statement must be defined as wire.

Here's the design we want to implement. (A MUX (multiplexer) is a device which selects an input based on the control signals).



So we'll start out by making some intermediate wires for these results:

```
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 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 registers, so we need to define the output of our always block as a reg.

```
      reg [3:0] result;
      always @* begin
            if (LEFT_pushbutton == 1'b1) begin
                  result = anded_result;
            end
            else if (RIGHT_pushbutton = 1'b1) begin
                  result = added_result;
            end
      end
```

For those of you who paid attention in section this 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.

Notice that we've defined result as both an output and a reg. This means is that it is the result of an always block that goes out of our module. If we had used an assign statement for 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 = (LEFT_pushbutton) ? anded_result : (RIGHT_pushbutton)
? added_result : 4'b0;
```

This does roughly the same thing as the above always block, but is a lot harder to understand! (Indeed it is slightly different, 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 lab 0 top module, now we just need to fire up ModelSim, write it, and see how it works.

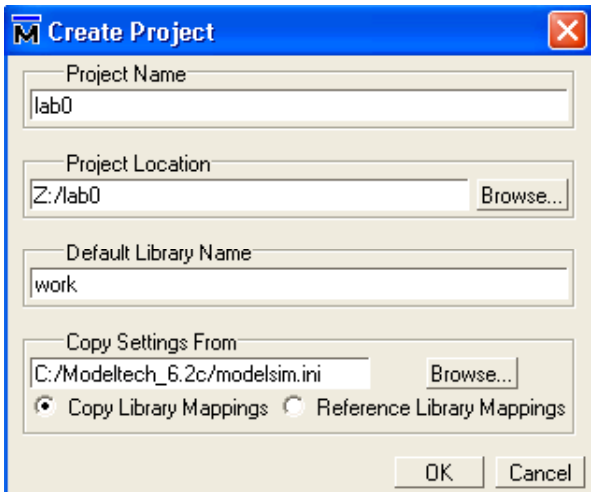
Entering your design in ModelSim

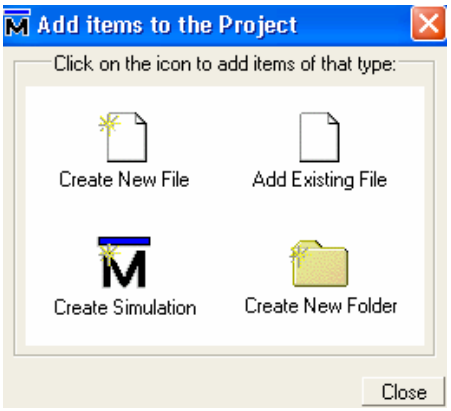
Log into your EE108a lab account on one of the computers in the lab and go to your networked Z drive. (You can access this from Start -> My Computer -> Network Drives.)

Create a new folder called “lab0” on your networked Z drive. You will use this to store all the files you create while working on Lab 0. When you are done with a lab you copy the final version to the Submit folder.

Now we’ll start ModelSim and create a new project. Run ModelSim by double clicking on the ModelSim 6.2c icon on the desktop.

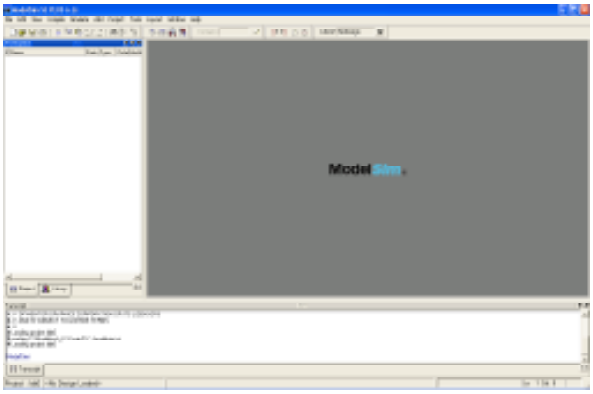
Create a new project for lab 0: File -> New -> Project…
The project name is “lab0” and the location should be in the lab0 folder you just created.
Keep the library name as work (this is the name of the folder inside your lab0 folder where ModelSim will keep all its data) and click ok.



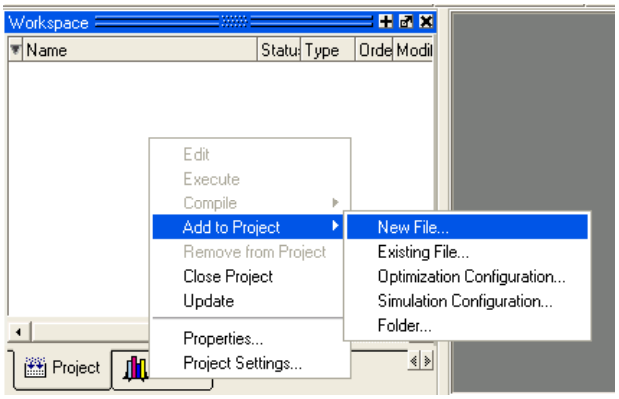


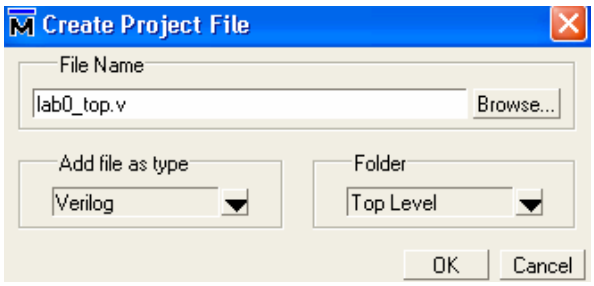
Close the “add new items” window that appears

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. If you don’t see the Workspace pane select it from the view menu.



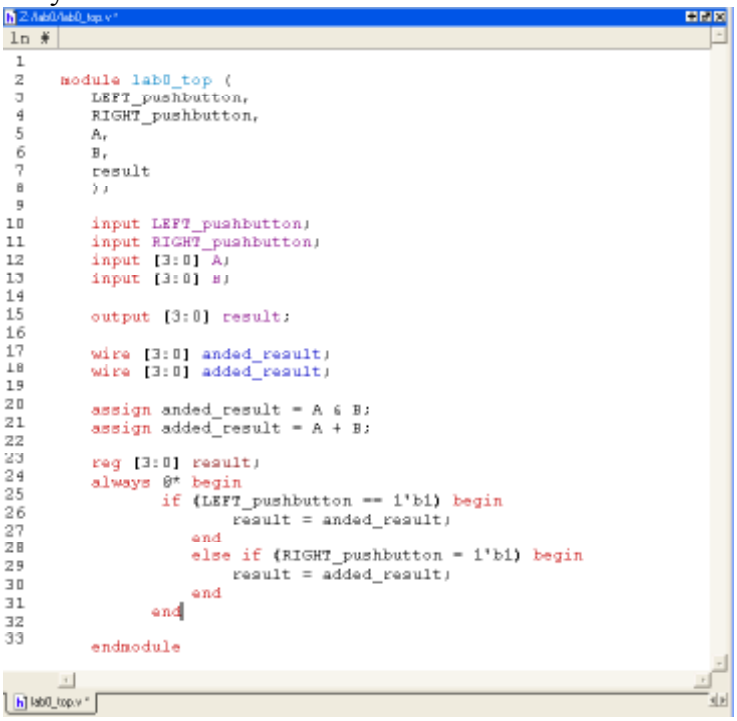
The next step is to create our Verilog files. At the moment we have one module (lab0_top) so we’ll only need one file. Make sure the Workspace Pane is set to the Project tab then right click and choose Add to Project -> New File…





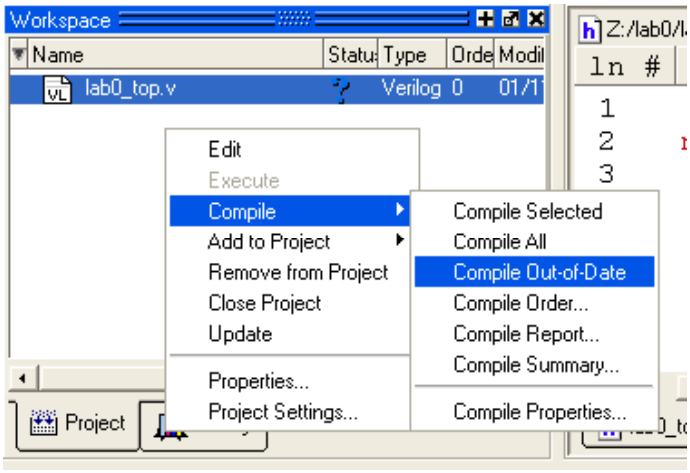
Name the file "lab0_top.v" and make sure you select the type as Verilog and NOT VHDL. Your file will now appear in the Project tab with a question mark indicating it hasn't been compiled yet.

