EE108 Lab 0

Introduction to Verilog and the EE108 Toolchain

Due: Read before your assigned lab session, complete sections 3, 4, and 5 during your lab session.

Objective: The purpose of lab 0 is to familiarize you with the concepts and tools we will be using to design digital hardware.

1. Introduction

So, what are we doing here?

In EE108 we teach you how to design digital logic.

Think about a stereotypical circuit board, like the one on the right. The flat black "computer chips" are **ASIC**s (application specific integrated circuits) with millions of transistors inside of them. In this class, we teach you how to design these chips.

Turns out we do this with a **Hardware Description Language (HDL)** that looks a lot like the programming languages your totally jealous CS friends use every day. There are many different HDLs such as Verilog, VHDL and Lava. In this class we will use **Verilog**. Engineers write Verilog code and then use a program to **synthesize** it into hardware, the same way a compiler compiles C code into machine language. The synthesis tool will translate the HDL into actual images of transistor circuits which are then burned onto silicon wafers using photolithography.
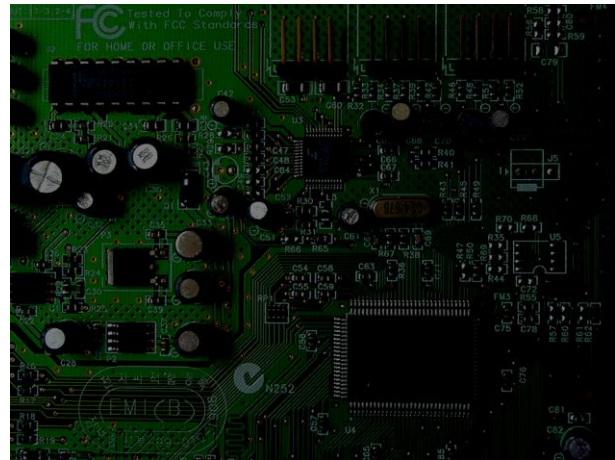
Fabricating an ASIC costs a lot of money though. Millions of dollars. In this class we will use **FPGA**s (field programmable gate arrays) to prototype your digital logic, which can be reconfigured with a new design as many times as you want and only cost a few hundred dollars each. An FPGA is like a blank canvas. It's a massive collection of general purpose logic gates (also called slices) with configurable connections between them. FPGA synthesis tools translate verilog into a description of how those gates should behave and how they should be connected together. Then, the FPGA is configured accordingly. They don't retain their designs when they lose power, though, so every time we turn them on we must re-configure them. Many companies make FPGAs; the two big ones are **Xilinx** and **Altera**. In this class we'll be using Xilinx FPGAs, and their associated software suite called **ISE**.

Synthesizing to an FPGA takes time and doesn't help much when we're trying to debug broken code, so we will also be **simulating** our designs in software using a program called **ISim**. Those simulations will dump giant files which contain the value of every wire in the design at every point in time. We view these dumps graphically, which renders these dumps like an oscilloscope would, as horizontal waves wobbling up and down over time. There are other Verilog simulators out there, the most common of which is **Modelsim** by Mentor Graphics.

Cool, so how do I write Verilog?

Verilog code is arranged into **modules** the same way C or Java code is arranged into functions. By convention we put only one module in a file and give both the module and file the same name.

```
module add_2_nums ( input  wire [7:0] num1,
                    input  wire [7:0] num2,
                    output wire [7:0] sum );
     // Add the two 8-bit inputs together
     assign sum = num1 + num2;
endmodule
```

This module would be placed in a file called 'add_2_nums.v'. We give the module a name and a **port list**, a comma-separated list of signals that flow in and out of our module. Let's look at the first input:

```
input wire [7:0] num1,
```

We start by declaring the **direction** of the port – input means some logic outside of our module is setting this port's value. We say the port is a **wire** – we will get more into this later. We then specify the **width** of the port, [7:0]. This defines num1 as an 8-bit signal. Think of it as a bundle of 8 physical wires, numbered from 0 to 7. We call wire 0 the **least significant bit** and wire 7 the **most significant bit**.

In the body of the module we have to set the value of every output. We call this **driving** the output signal. The simplest way of doing this is with an **assign** statement:

```
assign sum = num1 + num2;
```

Here, we assigned the value of sum to equal the binary addition of num1 and num2. Verilog has many built in operators you can use:

Arithmetic operators	+	-	*	/	%
Bitwise boolean	& (and)	| (or)	^ (xor)	~ (inverse)	
Logical boolean	&&	||	! (not)		
Comparators	==	<=	>=	!=	
	===	<	>	!==	
Shift	<<	>>	<<<	>>>	
Tertiary	?:				

In this lab you will just be using bitwise-AND and addition, but you will learn about and use the rest of these operators throughout the quarter. The use of division and modulo operators is discouraged as they are particularly resource hungry.

We can add comments to our code using C and Java style syntax – all text on the line after a // is ignored by the Verilog interpreter. Similarly, all text between /* and */ is ignored.

More complex modules

We can declare wires internal to the module which outside code cannot see, and we can **instantiate** other modules we've written much like we call functions in other languages:

```
module add_4_nums ( input  wire [7:0] a,
                    input  wire [7:0] b,
                    input  wire [7:0] c,
                    input  wire [7:0] d,
                    output wire [7:0] out );
    wire [7:0] a_plus_b, c_plus_d;
    add_2_nums first_adder  ( .num1(a), .num2(b), .sum(a_plus_b) );
    add_2_nums second_adder ( .num1(c), .num2(d), .sum(c_plus_d) );
    assign out = a_plus_b + c_plus_d;
endmodule
```

We instantiate modules by writing the module name, giving it a specific **instance name**, and then hooking up all of its ports. The order we specify the ports doesn't matter because we specify both the name of port from the module's port list, and then the name of the signal we're connecting to it:

```
.port_name(signal_name)
```

Verilog is not a sequential programming language!

