Digital Design: A Systems Approach
Lab 0: *Introduction to Verilog and the Tool Chain*

Introduction/Objectives

Welcome to Lab 0. 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)
•**Icarus Verilog / iverilog** (a program that takes Verilog descriptions of digital circuits and **simulates** them to test for functional accuracy)
•**GTKWave** (a program that takes raw simulation dumps from Icarus Verilog and graphically displays them as waves changing over time), 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 iverilog and GTKWave 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 TA 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 applications[1].

[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 lab 0. We can think of this block diagram as describing a black box that has inputs and outputs.

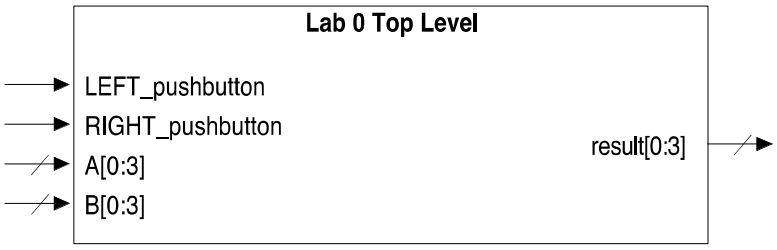

Figure 1: Top level block diagram of lab 0.

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


```
module lab0_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`. `lab0_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 "Lab 0 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`).

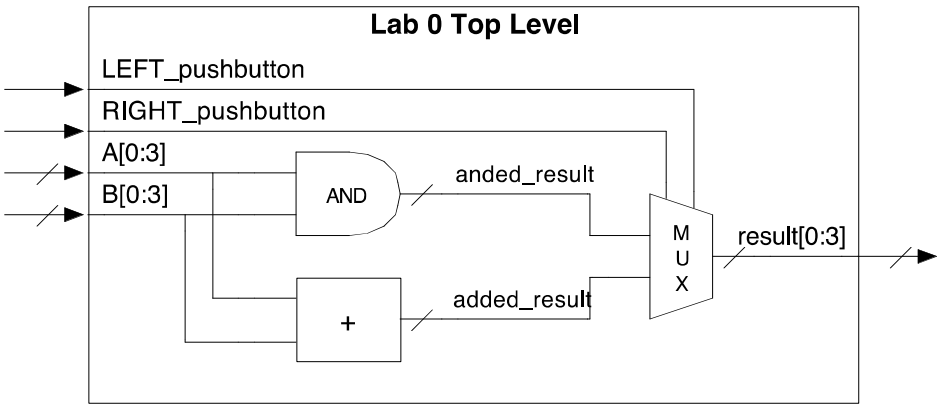

Figure 2: Block diagram of lab 0 top level with logic drawn into it. (Note this is for sake of example – in design descriptions you typically don't draw in the logic to the block diagram.)

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 `reg`s**, 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 `reg`s. There other blocks that do so as well, but we'll learn about them later, The `@*` part tells the simulator (iverilog) when to simulate the logic contained between the `begin` and `end`. Often times you may see something like `@(LEFT_pushbutton)` which tells iverilog 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_pushebutton` 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 lab 0 top module, now we just need to fire up ISE, write it, and see how it works by simulating it.

Step 2: Entering your design in ISE

The following will guide you through a step-by-step procedure to open the design tools we will use. For those of you who are familiar with Linux, Unix, or even Mac OS's command line, this may be painfully slow, but bear with us as we bring everyone up to speed.

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 from anywhere with an internet connection, but will have more on that later.

Now, we'll open ISE. To do so, simply double click the ISE Design Suite 14.7 icon on the desktop.

Once ISE opens, create a new project for lab 0 by going to File → New Project…

First, under location, click the ellipses to open up a standard windows dialogue to pick a folder for your projects. **Remember which directory you've chosen. You will need to move files onto external storage and delete them from this system to ensure their security.**

Then, fill in the name of the project, `lab0`. The location and working directory should get automatically updated as in Figure 5.

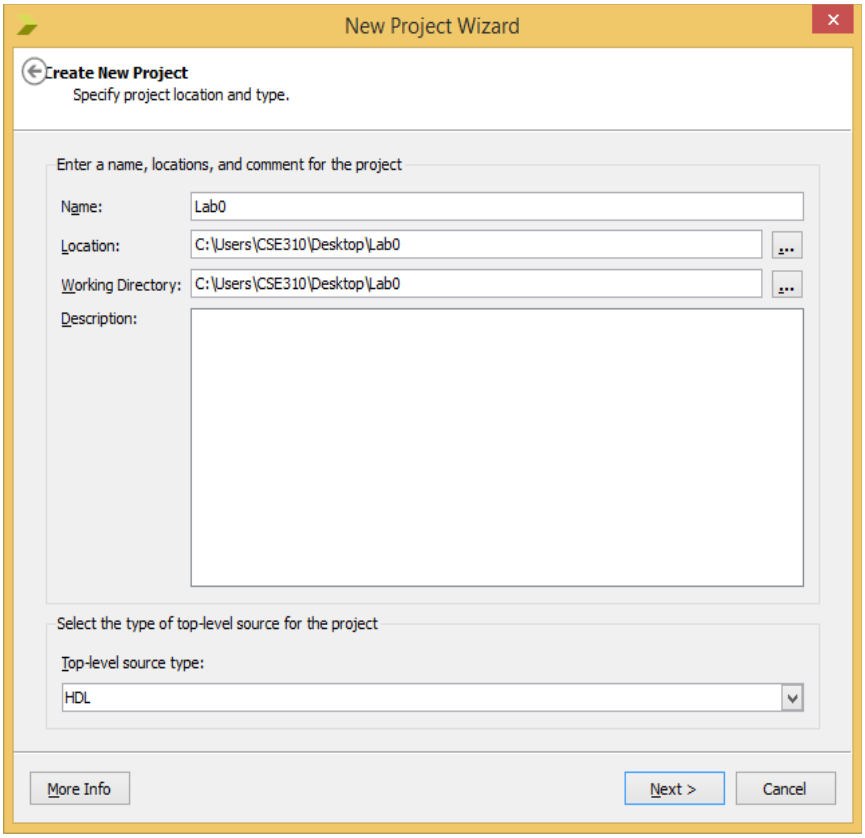

Figure 5: *Project creation*

Click **next**, and change the options in ISE to match the ones shown in Figure 6. This is where you tell ISE which exact chip we have and what language you're using. Click **next** and then **finish** to complete the new project wizard.

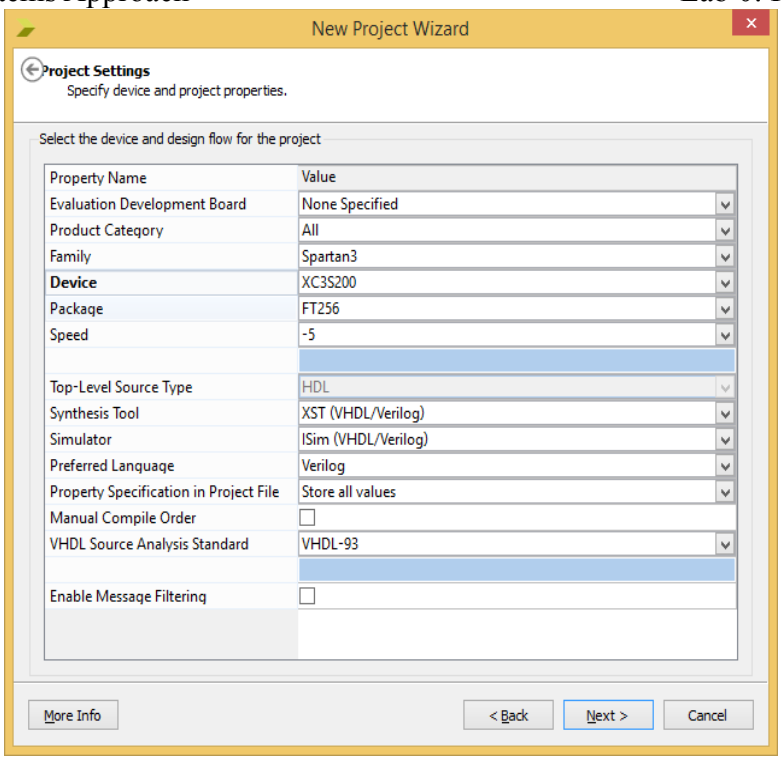

Figure 6: *Important project settings*

You will now be at the ISE main screen. There is a list of all the modules in your project in the upper left (called the *design hierarchy*, although there is nothing there yet), a transcript window on the bottom, and a space on the right for displaying reports (Figure 7).