Double click on your file to edit it. It will open in a new editor tab on the right. Now enter the Verilog for your module from above:

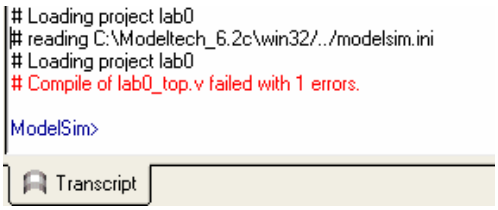


```
module lab0_top (
    LEFT_pushbutton,
    RIGHT_pushbutton,
    A,
    B,
    result
    );

    input LEFT_pushbutton;
    input RIGHT_pushbutton;
    input [3:0] A;
    input [3:0] B;

    output [3:0] result;

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

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

    reg [3:0] result;
    always @* begin
          if (LEFT_pushbutton == 1'b1) begin
                result = anded_result;
             end
             else if (RIGHT_pushbutton = 1'b1) begin
                result = added_result;
             end
          end

    endmodule
```

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

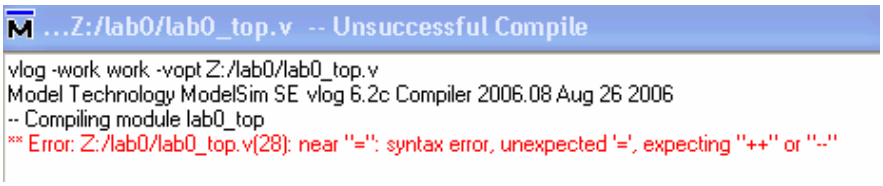


If it compiled you're all set, if not it will tell you there is an error in the Transcript window:



Double-click on the error line and a window will pop-up with 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.

In my case I got:



Which isn't very helpful, until I look at line 28 and see that I wrote:

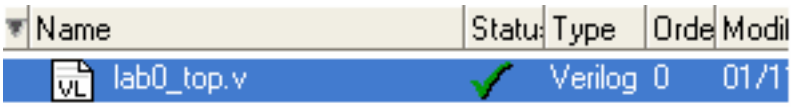
```
      else if (RIGHT_pushbutton = 1'b1) begin
```

When I should have written:

```
      else if (RIGHT_pushbutton == 1'b1) begin
```

Changing that, saving the file, and re-compiling produces:

And a pretty green check mark:



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 don't have any inputs. Our module takes in 4 inputs and produces 1 output. 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 lab0_top) is called the Device Under Test, or dut.

The difference between Test Benches and regular synthesizable Verilog modules are that Test Benches often have statements like $display() for printing output to the screen, and they can use initial, for, repeat, and while blocks, which are not allowed in your synthesizable Verilog code. These extra commands help you to test your design, and that's really important.

Do you remember how much time it took to compile your lab0_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. Speed 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*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*256 = 65,535 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 1.84x10^19 possible choices in a 64-bit value, and with two of them that's 3.4x10^38 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 1x10^20 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.

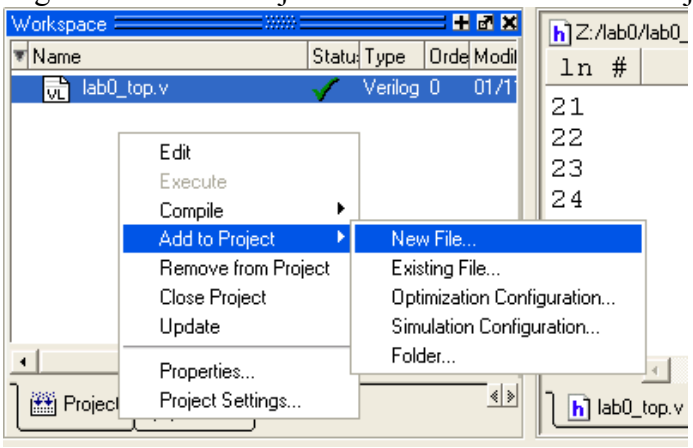
As an aside: if someone ever tells you that something won't work because an algorithm or problem runs in exponential time, this is exactly what they mean. As soon as you get a tiny bit bigger (64 bits is only 16 times as many as 4) the problem becomes absurdly impossible.

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…



Create a new Verilog file called lab0_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 ee108a. (Trust me: if you do this carefully it will save you an amazing amount of time with labs 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. (You'd need another Test Bench for the Test Bench if it needed inputs itself. Sometimes this happens on much larger projects, though.)

So let's start out our Test Bench:

```
module lab0_top_tb ();
      // No inputs or outputs

endmodule
```

So far pretty easy. Now we need to add the signals we're going to need to send into the device under test, which is our lab0_top module.

```
            // These are the signals we need to generate for the device under
test
            reg sim_LEFT_button;
            reg sim_RIGHT_button;
            reg [3:0] sim_A;
            reg [3:0] sim_B;

            wire [3:0] result;
```

Note that the signals that are going to go into the device under test are registers 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. For example, if we wanted to hookup a note_player module called my_note_player_instance we might do the following. Here the current_note, play_button, and pause_button are signals in our module and note, play, and pause are the inputs in the note_player to which we are attaching them.

```
      EXAMPLE:
      note_player my_note_player_instance (
            .note(current_note),
            .play(play_button),
            .pause(pause_button)
      );
```

So here is our instantiation of a copy of our lab0_top, which we are going to call "dut" for Device Under Test. (You can call it anything you want in general.)

```
      lab0_top dut (
            .LEFT_pushbutton(sim_LEFT_button),
            .RIGHT_pushbutton(sim_RIGHT_button).
            .A(sim_A),
            .B(sim_B),
            .result(result)
            );
```

Note how we hooked up every input or output from the lab0_top module. If you forget 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 running in parallel at the same time*.

So now our test bench defines the signals going into our Device Under Test (our lab0_top module) and instantiates 1 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 Test Benches, that is, they are not synthesizable into logic. When your simulation starts Verilog will process 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.

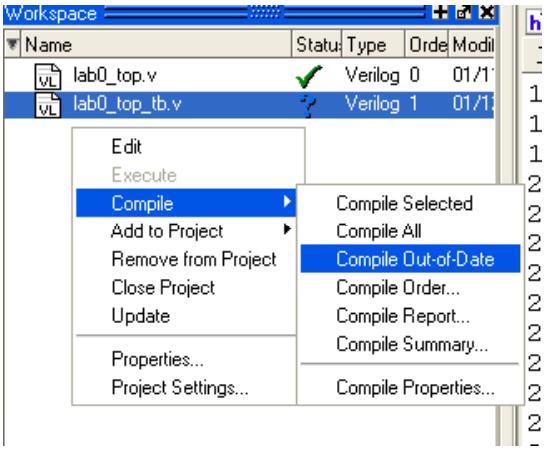
So here's what we'll use to start with:

```
      initial begin
            // Start 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 = 1'b0;
            sim_B = 1'b0;

            // Now let's run the simulation for 5 timesteps
            #5

            // Our first test: try ANDing
            sim_LEFT_button = 1'b1;
            sim_A = 4'b1100;
            sim_B = 4'b1010;
            #5

            $display("Output is %b, we expected %b", result, (4'b1100 &
4'b1010));
            $stop
      end
```

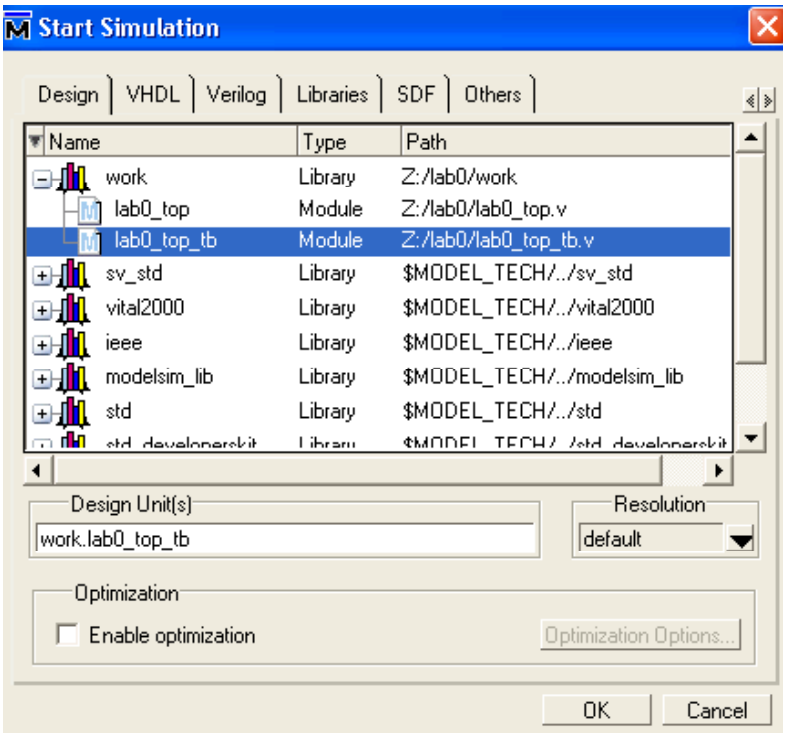
The key parts of this test bench so far are that we set our inputs, then we delay and change our inputs. The delay (#) tells the simulator to run for some number of time steps before the next statement. The $stop command tells the simulator to stop, and the $display() tells the simulator to print out some text. 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.