The most important thing about Verilog to keep in mind is you are not writing a list of instructions to be executed from top to bottom. You are describing physical circuits which will have electricity flowing through them at all times. **Everything you write will be translated into a physical object. Everything you write is "executing" all the time**. In our add_2_nums module, we declared three physical, metal sets of wires that are connected to an adder circuit made of transistors. In our add_4_nums circuit, we instantiated two duplicate copies of this module which will be placed next to each other on the chip. They compute their sums at the same time and have no sense of "before" or "after". Because of this we can rearrange the lines of our add_4_nums module in whatever order we want:

```
    add_2_nums first_adder  ( .num1(a), .num2(b), .sum(a_plus_b) );
    add_2_nums second_adder ( .num1(c), .num2(d), .sum(c_plus_d) );
    assign out = a_plus_b + c_plus_d;
```

```
    add_2_nums second_adder ( .num1(c), .num2(d), .sum(c_plus_d) );
    add_2_nums first_adder  ( .num1(a), .num2(b), .sum(a_plus_b) );
    assign out = a_plus_b + c_plus_d;
```

```
    add_2_nums second_adder ( .num1(c), .num2(d), .sum(c_plus_d) );
    assign out = a_plus_b + c_plus_d;
    add_2_nums first_adder  ( .num1(a), .num2(b), .sum(a_plus_b) );
```

All of these pieces of code behave exactly the same. **We can only assign the value of a wire once**, or equivalently, **a signal can only have one driver**. We will see more consequences of this soon.

'Always' blocks, regs, and if statements
Verilog has **if statements** that look similar to their C and Java counterparts. When we use if statements, they must be enclosed in an **always block**. Always blocks begin with the line "always @(*) begin" and end with "end":

```
module one_or_the_other ( input wire A,
                          input wire B,
                          input wire [7:0] val_1,
                          input wire [7:0] val_2  );

reg chosen_value;

always @(*) begin
     if (A&B) begin
          chosen_value = val_1;
     end else begin
          chosen_value = val_2;
     end
end

endmodule
```

In Verilog you use the keywords **begin** and **end** the same way you use { and } in C++ or Java. The @(*) part is called the **sensitivity list**. Later in the quarter you will learn more about this and what it is used for.

Any signal that is set inside of an always block must be declared as a **reg** instead of a wire. We can output regs as well, we just replace 'wire' with 'reg' in our port list. There is no difference between a wire and a reg other than who is driving it. You can follow these rules to decide when to use one or the other:
  • A signal changed in an always block must be declared as a reg.
  • A signal changed in an assign statement must be declared as a wire.
  • Looking from the outside of a module, signals coming out must be declared as wires.
  • Looking from the inside of a module, signals coming in must be declared as wires

Also notice we don't use the assign keyword inside of an always block. We can put as many if statements inside of one always block as we want, and we can nest if statements inside of each other, but the more complex we make our logic the easier it will be to make mistakes and get confused about what is actually going on. In general, keep the contents of your always blocks small and related to each other. Otherwise, consider breaking it into separate always blocks or even decomposing into multiple modules.

Again, Verilog is not a sequential programming language!!

Though it looks similar to its C and Java counterparts, this if statement is actually instantiating a physical switch that routes one of two input wires to an output wire, depending on the value of a third signal called the **select** signal. This switch is called a **multiplexer**. We illustrate multiplexers in block diagrams like this:
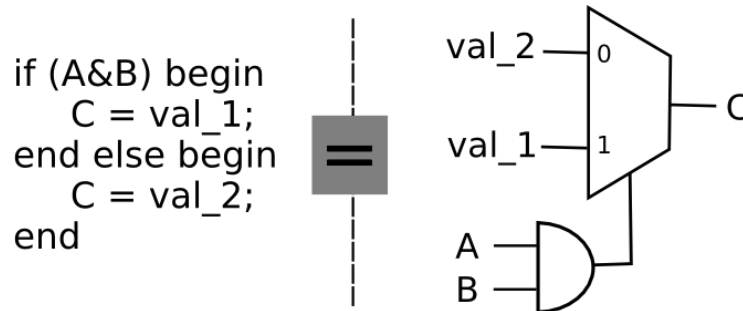
Instead of saying "if condition A&B is true, *do* this, otherwise, *do* that", a better way to think about it is "if condition A&B is true, C will be connected to this wire, otherwise it will be connected to that wire". This way we avoid accidentally thinking about our module as a program that executes from top to bottom.

With this in mind we need to remember some things when using if statements:
- **There always has to be an 'else' clause**

Without the else, what is the value of C when A&B is 0? It's undefined, and that is bad! The synthesizer will **infer a latch**. A latch is a memory element that retains a value over time, and its effect here will be to drive C with whatever value it had when it was last defined. We will never do this in ee108, and if the tools report to you they have found inferred latches you are doing something wrong.
- **A reg should only be set in one always block, and should always be set at least once in that always block**

We can't have one if statement assign a value to a reg and then, elsewhere in the module, have another if statement that also assigns a value to that reg. In general, if you give values to a set of regs in one clause of an if statement that exact same set of regs should be given values in every other clause of the if statement.
- **Everything is happening all the time**

In Verilog this code is nonsensical:

```
// Take the absolute value of 'number'
// (THIS CODE IS NOT CORRECT!)
always @(*) begin
    if (number < 0) begin
        number = -number;
    end else begin
        number = number;
    end
end
```

By **using a reg in the condition of an if statement and assigning it a value it its body**, we are creating a feedback loop which will not do what we intend. Similarly, by **using the same value on both the left and right side of the equals sign** we're creating an even tighter feedback loop. You should take extra care to never do either of these things when writing Verilog.

Case statements, numerical constants, and nifty bit swizzling notation

There are a few more constructs in Verilog which you won't be using in this lab but will want to know for the future.