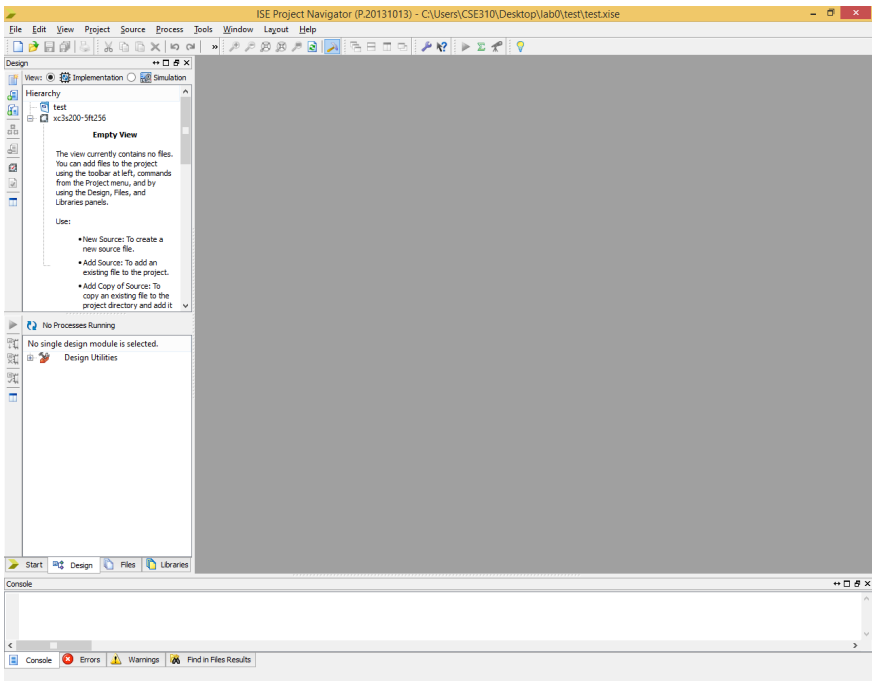

Figure 7: *A fresh project*

The next step is to create our Verilog files. Click the top menu Project → New Source. Select 'Verilog Module' form the listbox on the left and type 'lab0_top.v' in the file name box. In general, it's a good idea for a Verilog file to only contain one module in it, of the same name (Figure 8). Click **next**.

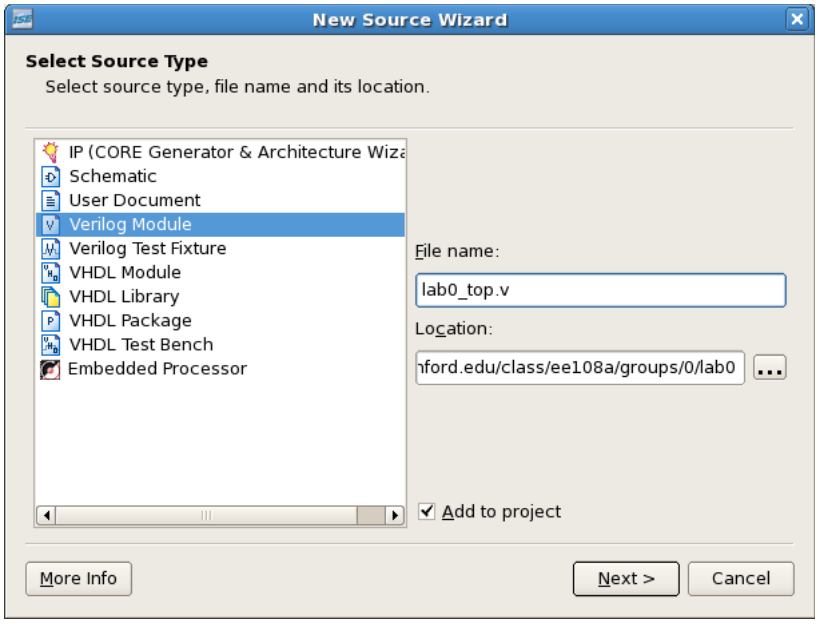

Figure 8: *Creating a new Verilog module*

Leave the "Define Module" screen blank by clicking 'Next', and then click 'Finish'. In the future this screen will allow you to quickly specify your module's inputs and outputs, but for lab 0 we're going to do it ourselves.

An instance of the xilinx editor should automatically open up to edit your new Verilog file. If the syntax is not colorfully highlighted (Figure 9), you can make it so by clicking View → Highlight Mode → Sources → Verilog.

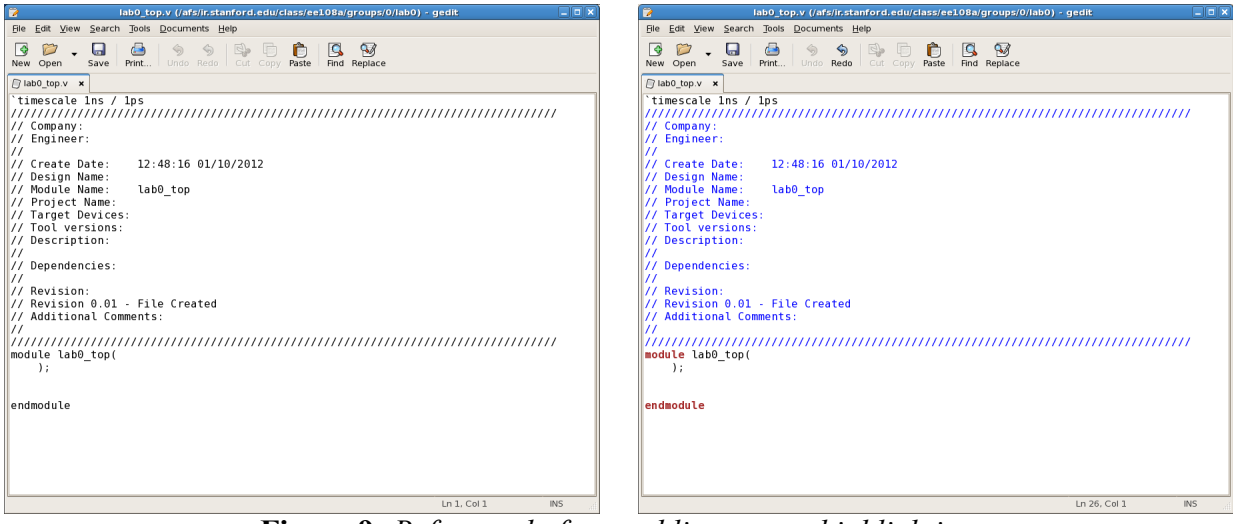

Figure 9: *Before and after enabling syntax highlighting*

Now enter the Verilog code for your module from above. Careful if you're cutting and pasting! Oftentimes the ' and " symbols in formatted documents like this one are the "smart" directional variants (such as those around the word "smart"); Verilog will choke on those!

Once you're done, save and close the window. ISE will automatically detect syntax errors when you close the editor, and report them in the Transcript panel at the bottom (Figure 10). To edit the file and fix these errors, you can click the link provided in the error or double-click the problematic file in the Design Hierarchy in the top left.

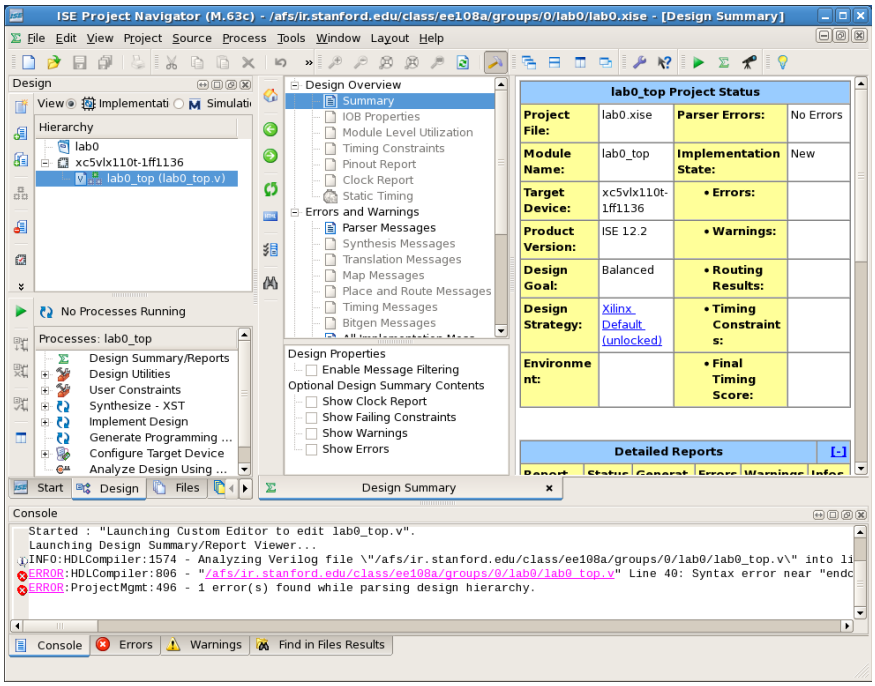

Figure 10: *ISE reporting a syntax error. Here, we forgot a semicolon at the end of a line.*

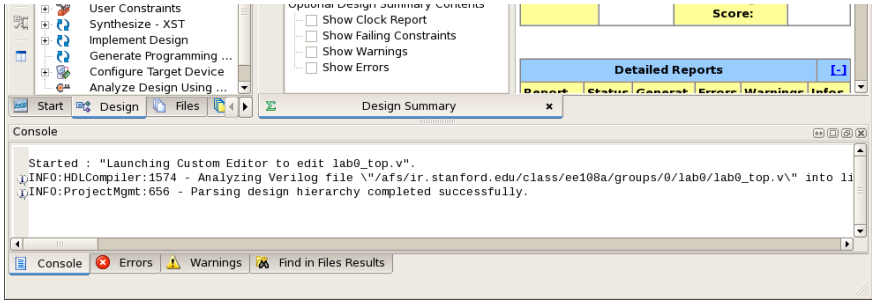

Figure 11: *ISE reporting all is well*

If all is well, you should see "Parsing design hierarchy completed successfully" in the transcript window (Figure 11).