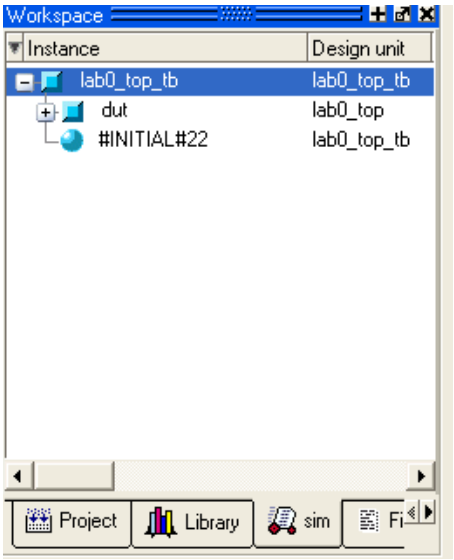
Right-click in the project tab and choose Compile -> Compile Out-of-Date.



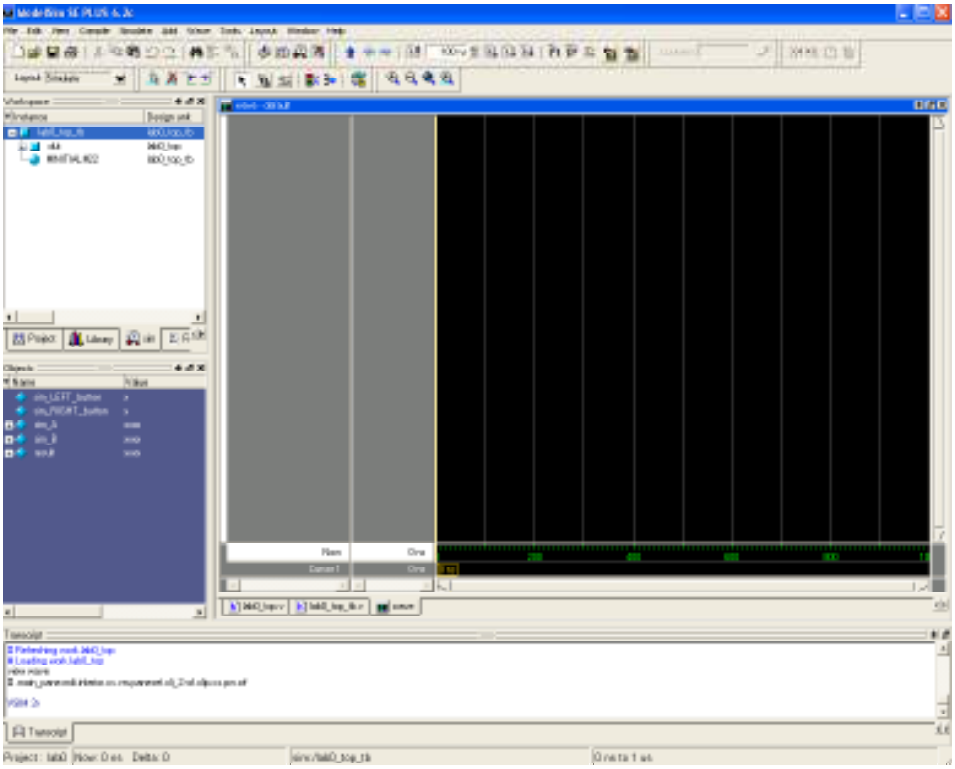
If everything compiles, we now want to run our simulation and view the results. Choose Simulate -> Start Simulation, then under the Design tab expand "work" (remember when we created the ModelSim project we told it to use "work") and select your lab0_top_tb as the top module. Also **UNCHECK THE "Enable 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.



Note that the Project tab has been replaced with the sim tab. The sim tab shows you the hierarchy of your design. Here we just see that we have one module in lab0_top_tb, and that it is called "dut".

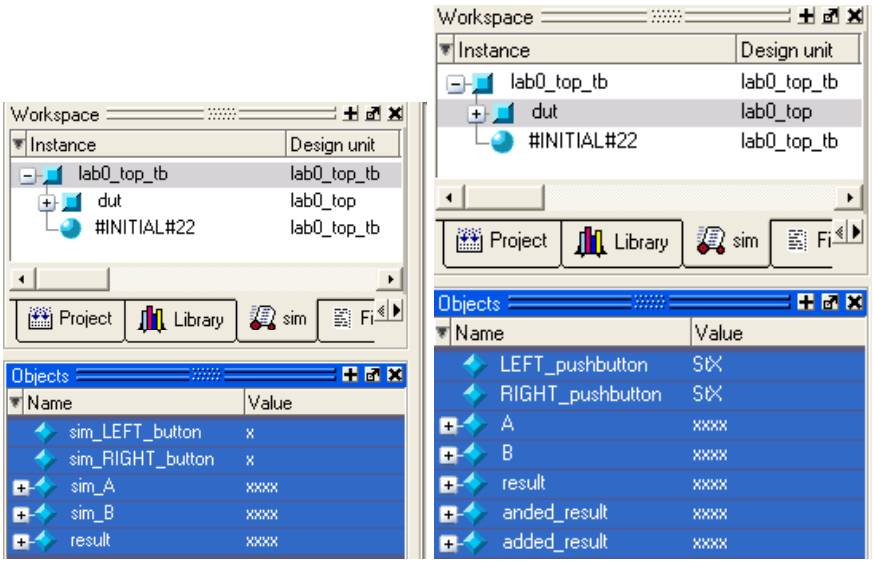


From the View menu choose View -> Objects and View -> Wave. I like to arrange my screen with the Objects pane below the Workspace pane as you can see.



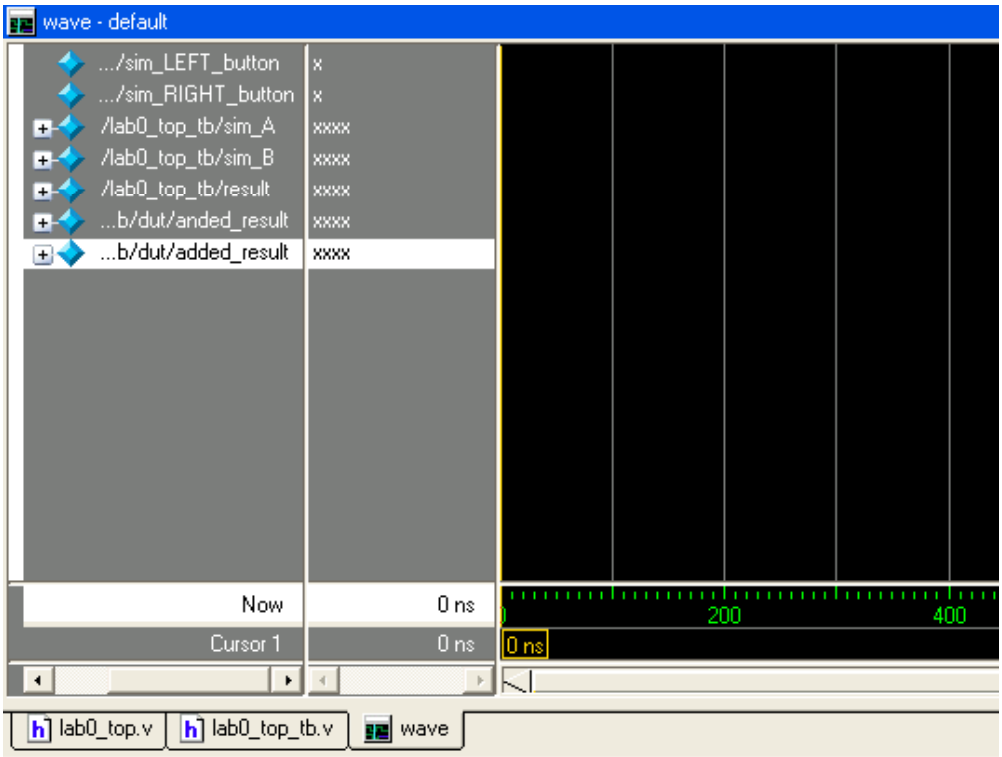
Now we want to watch what's going on in our simulation, so we will add objects from the Objects pane to the Wave pane.

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. If I have the dut module selected I see the signals inside the dut module.



For this simulation we want to look at all the signals in the top-level lab0_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:



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 delay 5s), but later on we’ll get much longer test benches!