The first are the **case statement** and **numerical constants**. Like the if statement, it also must reside in an always block and translates into a multiplexer:

```
module select one ( input wire [7:0] index,
                    input wire [7:0] a,
                    input wire [7:0] b,
                    input wire [7:0] c,
                    input wire [7:0] d,
                    output reg [7:0] out );

always @(*) begin
    case (index)
        8'd1: out = a;
        8'd2: out = b;
        8'd3: begin
            // Cases can be multi-line and nested, too!
            out = c;
        end
        8'd4: out = d;
        default: out = 8'd0;
    endcase
end

endmodule
```

If *index* is 1, *out* will be given *a*'s value. If *index* is 2, *out* will be given *b*'s value, and so on. The **default** clause is the equivalent of our if statement's else clause. It catches all cases where *index* has a value which we didn't specify in our case statement. **All case statements must have a default clause.** This just boils down to a giant multiplexer.

Also notice the way we wrote numbers. In Verilog we write numbers in the following format:
[bit width]'[base][value]

The bit width field specifies how many physical bits represent this number. 8'd2 will reperesent the decimal number 2 with 8 wires, equivalent to the binary 00000010. 3'd2 will also represent the decimal number 2, but with 3 wires 010. It's important you choose bit widths equal to the wire or reg you're assigning the number to.

The base field clarifies whether the number we wrote is in binary, octal, decimal, and hexadecimal (represented using b, o, d, and h, respectively). For example, we could represent the 4-bit number 13 in these ways: 4'b1101, 4'o15, 4'd13, and 4'hD.

The last thing to mention is that, since Verilog's purpose is to describe how to push around bits, it has very powerful bit-level notation.

To take a slice out of a large signal, we use array notation similar to C or Java:

```
wire [49:0] big_one;
wire [4:0]  little_one;
wire        tiny_one;

assign little_one = big_one[15:11];
assign tiny_one = big_one[3];
```

This takes five-bit and one-bit slices out of *big_one* and hooks them up to two smaller wires.

To concatenate multiple signals into one larger signal we can use curly braces:

```
wire [6:0] seven_bit_signal;
wire [2:0] three_bit_signal;
wire [9:0] ten_bit_signal = {seven_bit_signal, three_bit_signal};
```

This will assign the *seven_bit_signal* to the seven most significant bits of *ten_bit_signal* and *three_bit_signal* to the three least significant bits of *ten_bit_signal*.

Inside of these curly braces we can string as many signals together with commas as we want to, use the bit slicing mentioned above, and can include numerical constants as well.

We can also use concatenation to do compound assignment:

```
assign {two_bit_signal,three_bit_signal} = five_bit_signal;
```

This is equivalent to saying:

```
assign two_bit_signal   = five_bit_signal[4:3];
assign three_bit_signal = five_bit_signal[2:0];
```

To repeat one signal many times, we use double curly braces:

```
wire [2:0] awooga;
wire [8:0] awooga_awooga_awooga;
assign awooga_awooga_awooga = {3{awooga}};
```

This will duplicate the value of awooga three times, creating a 9-bit signal. Again, we can mix and match this with concatenation and slicing notation however we wish:

```
assign ten_bit_signal = {six_bit_signal[4:0], {3{1'b1}}, two_bit_signal};
```

One last warning

If you misspell a variable name somewhere in your code, some Verilog tools will assume you want to declare a new wire with that name of bit width 1. For experienced designers this can save writing some extra code, but proves to be a huge nuisance when you don't get any syntax errors yet your simulations don't work. If things ever seem to be going wrong, one of the first things you should do is go back and check that you spelled all of your wires and regs correctly.

2. Designing a simple circuit

Designing complex digital circuits can get unmanageable fast if we're not careful. To keep our heads from exploding we will carefully map out our designs on paper first and then write actual code. If we do things right, by the time we write the Verilog it should just feel like copying the block diagrams we draw word-for-word into a format the computer can read.

In this lab we will build a very simple circuit to give you practice writing Verilog and using the class software. Our design will take two four-bit binary numbers, A and B, and output either (A & B) or (A + B) depending on whether the left or right pushbutton is held down.

Specify the design

Start by clearly and specifically explaining what your design is going to do. In this case,

We will take two four-bit binary numbers from the 8 DIP switches on the FPGA board. Switches 1-4 will form number A and switches 5-8 will form number B. We will also input the value of the left and right push-buttons. We will output a 4-bit number to the row of LEDs on the board, depending on the following:
- When the left push-button is pressed, we will output A and B ANDed together.
- When the right push-button is pressed, we will output the sum of A and B.
- If neither button is pressed, no LEDs should be turned on.
- If both buttons are pressed, the circuit should act as if only the right button is pressed.

It might seem obvious but it's important to fully understand the function of the machine you're trying to build before you dive into the logic of it. Talking this out with your teammates before touching a computer will help you design cleaner and less buggy logic.

Draw block diagrams

From this very general description, block the design out on paper so you can follow the flow of data from input to output. In this class we will expect you to draw a high-level block diagram showing how all of your modules connect to each other, and then more detailed diagrams of each module showing exactly how they function.

High-level/system-level block diagram

Let's start with the high level block diagram. Draw a box for every module you will eventually instantiate in Verilog and a line for every signal going in or out of it. If a signal is more than 1 bit wide, draw a slash on the line and write the bit width above it. Inside the box, write the module name and label each input and output. For this lab we only have one module and we only instantiate it once, so our diagram is going to be pretty simple:
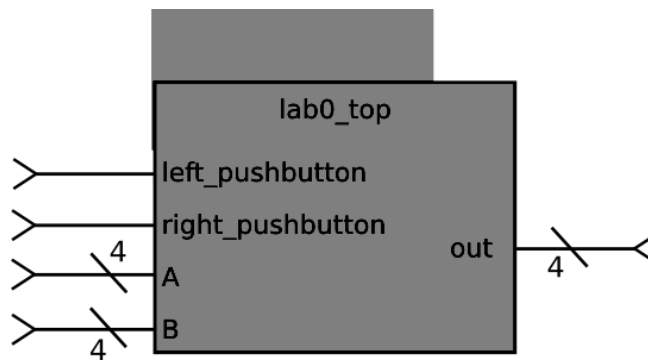
This level of block diagram is good for dividing up work amongst members of your lab group. One of the biggest difficulties you will face is designing modules that integrate well with the work from the rest of your group. Spend time with your teammates concretely laying out this system-level block diagram so that you can rely on the interfaces between modules and don't have to know their implementation to get your parts to work.