Writing a Test Bench

In order to simulate our module, we need to make a "test bench"—a Verilog module which drives our module-to-be-tested's inputs and receives its outputs. The module we are testing (in our case lab0_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 possible 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 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. 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 iverilog, 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 1.84×10^{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. First of all, since we are using ISE to both simulate and synthesize (compile for the FPGA) our design, we have to first switch it into simulation mode. Click the Simulation view button at the top of the Design pane. Make sure the dropdown box below it says 'Behavioral' (Figure 12).

Next, create a new Verilog file called lab0_top_tb.v in the same way as before (Project → New Source...). It is possible for ISE to generate a test bench skeleton for you by selecting "Verilog Test Fixture," but for lab 0 we're going to do it from scratch.

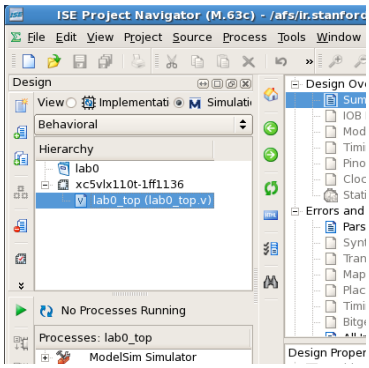

Figure 12: *Switching ISE into simulation mode*

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.)

As before, double-click to edit the file in gedit.

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 lab0_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 `lab0_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 `reg`s since we are going to define them in a combinational block and the signals coming out are `wire`s. Signals coming out of modules are always `wire`s. 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 `lab0_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.)

```
lab0_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 `lab0_top`, and then we named it `dut`. This is because even with the same module, we can instantiate multiple copies of `lab0_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 `lab0_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 `lab0_top` instantiation. You could also use the following (again, **NOT suggested)**

```
lab0_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 (sometimes an error) when you simulate, 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 `lab0_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 log
   $display("Output is %b, we expected %b", sim_result, (4'b1100 & 4'b1010));

   // stop simulating
   $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).

The 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).

Save your testbench Verilog and quit gedit. Notice the hierarchy has changed, with lab0_top_tb now as the top-level module (Figure 13). This means the testbench module instantiates lab0_top. Let's try running the test bench.

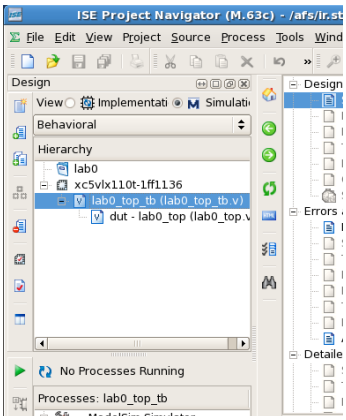

Figure 13: *The new design hierarchy*

Highlight lab0_top_tb in the hierarchy panel, since this is the testbench we want to run. Expand 'ModelSim Simulator' in the processes panel, and double click 'Simulate Behavioral Model' (Figure 14). Be sure to always select a testbench; if you select a module by itself, you will either get errors or an empty simulation result. While we're using iverilog and GTKWave for simulation, the programs are in fact being run by a special script that pretends to be ModelSim. That is why ISE will always refer to it as ModelSim; but it should prove obvious that we're not using that.

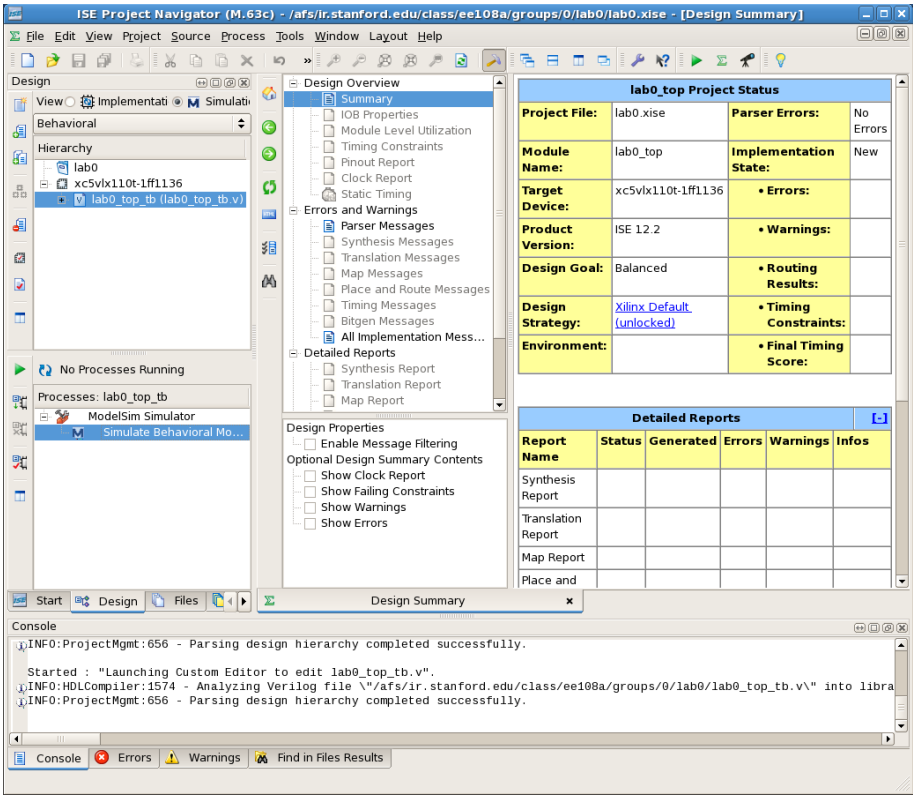

Figure 14: *Starting the simulation*

Since our simulation is short, everything will happen very quickly. Before you can say "domino logic," three windows will have popped up: the iverilog simulator (Figure 15), GTKWave's wave display window (Figure 16), and GTKWave's log viewer (Figure 17). The first, titled 'ISE Verilog Simulation,' is a terminal that will report any syntax or simulation errors iverilog encountered while running your testbench. This terminal is color coded: errors in red, tips in green, file names in orange, and simulation

output (such as $display statements) in blue. The GTKWave log viewer also displays the same simulation output.