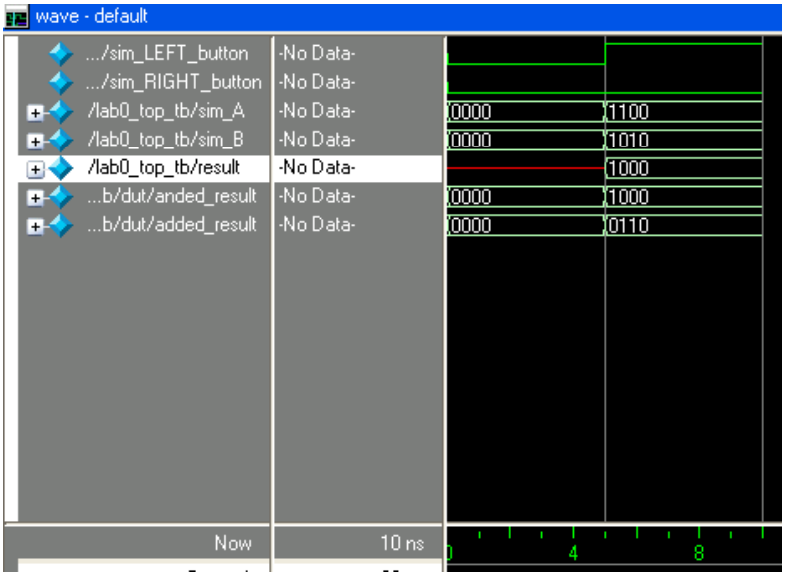
So now let’s run the simulation. Choose Simulate -> Run -> Run –all

Your simulation will run until it gets to the $stop we inserted, and you should see the output:

```
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
```

Go to the Waves pane and let's see what we got:

Kind of hard to see, hm? Well, just zoom in on it.



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:

```
            // 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", 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", 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", 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
```

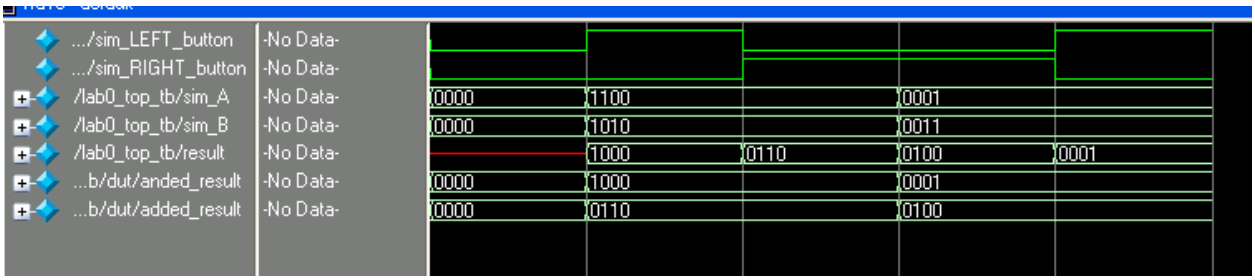
That is, the text command for choosing that menu was "restart –f", so you could just type restart –f instead of choosing the menu. That's not much faster, but what is faster is that you will be making lots of changes and recompiling so what you want is a fast way to execute: "restart –f; run –all". In 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:

```
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
```

And your Wave pane should now have:



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.,

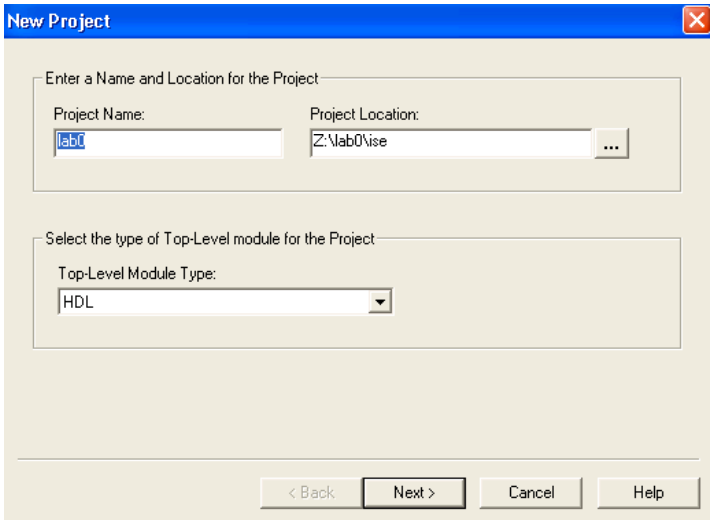
If you don't do this we **will not even read** your waveform diagrams. It doesn't have to be anything fancy, but you **must** annotate your waveforms. A simple way to do this is to use the Print Screen button on the keyboard after getting an informative portion of the wave centered on the screen, and then pasting into a picture editing program such as Paint. Use the pop-out button in the upper right to get a larger view of the wave before using Print Screen.

Synthesizing your design

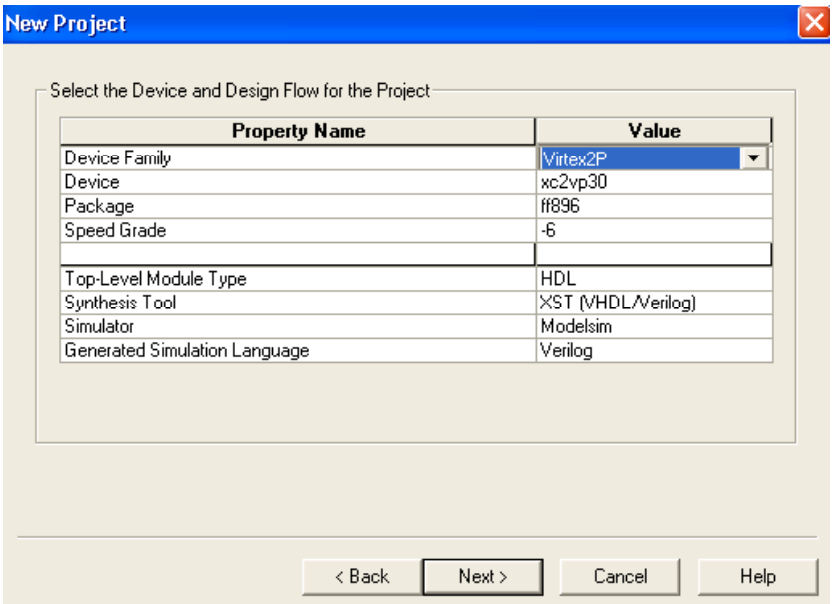
So at this point it looks like the design is working. Now we want to actually synthesize it for the FGPA. To do this we need to create a new project in Xilinx ISE and add our files to it. We'll also need one

important file: the UCF, or user constraints file. This file is what will tell us which physical wire on the FPGA connects up to our inputs.

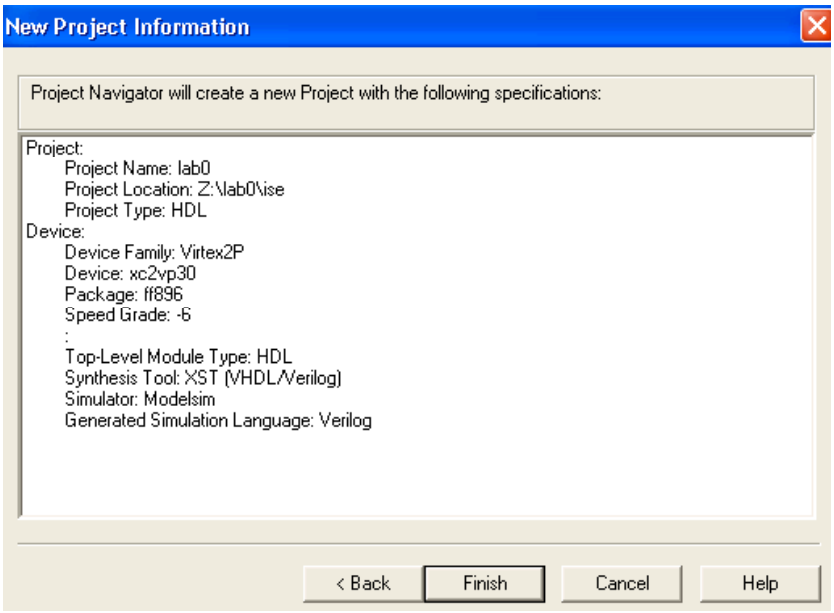
To get started create a new folder in your lab0 folder called “ise” and run Xilinx ISE by double-clicking the icon on the desktop. Just like ModelSim we need to create a new project. Choose File -> New Project… Name your project lab0 and put it in the ise folder in your lab0 folder. Click Next.