You might also find it useful to write out the interface for each module in table form:

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)
Out	Output	4 bits (binary 0-15)	Our output, which is either A & B or A + B according to the button(s) we press

Low-level/module-level block diagram
Next we plan out the actual guts of our design. At this level we will draw logic gates, multiplexers, adders, and wires that describe exactly how our machine functions. If we instantiate other modules inside of the one we're drawing, we will represent them with blocks the same way we did in our high-level diagram.

So let's look at what our design has to do, starting from the output:

We will output one of three things based on some other logic. That means we're going to want a multiplexer with its output connected to *out*.
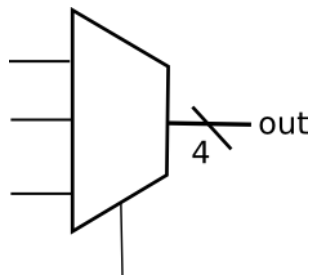
One of those things we could output is A+B. So, lets draw an adder at one of the multiplexer's inputs, with A and B connected to the adder's inputs:
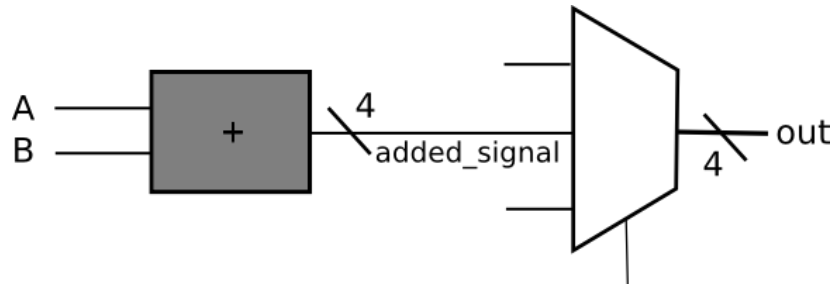

Another thing we could output is A&B. We'll just draw one AND gate and put slashes on its inputs and outputs to indicate we're actually ANDing two 4-bit signals to produce another 4 bit signal. This is equivalent to drawing 4 AND gates, one for each bit (don't draw that, don't ever draw that).
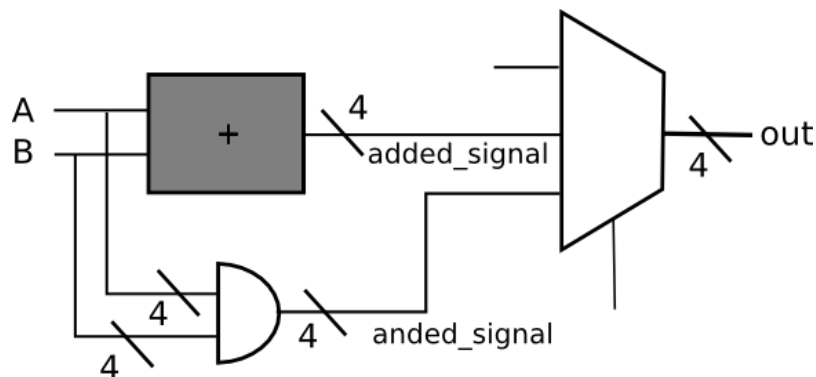