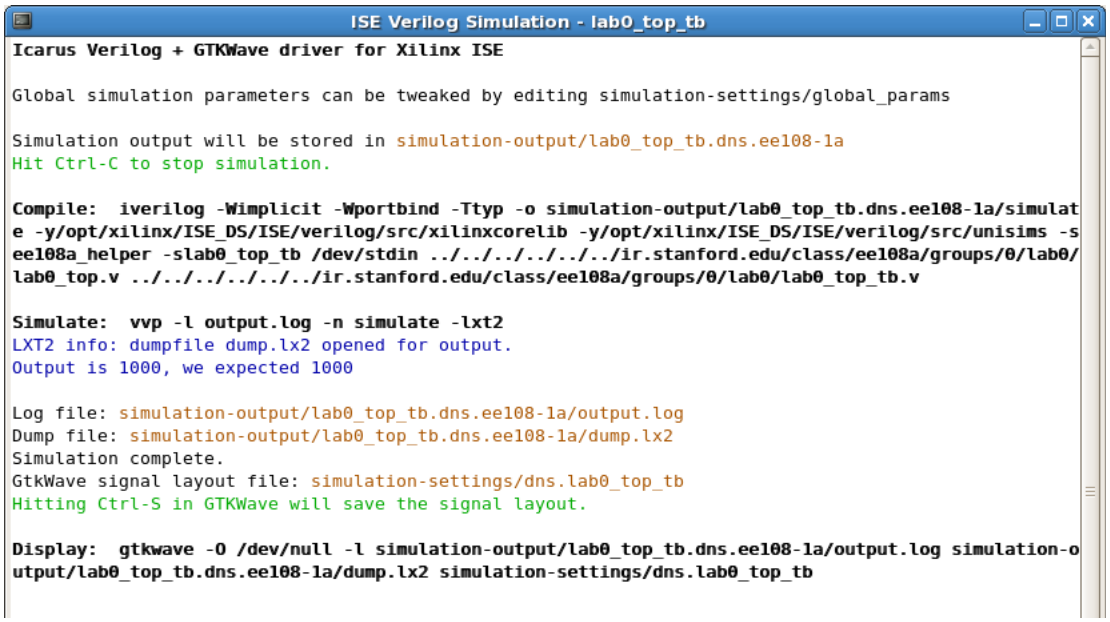

Figure 15: *Icarus Verilog*

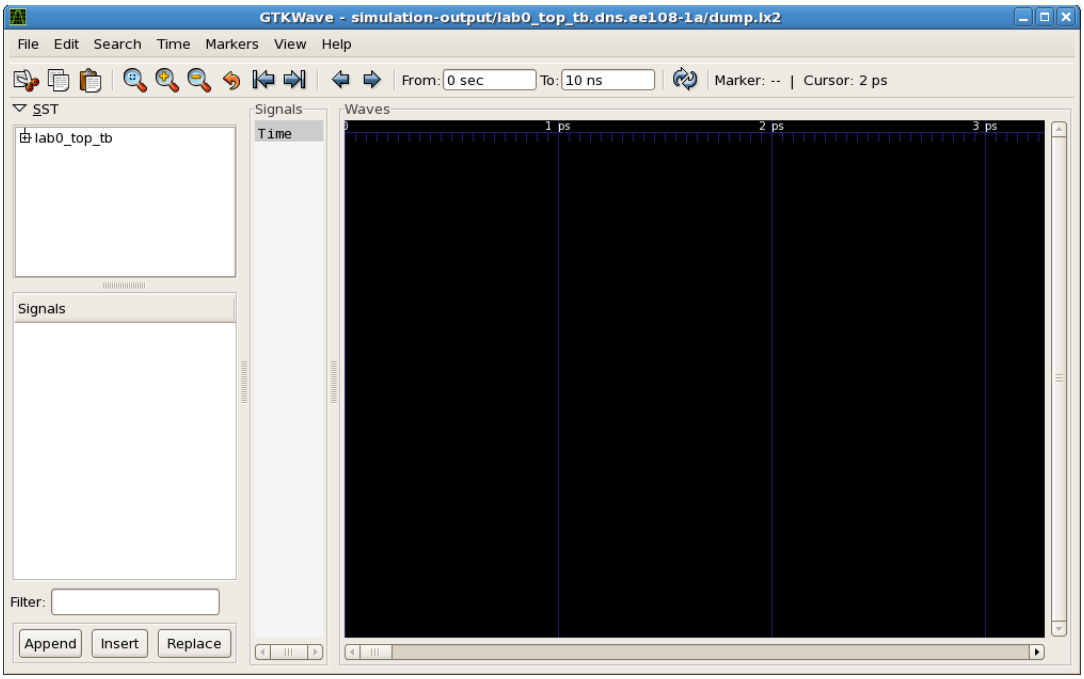

Figure 16: *GTKWave wave window*

While this simulation was quick, some simulations can go on for far too long, in which case you will spend a lot of time in-between the “Simulate” and “Display” steps. In that case, if you want to manually stop the simulation and look at the output, you can hit control and C at the same time.

Now that our simulation is done, we want to view what happened to the individual signals. For that, we use the opened GTKWave wave viewer. To view signals from our design, we need to use the “signal search tree” (SST) on the left to find everything. You can explore the hierarchy in the SST, and

highlighting a module will show its internal signals in the pane below. For example, highlighting the lab0_top_tb module will show the testbench signals, while expanding the testbench and highlighting the `dut` module will show the top module instantiation's internal signals (Figure 18). If your module has many signals, you can use the filter box to narrow down the signals shown.

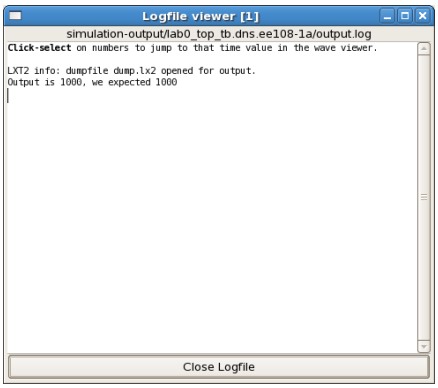

Figure 17: *GTKWave log viewer*

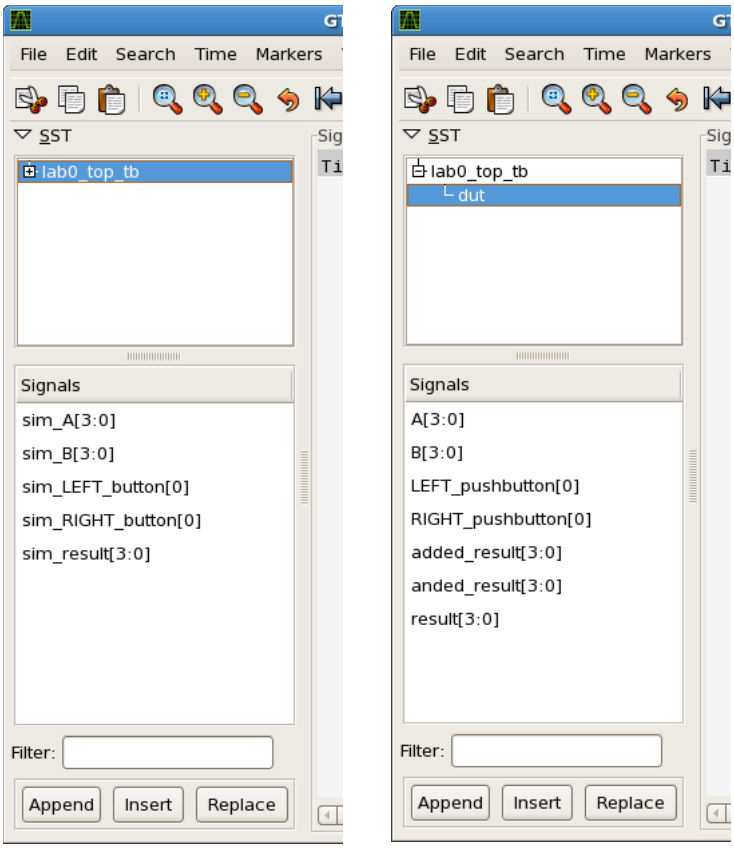

Figure 18: *Showing internal signals*

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. For each module, select the signals we want to view and hit **append** to add them to the end of the signal viewer. GTKWave should now look something like Figure 19. Note that the wave viewer shows a graph of signals vs. time. Our test bench only simulated for 10 time steps before hitting the `$stop` we inserted

(we have two delays of 5 timesteps each), but later on we'll get much longer test benches!

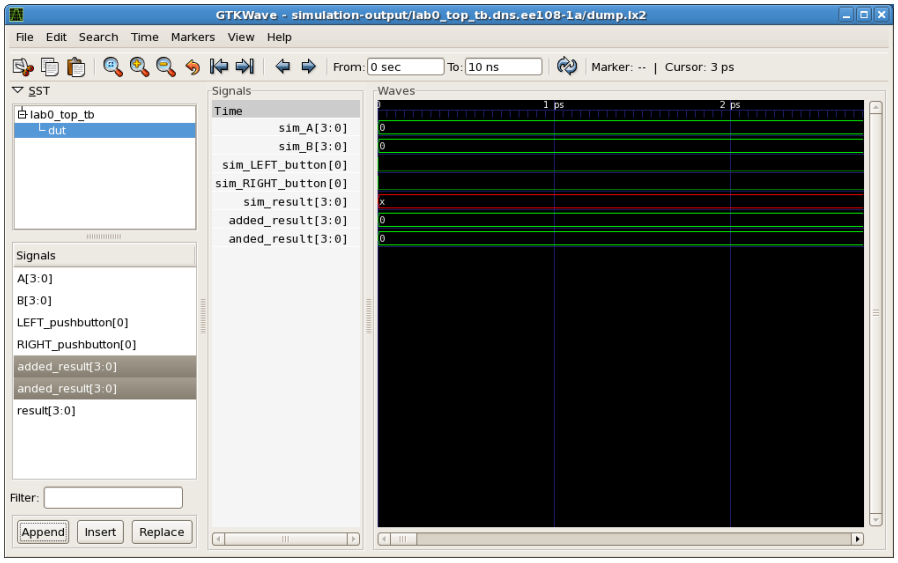

Figure 19: *GTKWave with signals added*

But wait, why does everything appear to be constant? We're actually quite zoomed in. You can fix this by hitting the "zoom full" toolbar button (the leftmost magnifying glass), or using the shortcut key Alt+F. Now you can actually see the exciting part of the testbench, as in Figure 20.