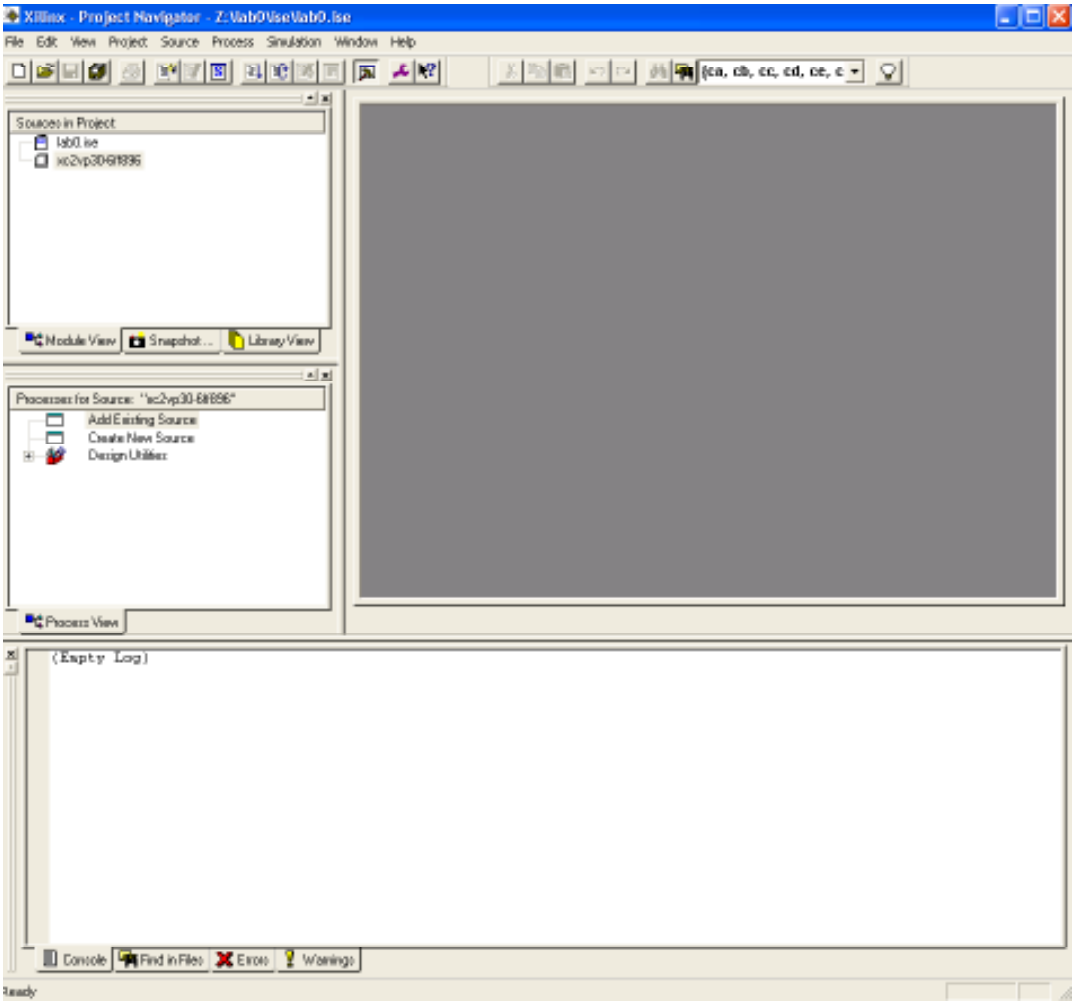
Make sure you select the right options here, or nothing will work. This is where you tell ISE which exact chip we have and what language you’re using. **Mimic the window below exactly!**



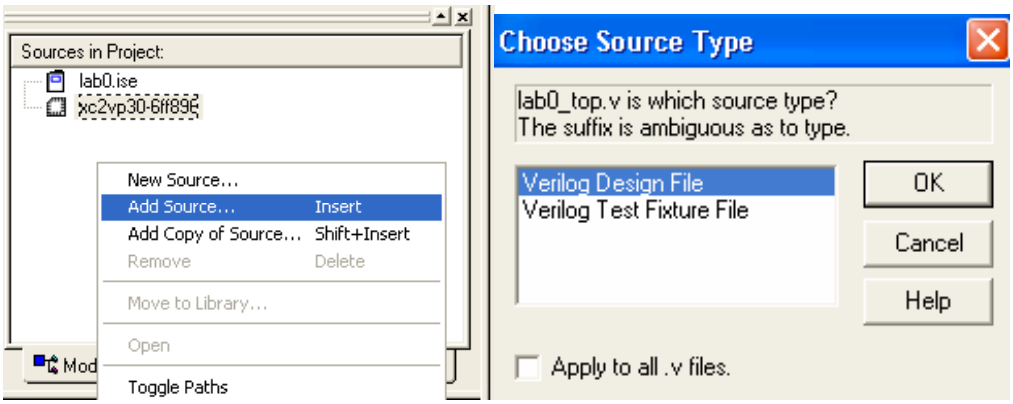
Click through until you get to the final screen. Make sure the device and language are correct, then click Finish.



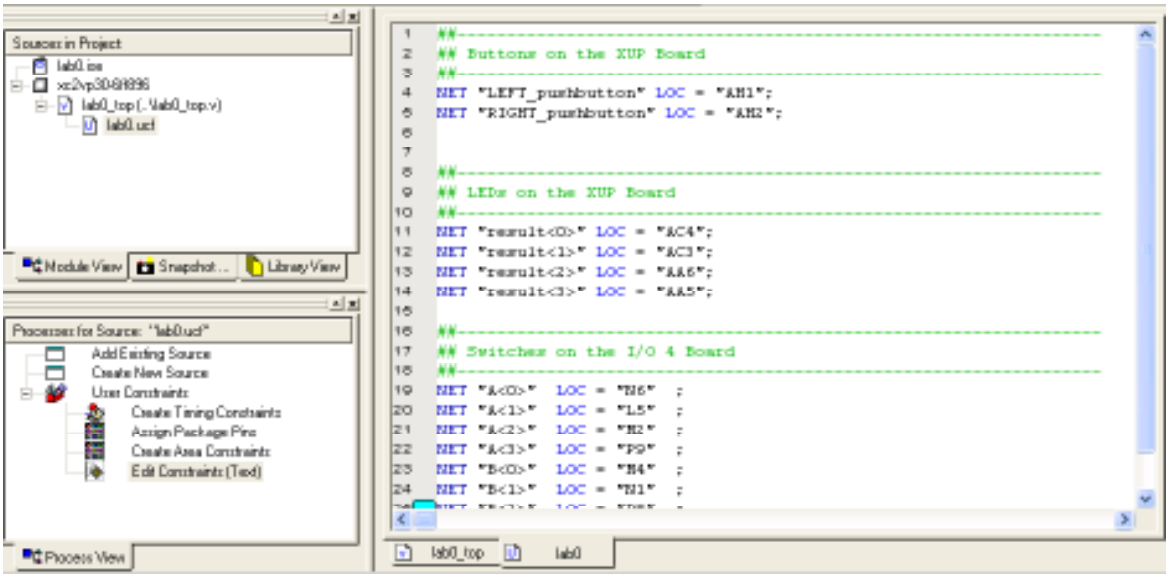
Note that your ISE project looks a lot like your ModelSim project. You have a list of all the files in your project in the upper left, a transcript window on the bottom and a space on the right for editing your files.



The first thing we need to do is add our file(s). We have only one synthesizable file, lab0_top.v. Remember that our test bench uses an initial statement which is not synthesizable. To add it right-click in the Sources pane and choose Add Source…, select your file and tell Xilinx it's a Verilog Design File.



Now we need to add the UCF file. It's up on the web on the class schedule page. Go and download it from the labs section and place it in the ise folder.
Now add the file to the project in the same manner. When you select the lab0.ucf file you can double-click on "Edit Constraints (Text)" below in the Processes tab to see what's in this file. As you see it just defines the inputs and outputs.



Now let's synthesize our design.

Select the lab0_top module in the Sources pane and double-click on Generate Programming File in the Processes pane. Note: Xilinx will generate a programming file **with whatever file you have selected**, **so make sure you select your top-level file!** After a long wait your project will finish synthesizing.

It's important that you make sure you don't have any errors or warnings in your design. To check them click on the Errors and Warnings tabs on the bottom and see if you had any. If you did you need to fix them before you continue.

In particular 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 in EE108a we will deduct points from your labs. (All storage has to be explicitly instantiated as flip flops in EE108a, but we'll talk about that later in class.)

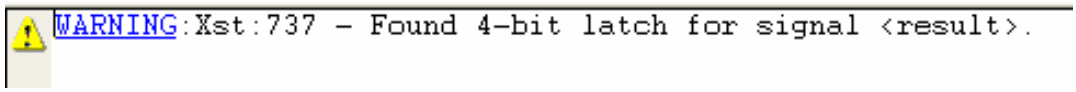
So how do you avoid this? Simple: for every **if** you need an **else** and for every **case** you need a **default**. If you remember those rules you will make sure that you define every output in every case.

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.