The last thing we might output is 'nothing', which is represented by 4 zeros. We'll represent this by writing "4'b0" as the third input:
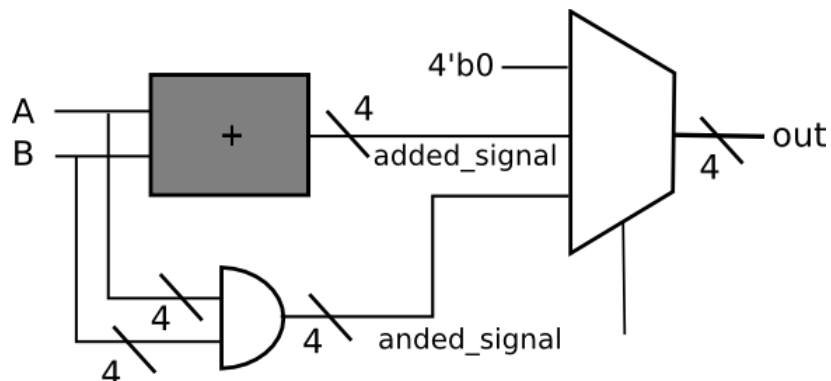
Finally we have to work out the select signal for the multiplexer. The inputs we have are *left_pushbutton* and *right_pushbutton*. We'll concatenate these into a two-bit signal by writing {left_pushbutton, right_pushbutton} and feeding that into the multiplexer. When neither button is pressed this will have the value 00. When the left button is pressed, it will be 10. When the right button is pressed, it will be 01. We will write these values on the multiplexer accordingly.
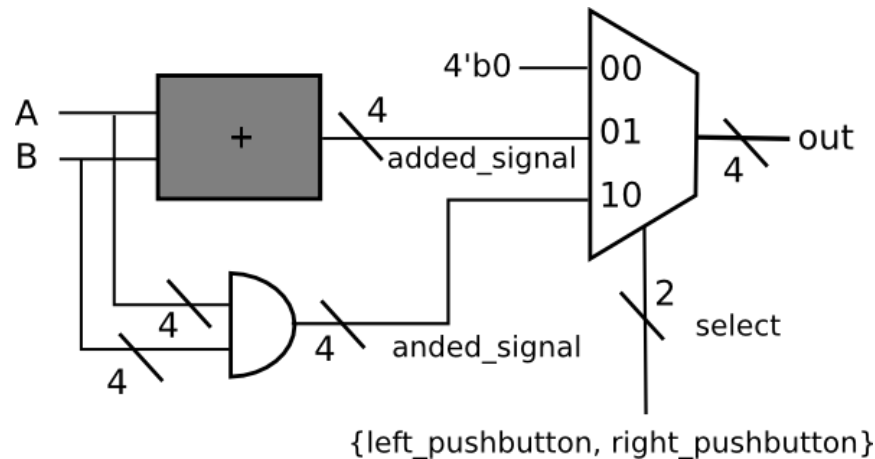
But wait, we forgot the case where both buttons are held down! With our design as is, we would generate an inferred latch when we went to synthesize because we never said what to do when the select input is 11. Our design specification says it should act as if only the right button is pressed (the select signal is 01), so we'll just add a 11 input to the multiplexer and short it to the 01 input.
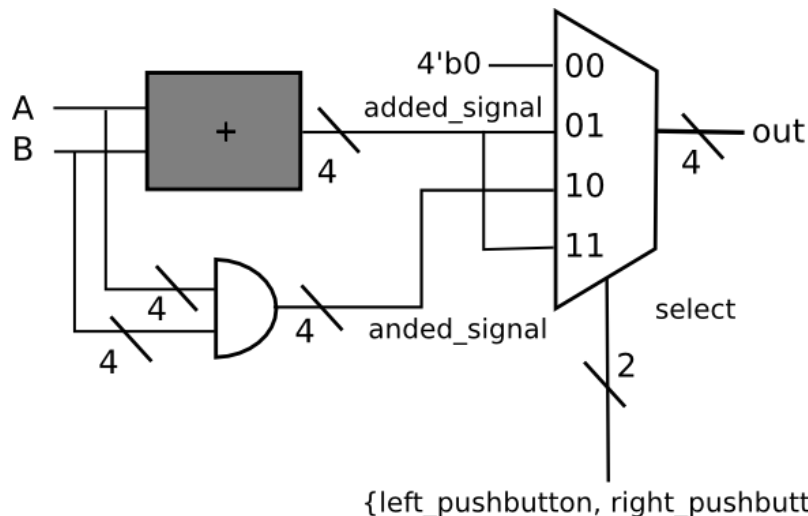
In general, because our multiplexers have binary select signals, the number of inputs they have should be a power of 2. With one select bit we have two possible inputs. With two select bits we have four, with three we have eight, etc.

Writing the Verilog

Thanks to all of our work on our block diagrams the Verilog itself shouldn't take much thought to write. From our system-level diagram we see we only have one module called lab0_top:

```
module lab0_top ( input wire left_pushbutton,
                  input wire right_pushbutton,
                  input wire [3:0] A,
                  input wire [3:0] B,
                  output reg [3:0] out );

endmodule
```

We'll start by defining the two inputs to our multiplexer and its select signal:

```
module lab0_top ( input wire left_pushbutton,
                  input wire right_pushbutton,
                  input wire [3:0] A,
                  input wire [3:0] B,
                  output reg [3:0] out );

  wire [3:0] anded_result;
  wire [3:0] added_result;
  wire [1:0] select;

  assign anded_result = A & B;
  assign added_result = A + B;
  assign select = {left_pushbutton, right_pushbutton};

endmodule
```

And finally we'll build the multiplexer:

```
module lab0_top ( input wire left_pushbutton,
                  input wire right_pushbutton,
                  input wire [3:0] A,
                  input wire [3:0] B,
                  output reg [3:0] out );

  wire [3:0] anded_result;
  wire [3:0] added_result;
  wire [1:0] select;

  assign anded_result = A & B;
  assign added_result = A + B;
  assign select = {left_pushbutton, right_pushbutton};

  always @(*) begin
    case (select)
      2'b00: out = 4'b0;
      2'b01: out = added_result;
      2'b10: out = anded_result;
      default: out = added_result;
    endcase
  end

endmodule
```

Note we declared *out* as a reg, because it is driven by logic inside of an always block. Don't worry if you get confused about when to use *reg*s and *wire*s, it's something everybody has trouble with and gets used to by the end of the quarter. The good news is if you mix it up your code simply won't compile. This is significantly better than inferring a latch, which lets everything synthesize just fine but produces unpredictable results when we go to test on actual hardware.

3. Simulating our design

Setting up the tools
Before we use the code we just wrote, we have to familiarize you with the tools you'll be using throughout the quarter. Each person in your lab group should do the following from one of the computers in Packard 128, the ee108 lab.

Log in with your regular Stanford username and password. If you have trouble, try adding a "win\" before your username, so that the field looks like "win\username".

To share your projects with your group, we have created group folders for each of you. You probably will want to map the EE108 class directory to a network drive. Open up a file explorer window, right-click on Network and select "Map Network Drive." From there, type in

```
\\afs\ir\class\ee108
```

You can then navigate to your group's folder in the group subdirectory.

This is the folder in which you can keep all your work for ee108. All the members of your group have access to the files and folders found within your group's directory. If you have problems accessing your files, you should let the course staff know immediately.

Creating an ISE project
Now, we'll open ISE. From the desktop, double click the ISE icon.