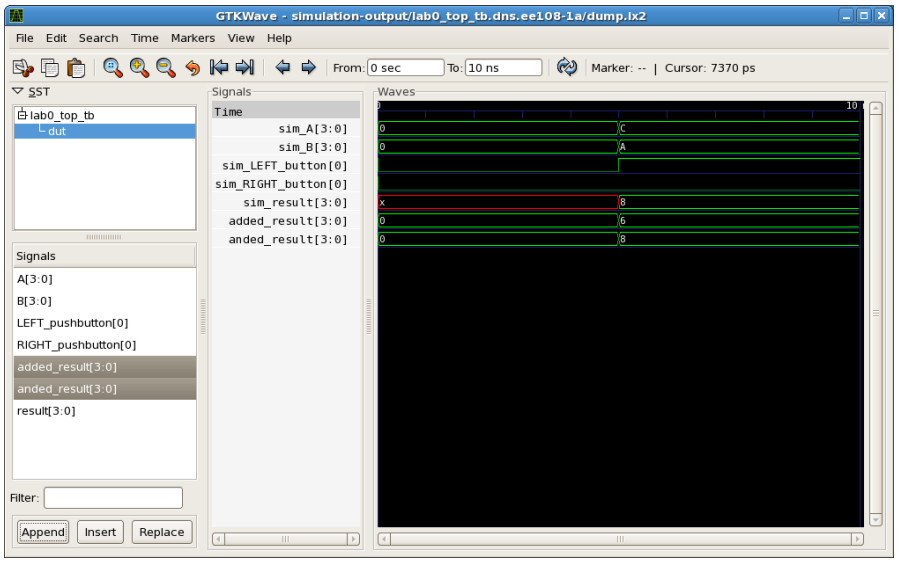

Figure 20: *Zoomed out on the action*

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. The values of the signals are displayed as hexadecimal digits. However, since we're doing binary logic, perhaps it'd be easier to verify the correct functionality if the numbers were displayed in binary form. To do so, highlight the signals in the wave viewer that you want to be shown as binary, then right-click one of them and choose Data Format → Binary. It'll turn out like Figure 21. That's a bit easier to read, isn't it? As you can see, there are a lot of display formats available; always be sure to choose the one that makes the most sense for the signal, as it makes

the waveforms that much easier to understand.

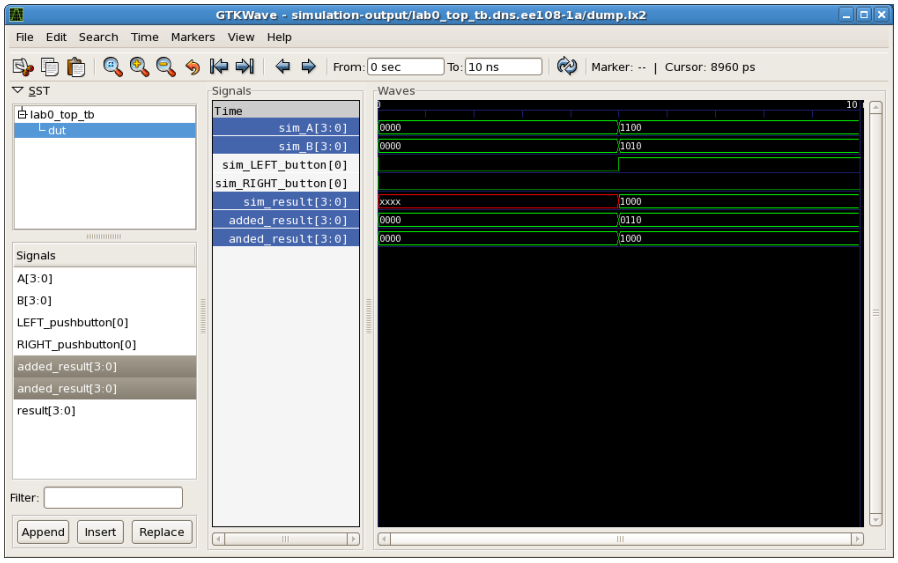

Figure 21: *Displaying signals as binary numbers*

This testbench by no means rigorously tests our module, so we're going to want to add more test cases. But first, so that we don't have to set up the wave view again, choose File → Write Save File, or just hit Control+S. The next time you run this simulation, GTKWave will pop up with the display configuration as it was at the point when you last saved it. Go ahead and close GTKWave.

Let's now 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.

We need to re-run the simulation since we've changed the files involved, but we don't need to close iverilog to do that. Notice that when you closed GTKWave, iverilog asked if you wanted to re-run the

simulation? If you left that window open, then once you've changed the testbench, you can just type **y** and hit enter to run the simulation again (Figure 22), bypassing the need to use ISE entirely for quick fixes. If you already closed the iverilog window, you can of course run the simulation again as we did before.

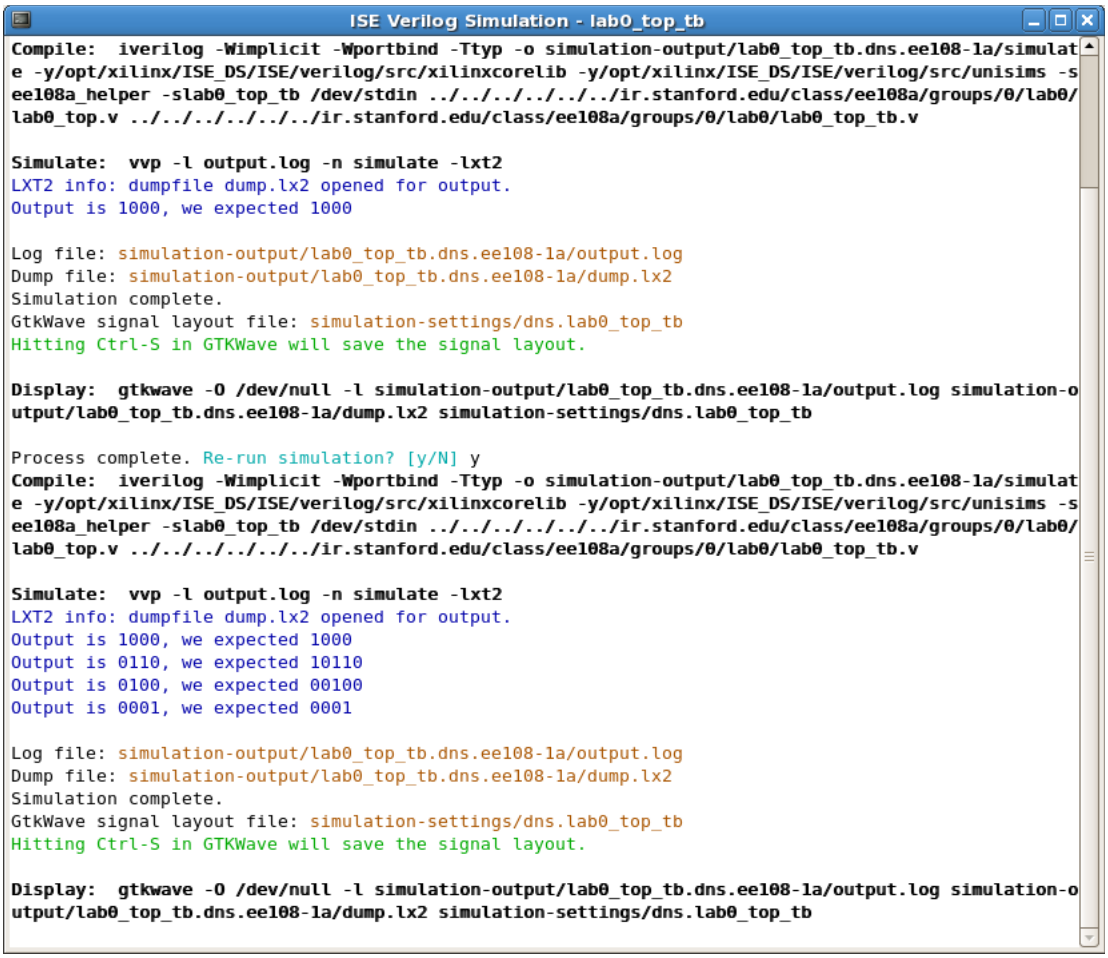

Figure 22: *Quickly re-running the updated simulation*

As you can see in Figure 22, the log shows the new tests. Some of the expected values are printed as 5 bits; this is because iverilog will never drop precision (bits) in intermediate values (such as the ones we were giving to `$display`). It will only drop the bits if you assign the value to a reg or wire that has fewer bits, and in that case it will always drop the most significant bits (MSBs). Thus, since the four least significant bits are the same in the printout, we can see that the output is as expected. We can also quickly confirm this in the waveform, since GTKWave popped up with the same signals displayed that we saved before (Figure 23). You may have to hit the "zoom fit" button again to accommodate the longer testbench.

Here's an important comment: How easy is it for you to understand the wave diagram below vs. the text above? 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, such as done in Figure 24. To help with annotating, you can either take a screenshot of the window, or use GTKWave's File → Print To File command.