If you followed this handout you should have:



Clicking on the blue WARNING link will take you to a rather useless webpage. Instead click back on the Console tab and scroll up until you find the warning.

```
Synthesizing Unit <lab0_top>.
    Related source file is "../lab0_top.v".
WARNING:Xst:737 - Found 4-bit latch for signal <result>.
    Found 4-bit adder for signal <added_result>.
    Summary:
      inferred   1 Adder/Subtractor(s).
Unit <lab0_top> synthesized.
```

So now we know that the error is in lab0_top.v (of course that's our only file, so it would be hard for it to be anywhere else). So open the file and find the logic where we define the result signal.

```
reg [3:0] result;
always @* begin
       if (LEFT_pushbutton == 1'b1) begin
           result = anded_result;
          end
          else if (RIGHT_pushbutton == 1'b1) begin
           result = added_result;
          end
      end
```

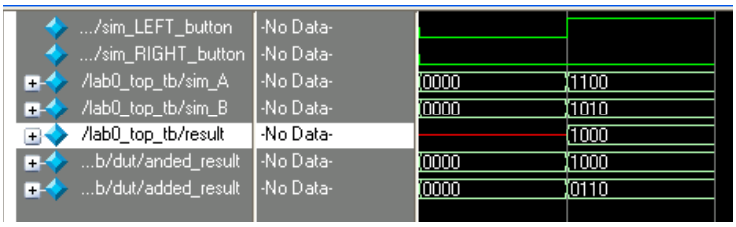
Can you see where the problem is? What is the value of result if neither of the buttons are pressed? Well, it's undefined, so Verilog very politely infers a latch to store the previous value and use that.

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

```
reg [3:0] result;
always @* begin
       if (LEFT_pushbutton == 1'b1) begin
           result = anded_result;
          end
          else if (RIGHT_pushbutton == 1'b1) begin
           result = added_result;
          end
          else begin
            result = 4'b0;
          end
      end
```

Now re-build everything by double-clicking on Generate Programming File.

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 out your design.

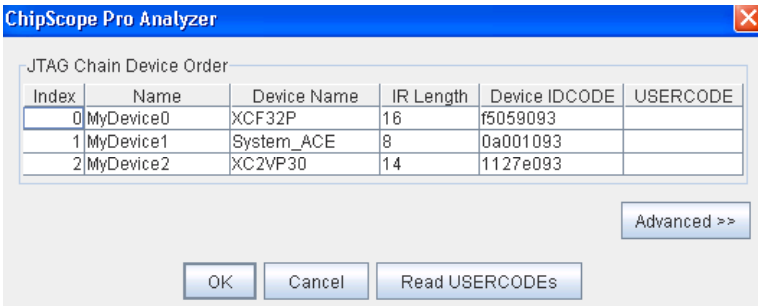
To download your design double-click on the Analyze Design Using ChipScope icon. This will launch ChipScope, which is the program we use to communicate with the FPGA boards. (Hint: if you double-click on this button two times you'll run two copies of ChipScope, which will cause problems. Make sure that if there is a ChipScope already running that you switch to it using the task bar and don't run another copy.)

Another hint: sometimes ISE is pretty stupid about realizing whether anything has changed in your design. If you double-click on Generate Programming file and then double-click on Analyze Design With ChipScope it may re-synthesize and implement your design all over again. So you can also run ChipScope from the Start menu under ChipScope Pro -> ChipScope Pro Analyzer.

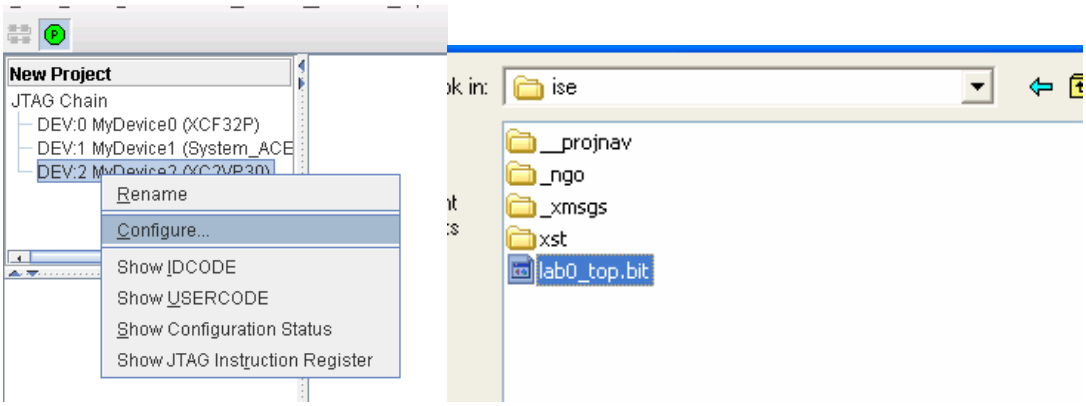
Once ChipScope is up and running click on the Open Cable/Search JTAG Chain button in the upper left. If your board is turned on and plugged in, ChipScope will shortly find it.

You may get a firewall warning. Just ignore this.

When it finds the board it will show you the three devices on the board. Just click okay.

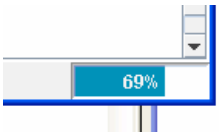


To program the device right-click on our device (XC2VP30, the bottom one) and choose configure.

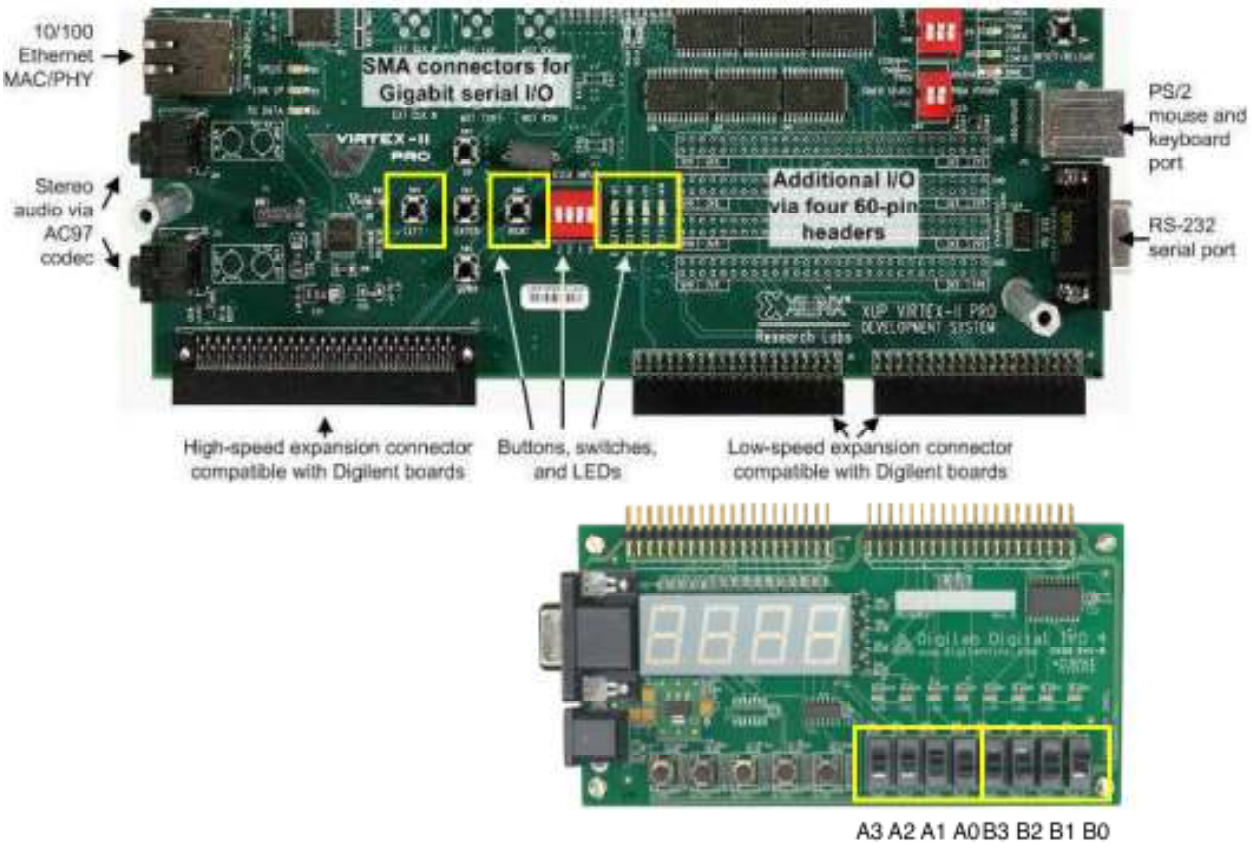


Tell it to load the lab0_top.bit file you created when you ran the Generate Programming File process. It will be located in the ise folder inside your lab0 folder.

Click okay and it will begin downloading. You can watch the progress in the lower right corner.



Once it's done downloading it's time to test out your design! Go ahead and play with the left and right buttons (on the FPGA board) and the 8 switches on the expansion board. You should see the results on the 4 LEDs to the right of the 4 small red switches on the FPGA board.



Does it work?

No? It doesn't?

Well, let's figure out what went wrong.

Put all 8 switches to the down (0) position. What is the output? Now play with them and see if you can see any patterns. (Ignore the left and right buttons for now.)

If you don't see a pattern, fill in this table by turning on the switches. (I.e., #4 is switch #4.)