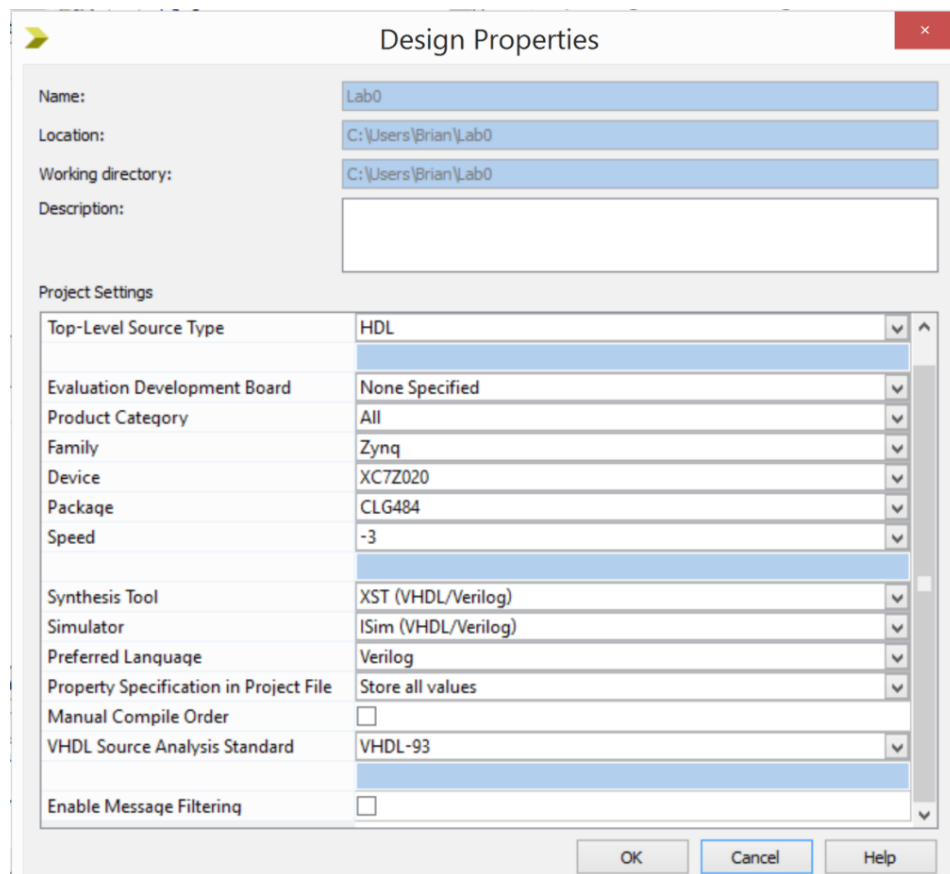Once ISE opens, create a new project for lab 0 by going to File → New Project…

ISE will choose directories more nicely if you fill out the fields in the right order. First, under "location," specify your group directory (in the mapped network drive):

```
/afs/ir/class/ee108/groups/<group number>
```

Then, fill in the name of the project, `lab0`. The location and working directory should get automatically updated:
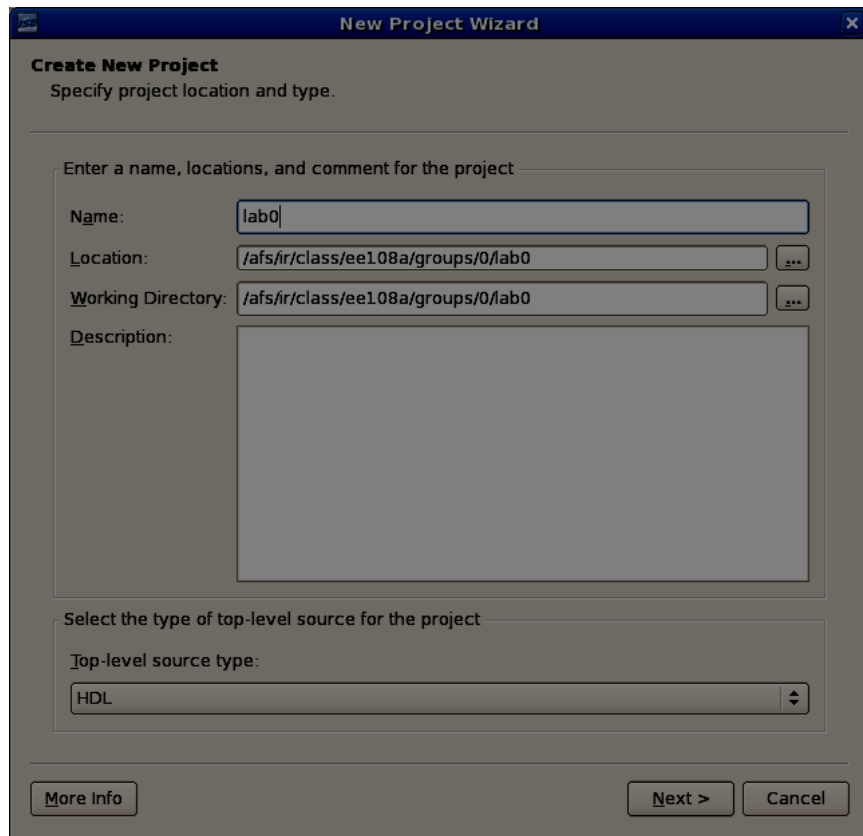
Click next, and make sure that the options ISE chooses by default match the ones in the screenshot below. This is where you tell ISE which exact chip you are using and what language your source code is in. Click next and then finish to complete the new project wizard.

Design Properties

Name:	Lab0
Location:	C:\Users\Brian\Lab0
Working directory:	C:\Users\Brian\Lab0
Description:	

Project Settings

Top-Level Source Type	HDL
Evaluation Development Board	None Specified
Product Category	All
Family	Zynq
Device	XC7Z020
Package	CLG484
Speed	-3
Synthesis Tool	XST (VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	Verilog
Property Specification in Project File	Store all values
Manual Compile Order	☐
VHDL Source Analysis Standard	VHDL-93
Enable Message Filtering	☐