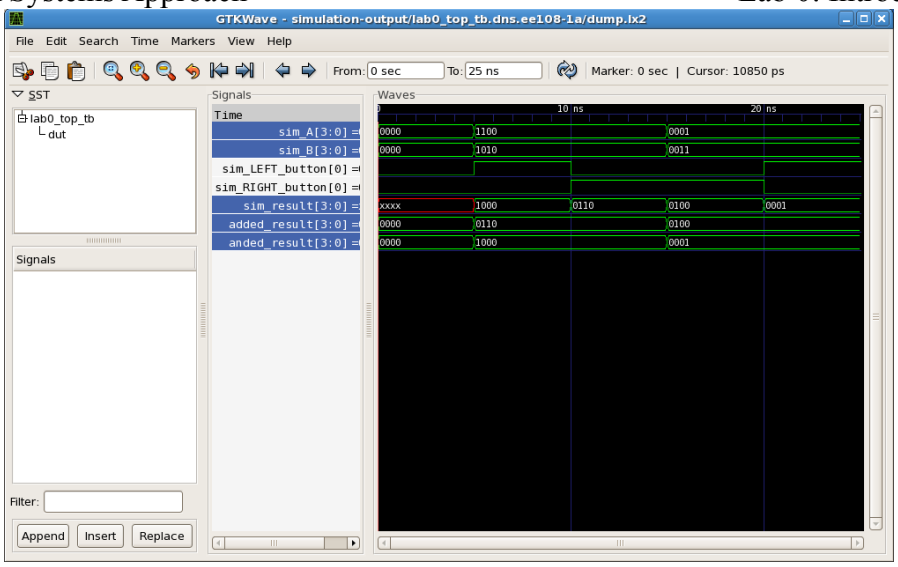

Figure 23: *The fancier testbench waveform*

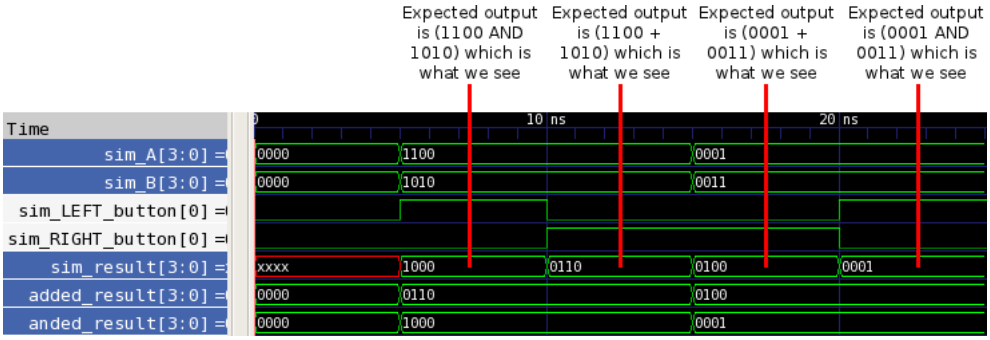

Figure 24: *Annotated version of the above waveform, made using the same screenshot and GIMP*

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. Towards the last few labs you'll find it less useful so we'll give you a break and not require annotating, but you will still need to understand what's going on.

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 FGPA. Since we're already set up in ISE for simulation, we only have to do a couple steps to synthesize our design. We have our top module, but instead of a testbench to instantiate the top module, we'll instead need one important file: the **UCF**, or "user constraints file". This file is what will tell us which physical pins on the FPGA connect to specific inputs or outputs from our top module (i.e. on what wire is the actual left pushbutton hooked up to which we'll then call, drum roll, `LEFT_pushbutton`). In essence, it's telling ISE how to "instantiate" the top module on the FPGA hardware itself. The testbench module, on the other hand, cannot be synthesized, as it uses an `initial` statement which is not synthesizable.

The UCF file for this lab is located on Blackboard: **lab0.ucf**

Now, in ISE, click Project → Add Source..., and find the UCF file in your group's lab0 folder. Alternatively, instead of separately copying the UCF file in the terminal, ISE can copy it for you by using Project → Add Copy of Source..., in which case you would have to find and select the UCF file in the lab starter files directory. Either way will bring up the "Adding Source Files..." window (Figure 25).

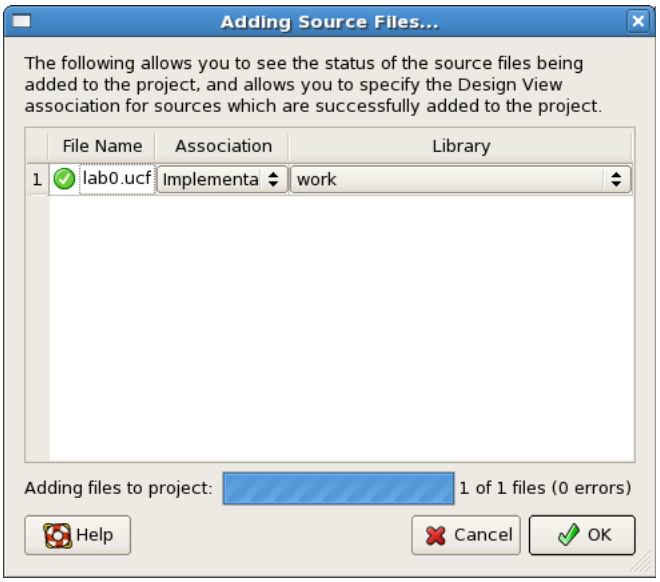

Figure 25: *Adding the UCF file to the project*

See how the *association* is "implementation"? This means that the UCF file will only be used when you synthesize your design. If you were to add a test bench in this manner, the association would be "simulation", since it cannot be synthesized. Module implementations, since they are both synthesizable and simulateable, will have "all" as their association. *Library* should always be left as "work."

Hit OK to add the UCF file. Since we're still in ISE's simulation mode, go ahead and switch the view to "implementation".

If you created the testbench as a normal Verilog module (as this handout told you to), you may find that even in "implementation" mode, `lab0_top_tb` is the top module. This is not correct! As you know, testbenches are *not* synthesizable, but since ISE didn't realize you were making a testbench when you created the module, it went ahead and set its association to "all." You can fix that by right-clicking the offending module and choosing "Source Properties." You can then fix its association to be "simulation" as in Figure 26, and then hit OK.

Now that the associations are correct, you should see the `lab0.ucf` underneath `lab0_top` (Figure 27), which is now the top module for implementation.

You can double-click the `lab0.ucf` file to see what's in this file (Figure 28). As you see it just defines the inputs and outputs in terms of the pins they correspond to on the FPGA. Most of the definitions are missing, however. It's your job to define the missing parameters by using the labels on your FPGA. **Use (Figure 39) on the last page as a reference for which switches and buttons to use.**

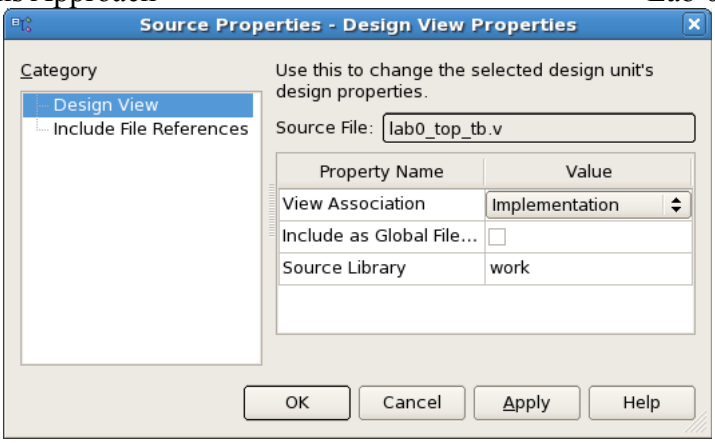

Figure 26: *Correcting the testbench's association*

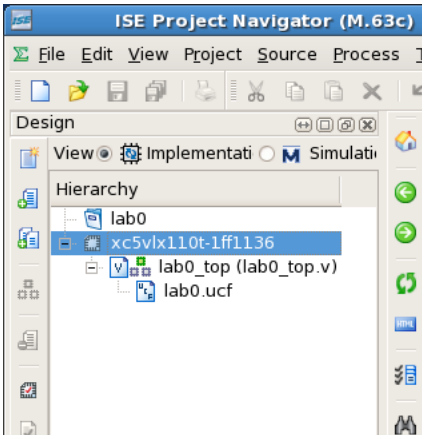

Figure 27: *Implementation mode, with the UCF file added*

lab0.ucf

```
##SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
##----------------------------------------------------------------------

##----------------------------------------------------------------------
## Switches on the SPARTAN3 Board
##----------------------------------------------------------------------
NET "A<3>" LOC = "K13";
NET "A<2>" LOC = "K14";
NET "A<1>" LOC = "J13";
NET "A<0>" LOC = "J14";
NET "B<3>" LOC = "";
NET "B<2>" LOC = "";
NET "B<1>" LOC = "";
NET "B<0>" LOC = "";

##----------------------------------------------------------------------
## Push buttons on the SPARTAN3 Board
##----------------------------------------------------------------------
NET "LEFT_pushbutton" LOC = "";
NET "RIGHT_pushbutton" LOC = "";

#----------------------------------------------------------------------
# LEDs on the SPARTAN3 Board
#----------------------------------------------------------------------
NET "result<3>" LOC = "N14";
NET "result<2>" LOC = "";
NET "result<1>" LOC = "P14";
NET "result<0>" LOC = "";
```

Figure 28: *Contents of the UCF file*

Now let's synthesize our design.