Switches on (up)	LED3	LED2	LED1	LED0
none				
#8 + #4				
#8#7 + #4#3				
#8#7#6 + #4#3#2				
#8#7#6#5 + #4#3#2#1				

Now do you see a pattern? It is ANDing the inputs, but the output is inverted! When we have 87 and 43 turned on we have an input of A = 1100 and B = 1100, but our output is off/off/on/on instead of on/on/off/off. The reason for this is that I forgot to tell you that the LEDs are (and they always are) active low. That means that when you send a 1 to an LED it turns OFF and when you send a 0 to an LED it turns on.

Let's fix that and then figure out what's going on with the left and right buttons.

So go back to ISE and change the three places where you assign values to result to assign the opposite. I.e., instead of

```
            result = anded_result;
```

you will have:

```
            result = ~anded_result;
```

Don't forget the "no button" case.

Now save your file and re-run the Generate Programming File process. When it finishes go back to ChipScope and right-click on the FPGA device and choose Configure and re-load your design.(Now you'll begin to appreciate why it's preferable to do all your debugging in ModelSim…)

So how does it work now? With neither the left or right button pressed, do you get the correct ANDing of the inputs? (You should.)

Now we need to figure out what's going on with the left and right button. Do you think I might have forgotten to tell you that the buttons are also active low? How can you test that? Maybe by filling in this table? It has started it for you. For rows where a button input is a 1, the output will depend on the inputs A and B, so choose values for A and B that will result in distinctive behavior for AND and addition.

Left	Right	Left if Active Low	Right if Active Low	Expected Behavior	Actual Behavior
Pushed	Pushed	0	0	0000	0000
Not pushed	Pushed	1	0		
Pushed	Not pushed				
Not pushed	Not pushed				

If the buttons are active low, that means that when they are pushed they give a 0 and when they are not pushed the give a 1. So in the case where neither button is pressed, what do you expect your system to do? Well, if they are active low and you don't press either one, that means they will both be 1s. If we look at the always statement that handles the pushbuttons you'll see that if they are both 1 the first if statement is true, so the result will be the anded output. Ah ha. That makes sense!

Now to check it, what combination would you need to press to make it add? Does that work?

If it does, go back to your Verilog file in ISE and change the if statements to look at the inverted pushbutton signals. I.e., instead of:

```
if (LEFT_pushbutton == 1'b1) begin
```

You would have:

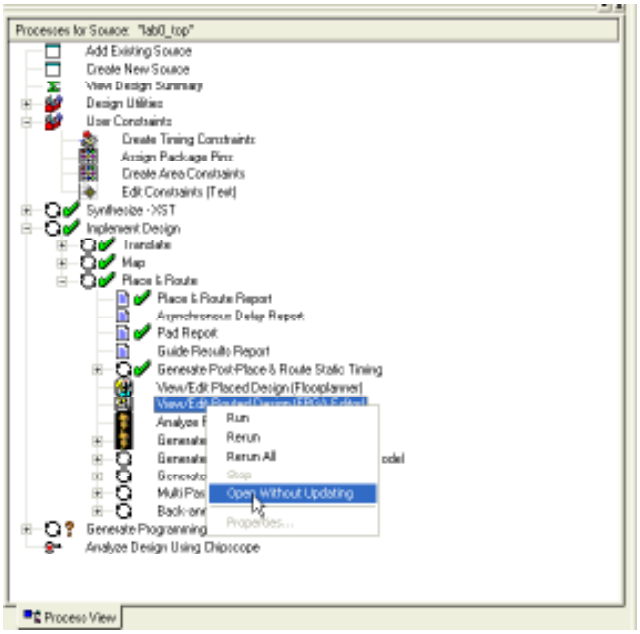
```
if (~LEFT_pushbutton == 1'b1) begin
```

Re-generate your programming file and re-download it. How are things now?

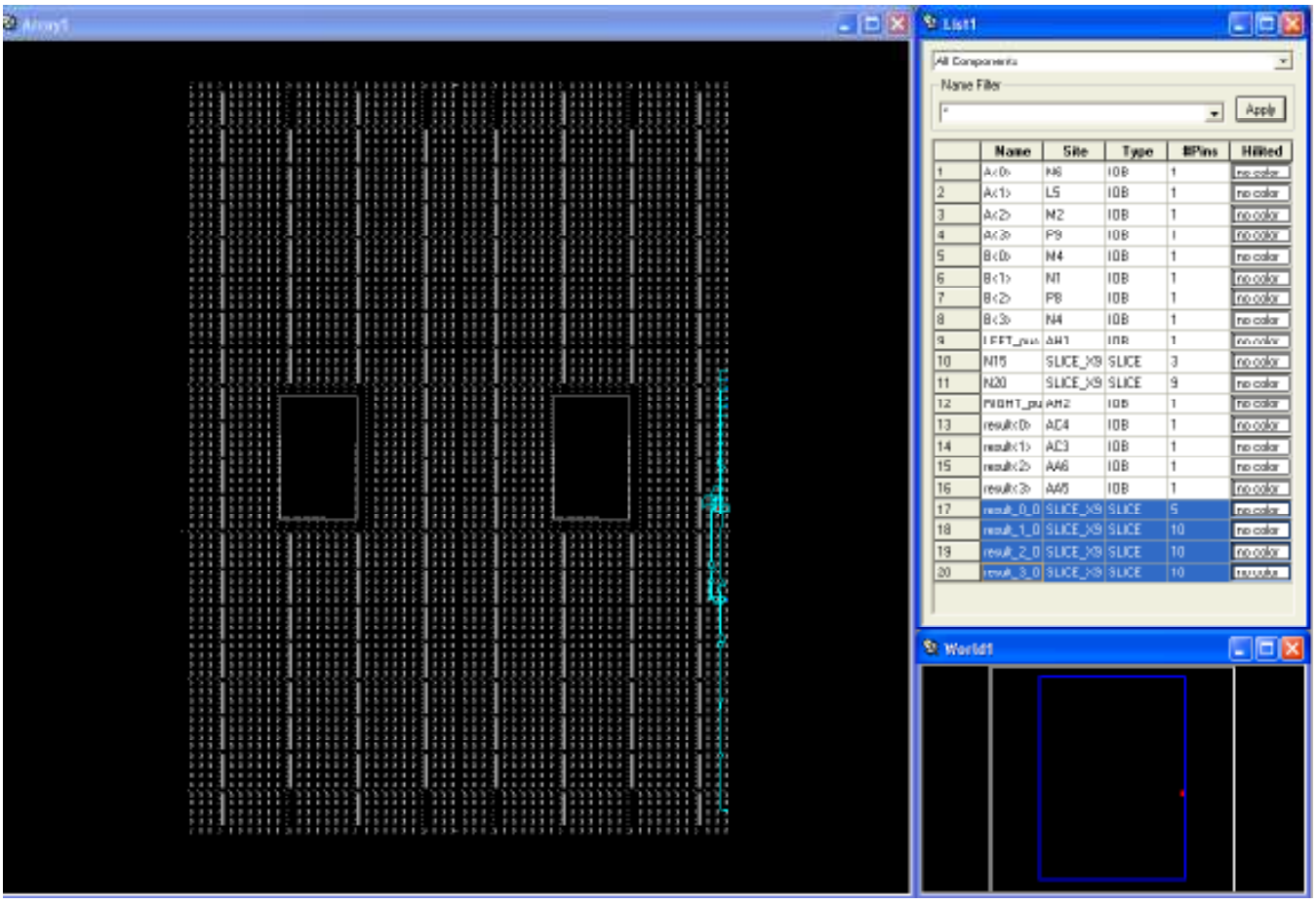
They should work at this point. If they don't it's time to talk to the TA.

What did you build?