OK   Cancel   Help

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:
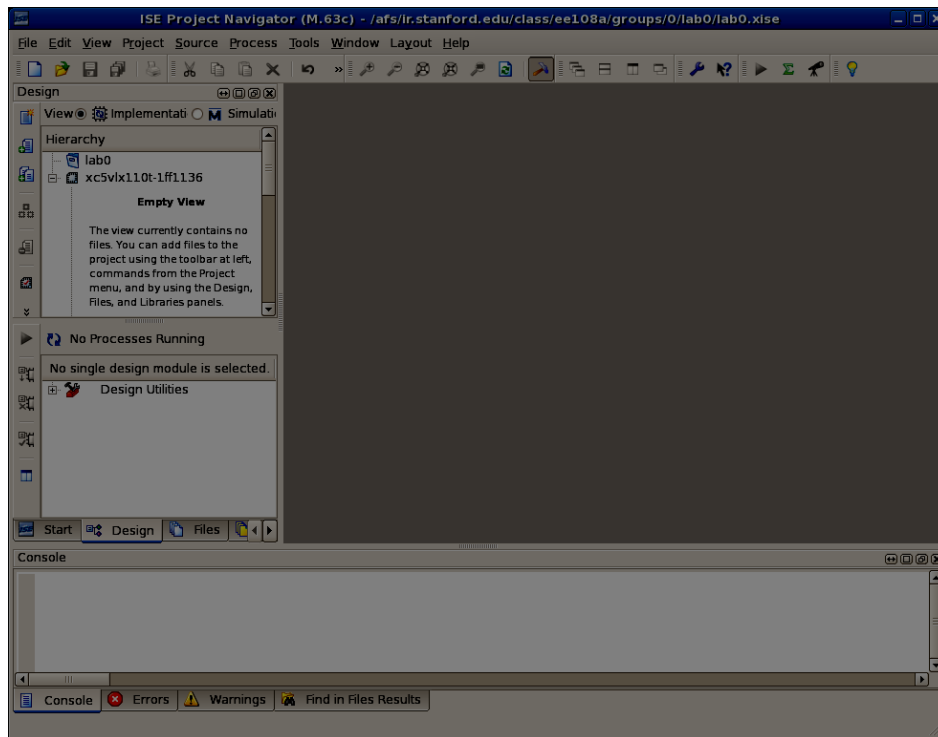
Now we'll add a Verilog source file to our project. Click Project → New Source. Select 'Verilog Module' form the listbox on the left and type 'lab0_top.v' in the file name box. Click next.
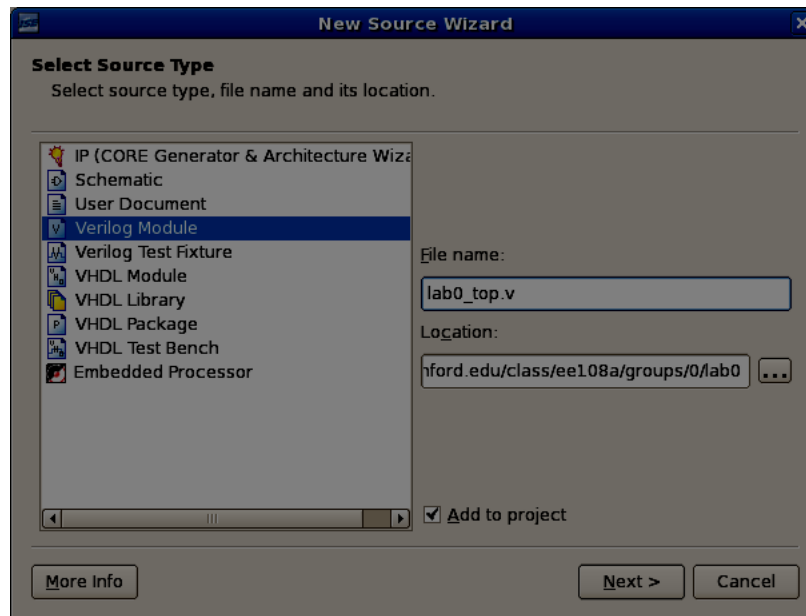
Leave the "Define Module" screen blank by clicking 'Next', and then click 'Finish'. In the future this screen will allow you to quickly specify your module's inputs and outputs, but for lab 0 we're going to do it ourselves.

ISE's editor should automatically open up to edit your new Verilog file. If the text is not colorfully highlighted, click View → Highlight Mode → Sources → Verilog.

Delete all the text ISE provided for us and enter the Verilog code that we wrote in part 2. Careful if you're cutting and pasting! Oftentimes the ' and " symbols in formatted documents like this one are replaced with special characters that look nicer but aren't interpreted by ISE as quotemarks. If you get syntax errors, go through your code and re-type any quotes you find.

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. 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.

If we forgot a semicolon in our code, the ISE window would look something like this:
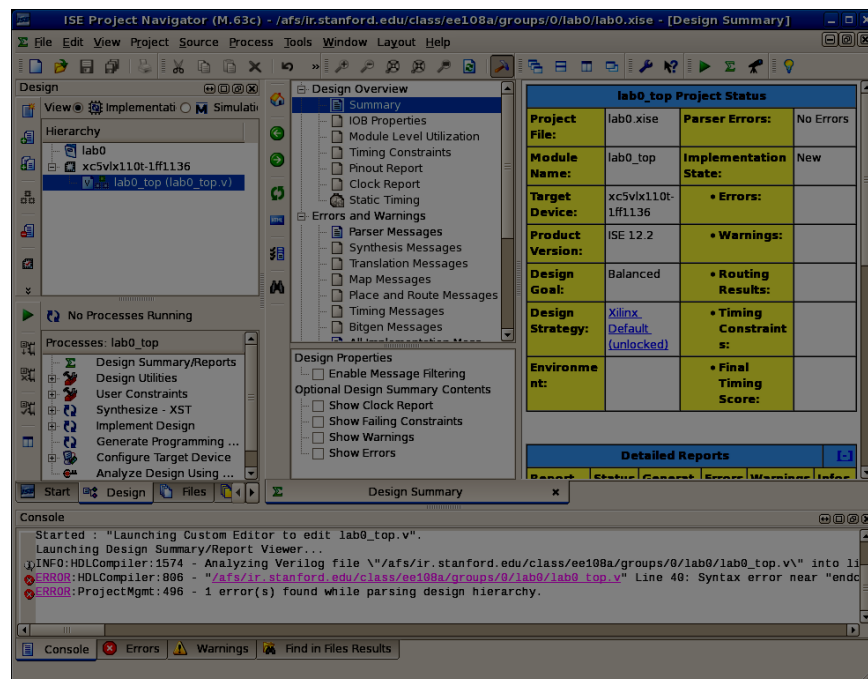
If all is well, you should see "Parsing design hierarchy completed successfully" in the transcript window:
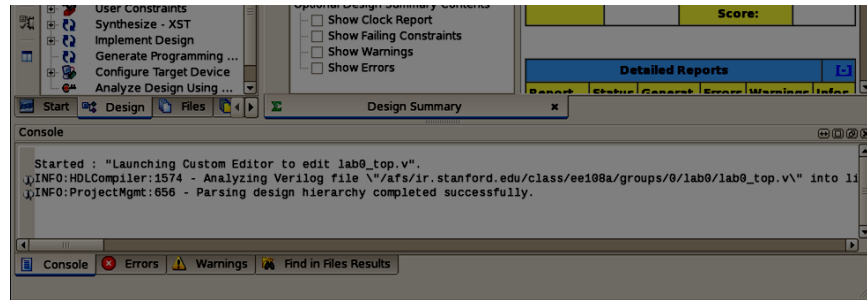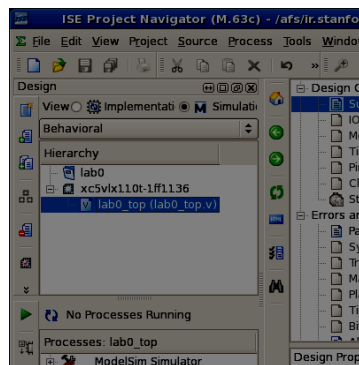
Testbenches