When you've completed the UCF file, select the lab0_top module (which is the default top-level module) in the Sources pane and double-click on Generate Programming File in the Processes pane (Figure 29). Note: ISE 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.

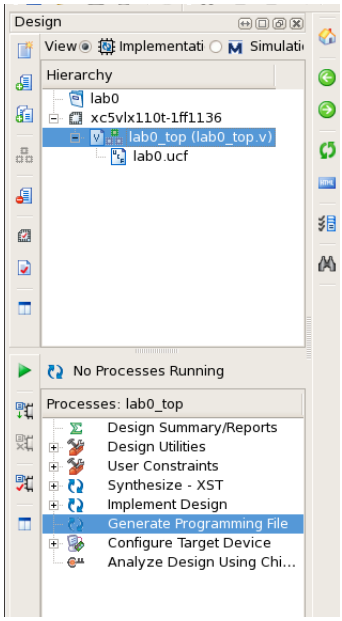

Figure 29: *Synthesizing the top module*

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 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.

Let's check the synthesis results. If you wrote lab0_top precisely as you were told to in this handout, then while the bottommost pane says that "Generate Programming File" completed successfully, and the errors tab is empty, there should be exclamation points next to the "Generate Programming File" and "Synthesize" entries. This means warnings were raised in the two steps. Clicking the Warnings tab will show what the warnings were (Figure 30).

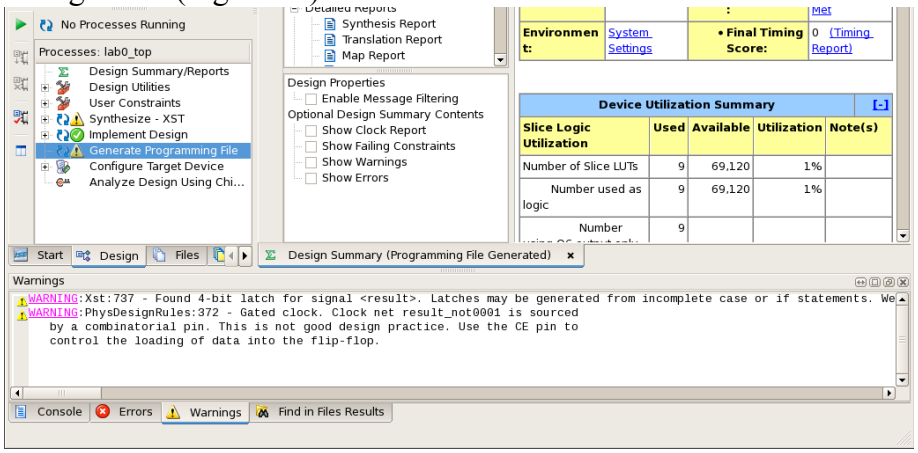

Figure 30: *Warning! Inferred latch!*

Oh no! ISE found a 4-bit latch for signal `result`! 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 (example in Figure 31).

```
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.
```

Figure 31: *More information about the inferred latch*

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
    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 a default case at the end which defines the value of `result` when no button is pressed.

```
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
```

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 **x**'s for `sim_result` before the `LEFT_button` goes high back in Figure 24. The **x**'s and the red box mean that 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**'s) in GTKWave, 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! If we were to simulate the design with the above added lines, we would see that now the **x**'s have gone away (Figure 32).

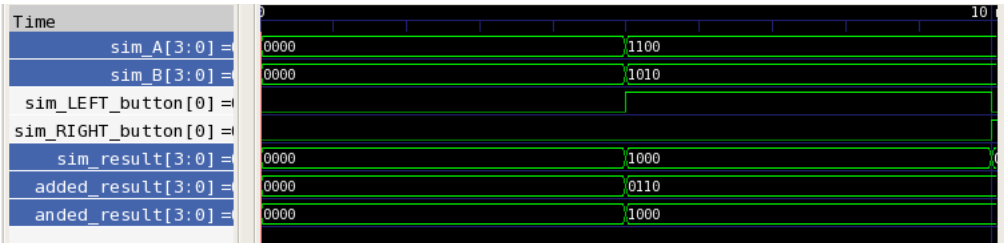

Figure 32: ***x**'s resolved;* `sim_result` *is defined even before* `sim_LEFT_button` *goes high.*

This time you should have succeeded in synthesizing your design with no errors and earned lovely green checkmarks next to the steps. You're now ready to download it to the FPGA board and test your design.

To download your design to the board, double-click on the "Configure Target Device" menu item (Figure 33). This will launch a program which will communicate with the FPGA boards. You may get a warning saying that there is no iMPACT project file, but just click OK.

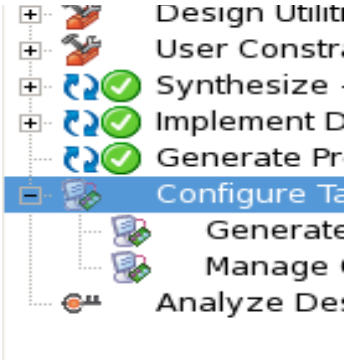

Figure 33: *Green checkmarks and launching iMPACT*

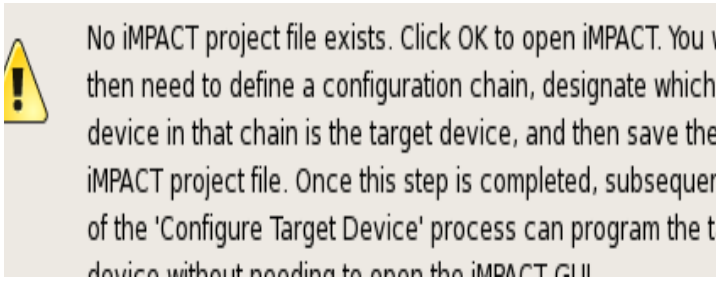

Figure 34: *No iMPACT project file exists. Just hit OK.*

Once open, you should double-click on the "Boundary Scan" item in the iMPACT Flows pane (Figure 35). Next, right click on the iMPACT screen where it says "Right click to Add Device or Initialize JTAG chain", and click "Initialize Chain". Click "No" in the "Auto Assign Configuration Files Query Dialog" and "Cancel" in the "Device Programming Properties – Device 1 Programming Properties" windows when they pop up. **Please remember to turn on FPGA board and make sure the JTAG ribbon cable is connected, otherwise you will get errors saying "many unknown devices detected".**

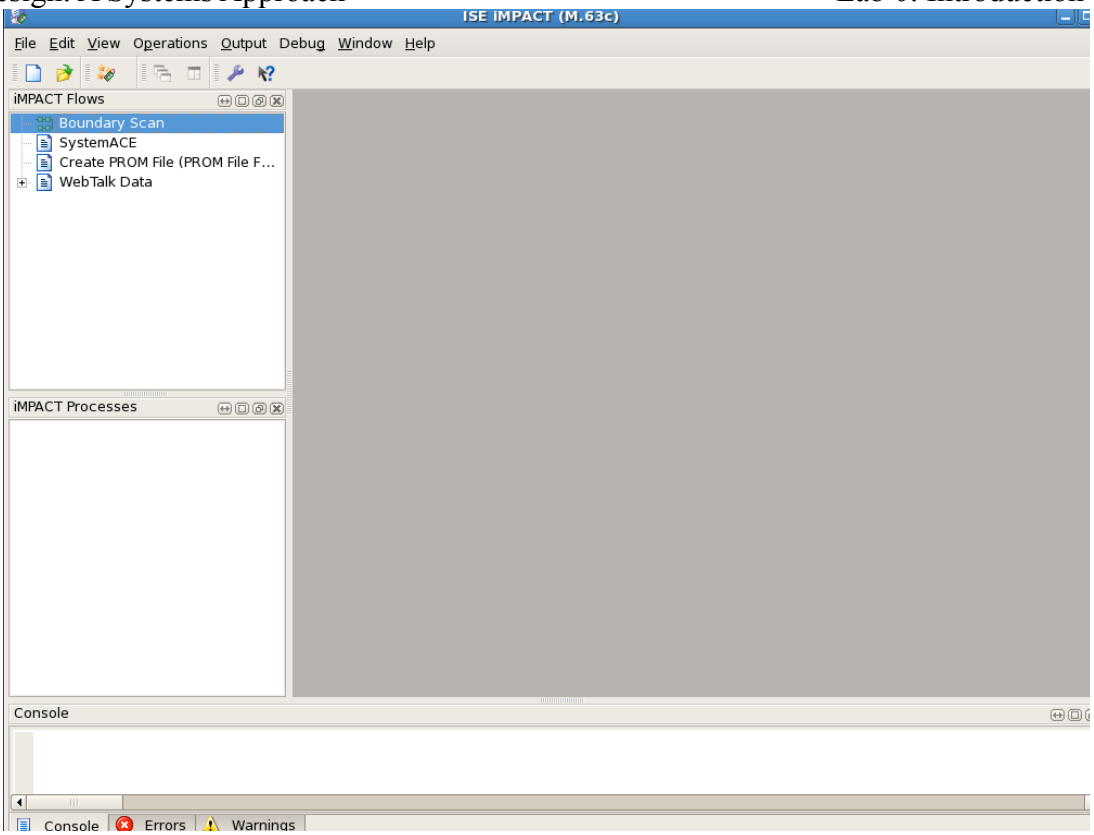

Figure 35: *iMPACT*

Now you should see something similar to Figure 36. Now you should right click on the device that is labeled "xc5vlx110t" and select "Assign New Configuration File". Navigate to and select the "lab0_top.bit" file that sits in your lab0 directory in the browser that appears and click open. When prompted, do NOT attach SPI or BPI PROM to the device (Figure 37). Once you do this, right click on the device labeled "xc5vlx110t" and click "Program". Click on OK in the Device Programming Properties – Device 5 Programming Properties" window that appears. You should then see a wait bar telling you about the programming progress.