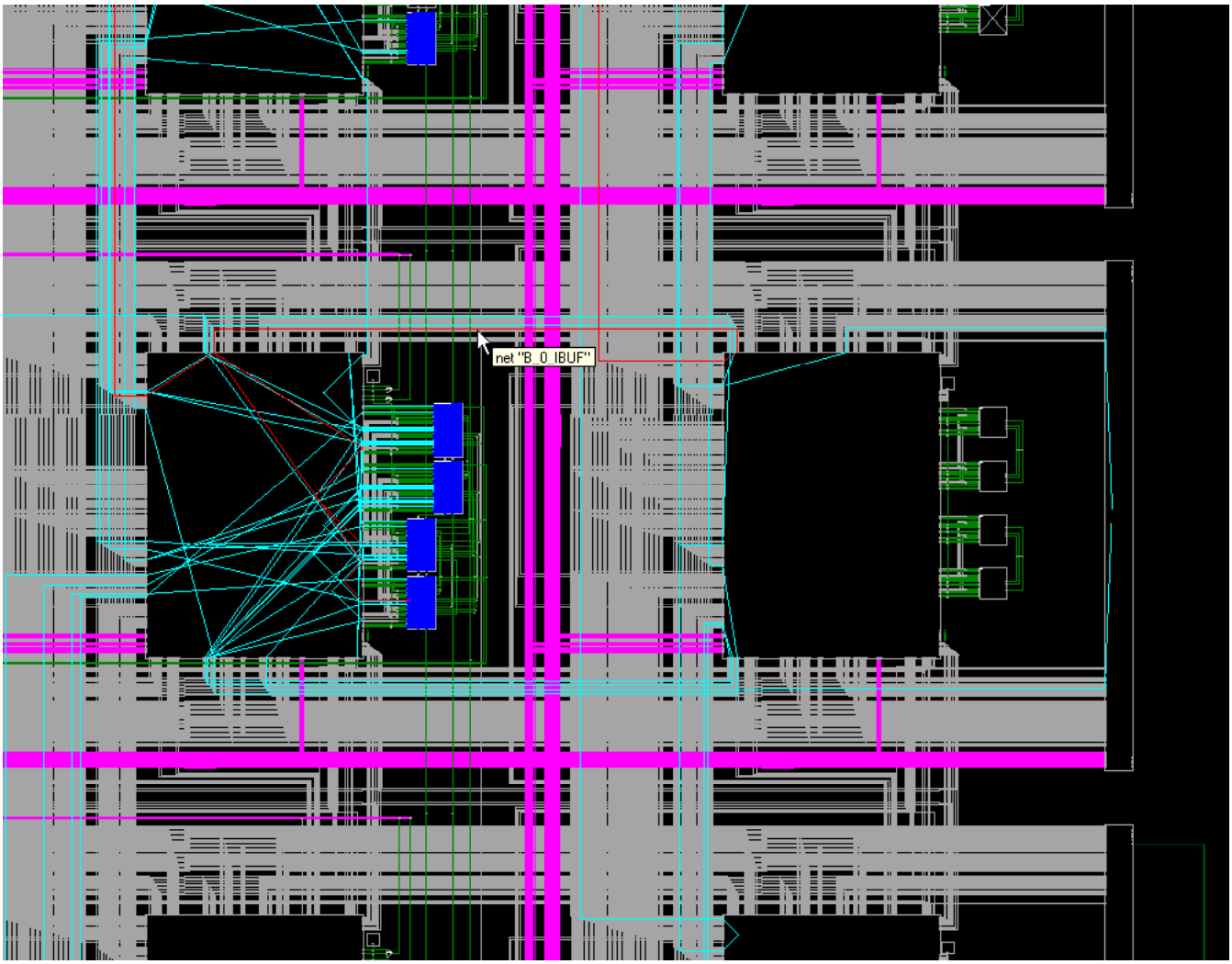
Before we leave let's take a look at what you actually built on the FPGA. To do this we will run the FPGA Editor, which is under Implement Design -> Place & Route -> View Edit Routed Design. Right-click and choose Open Without Updating.



The FPGA editor will show you a picture of the FPGA you are using and where your design fits on it it. When it first starts up you see the blank FPGA. The two boxes in the middle are the PowerPC processors on the FPGA. The array (the "A" in FPGA) is the grid of logic slices in the FPGA which you can program. Note that the list on the right shows your inputs (A<0>…A<3> and B<0>…B<3> as type IOB, or Input/Output Buffer) and your result signal result. Let's take a look at the result signal, so select result_0_0 through result_3_0 and click on the Routes button  .

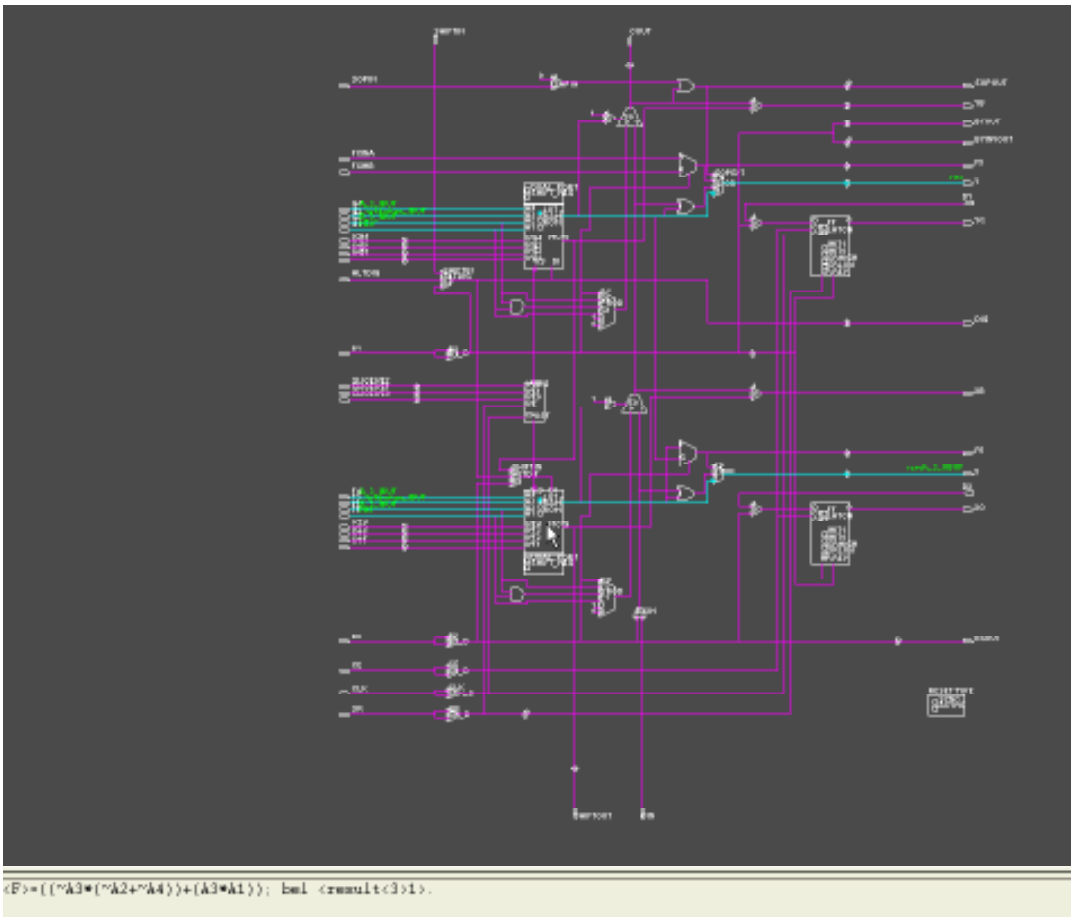


Here you see in cyan the wiring for your design. Not much, is it? The inputs are at the top, the logic is in the middle, and the outputs are at the bottom. To see your design up close click on the zoom to selection button,  then zoom out once.



Here you see the FPGA structure. The magenta wires are the long wires which go between every 8 or so logic blocks and the gray ones are the short wires that go between every two. Note that there are a lot more short wires than long wires. (How would you go between 3 blocks if all you have are 8 and 2 length wires?)

The big black boxes are the switches, and you can see your signals, in cyan, go through them to the actual logic blocks, which are in blue. Double click on one of those to see what is inside it.



Here we see what's going on inside the SLICE on the FPGA. We have two look-up-tables which actually implement the logic functions and a bunch of other programmable logic. When your design is compiled the logic equations are mapped to some number of these lookup tables (LUTs). If you click on one you can see the logic equation that it is implementing. In this case the bottom LUT is generating result_3, which is

```
<F>=((~A3*(~A2+~A4))+(A3*A1))
```

The important thing to take away from this is that most of the work in programming an FPGA is figuring out how to connect up the logic with the provided wires. Often you will see that your design spends more effort getting signals to the logic than it does actually running the logic. This is the penalty you pay to have a programmable device.

So that's it. You've now gone all the way through design, implementation, simulation, and synthesis of a simple project. Lab 1 is a bit more complicated than this, but not much. Lab 2 introduces multiple modules and hooking them up, and Lab 3 gives you finite state machines. By the time you get to Lab 4 you'll already be building a music synthesizer. Good luck!

This part is optional if you have time and your TA doesn't want to go home:

Instead of ANDing we want to implement an decoder. An decoder is a module that takes in a binary number and outputs a signal where the bit representing the binary number is true. I.e., if the input is 2, the output is 10, if the input is 3 the outputs is 100. The code for the module is as follows:

```
module decoder(
      in,
      out
      );

      input [3:0] in;
      output [3:0] out;

      reg [3:0] out;
      always @* begin
            case (in)
                  4'd0: out = 4'b0000;
                  4'd1: out = 4'b0001;
                  4'd2: out = 4'b0010;
                  4'd3: out = 4'b0100;
                  4'd4: out = 4'b1000;
                  default: out = 4'b1111;
            endcase
      end
endmodule
```

Now to instantiate it in your top level module you simply use:

```
      wire [3:0] decoded_result;
      decoder my_decoder(.in(A), .out(decoded_result));
```

And you hook it up to result instead of anded_result if the left button is pressed. Note that this only takes input from the A switches.

Does it work? If this was a real lab we'd expect you to write a test bench for this module as well, but not for lab 0.