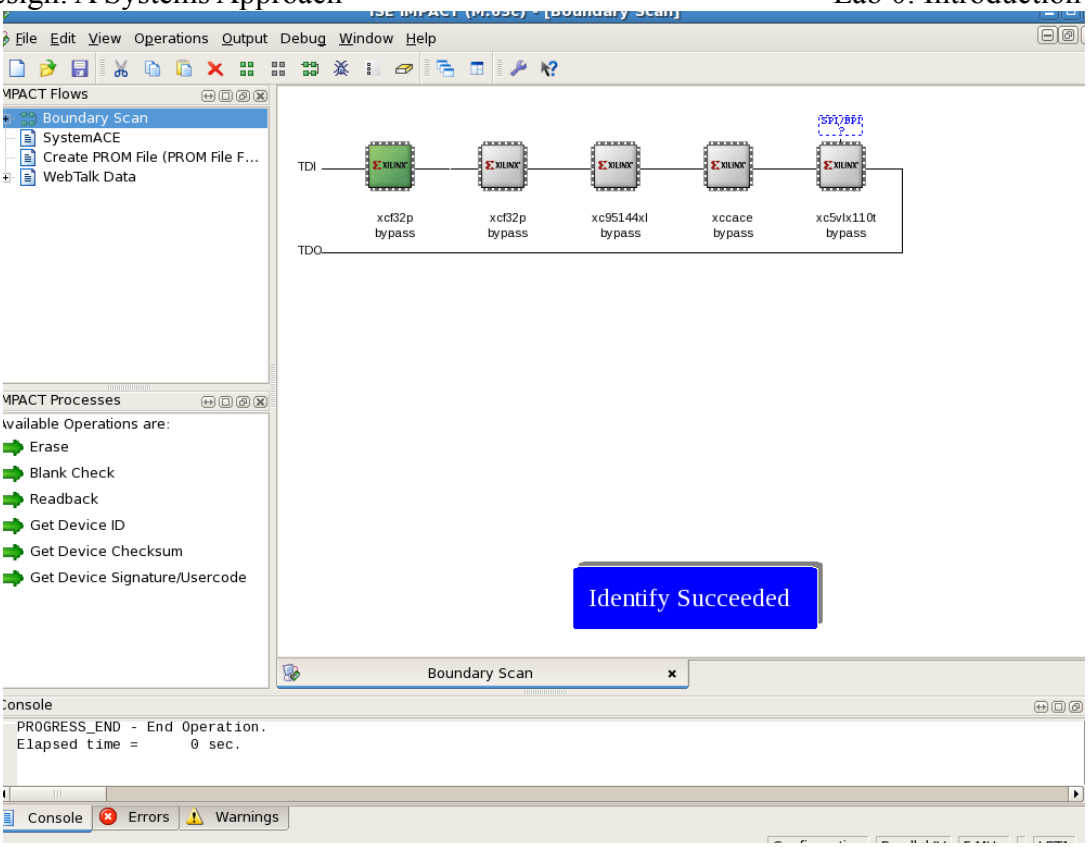

Figure 36: *Successful JTAG initialization*

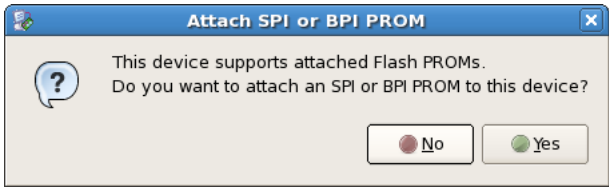

Figure 37: *Just say NO.*

Figure 38: *Program Succeeded*

Once it's done downloading (Figure 38) it's time to test out your design! Go ahead and play with the left and right buttons (on the FPGA board – 8 in Figure 39) and the 8 switches (6 in Figure 39). You should see the results on the 4 LEDs to the right of the 4 small red switches on the FPGA board (7 in Figure 39).

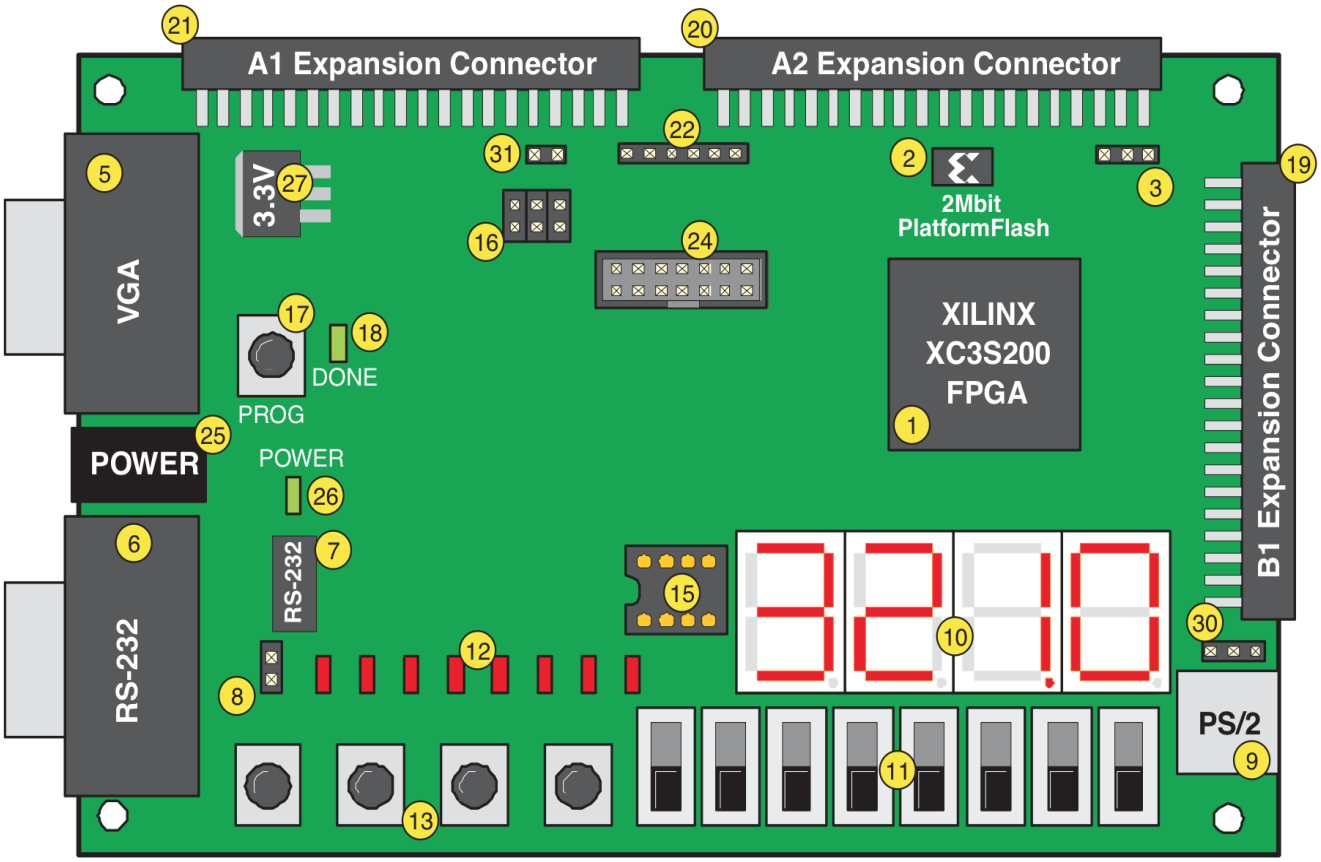

Figure 39: *The switches you will use in this lab are shown at 11. The Pushbuttons are at 13. The 4 left switches represent the bits A[3:0]. The 4 right switches represent the bits B[3:0]. The 2 leftmost buttons are the LEFT and RIGHT pushbuttons. The 4 rightmost LEDs, shown at 12, represent result[3:0].*

Your project should work at this point. If it doesn't, it's time to talk to the TA.