A large part of our time as digital designers is spent just verifying that the circuits we are describing actually work. One effective way of doing so is by simulating our designs in a Verilog simulator.

To do this we write **testbenches**, also in Verilog. Testbenches **are not synthesized into real circuits**, so they can contain **initial blocks** which are executed from top to bottom like a computer program. We instantiate the module we are testing, the **device under test**, and then use initial blocks to feed its input ports different values over time. Once the simulation concludes we can inspect the value of any variable in the module at any point in time, verifying correctness.

Think of a testbench as a movie script. The device under test is our leading actor, and the intial blocks are supporting characters in the scene. All of them run in parallel and have a conversation with each other as simulated time moves forward. We write our initial blocks with the expectation that the device under test will act a certain way, and by reviewing the recorded scene after it's over we can see if the device under test had any idea what it was doing in the first place.

So let's get started. Switch ISE into simulation mode by clicking the Simulation view button at the top of the Design pane. Make sure the dropdown box below it says 'Behavioral':

Next, create a new Verilog file called lab0_top_tb.v in the same way as before (Project → New Source...). By convention when writing a testbench for a module, we use the module name with '_tb' appended to the end. Double-click the file in the hierarchy to open it in gedit.

Because the testbench is self-contained it doesn't have any inputs or outputs:

```
`timescale 1ns/1ps
module lab0_top_tb();
```

```
//Test-bench body

endmodule
```

Next we'll create a reg for every input to our lab0_top module, and a wire for every output:

```
`timescale 1ns/1ps
module lab0_top_tb();
    reg sim_left_pushbutton;
    reg sim_right_pushbutton;
    reg [3:0] sim_A;
    reg [3:0] sim_B;
    wire [3:0] sim_out;

endmodule
```

Now we'll instantiate our device-under-test:

```
`timescale 1ns/1ps
module lab0_top_tb();
    reg sim_left_pushbutton;
    reg sim_right_pushbutton;
    reg [3:0] sim_A;
    reg [3:0] sim_B;
    wire [3:0] sim_out;

    lab0_top DUT( .left_pushbutton(sim_left_pushbutton),
                  .right_pushbutton(sim_right_pushbutton),
                  .A(sim_A),
                  .B(sim_B),
                  .out(sim_out) );

endmodule
```

But without our movie script the actor will just stand around and not do anything. Let's add an initial block which wiggles some regs:

```
`timescale 1ns/1ps
module lab0_top_tb();
    reg sim_left_pushbutton;
    reg sim_right_pushbutton;
    reg [3:0] sim_A;
    reg [3:0] sim_B;
    wire [3:0] sim_out;

    lab0_top DUT( .left_pushbutton(sim_left_pushbutton),
                  .right_pushbutton(sim_right_pushbutton),
                  .A(sim_A),
                  .B(sim_B),
                  .out(sim_out) );
```

```
    initial begin
        // start out by setting our buttons to "not-pushed"
        sim_left_pushbutton = 1'b0;
        sim_right_pushbutton = 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_pushbutton = 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_out, (4'b1100 &
4'b1010));

        // stop simulating
        $stop;
    end
endmodule
```

We start by setting initial values for our regs and then waiting 5 time steps. In Verilog, a pound sign followed by a number instructs the simulator to wait that number of 'timesteps'. This has no hardware counterpart and is ignored if you try to synthesize it.

The timescale directive at the top of the module controls what exactly a timestep means. The first number is the size of one timestep, and the second number is the precision. In this module, #5 means to wait 5 nanoseconds of simulated time. If we had a timescale of 10ns/1ns, we would be able to wait fractional amounts of time: #1 would wait 10 nanoseconds, while #1.1 would wait 11 nanoseconds.

After 5 timesteps we change our input values and give everything time to settle down. After another 5 timesteps, we use the $display command to print text out to the simulation transcript. The syntax of this command is similar to the standard C printf() function. You can read more about $display and other simulation commands at http://www.asicguru.com/verilog/tutorial/system-tasks-and-functions/68/.

We stop the simulation with the $stop command. We only need one in our testbench, but without it the simulation would never complete and we wouldn't get back any meaningful data.

Running the simulation
Save your testbench and quit gedit. Notice the hierarchy has changed, with lab0_top_tb now as the top-level module. This means the testbench module instantiates lab0_top:
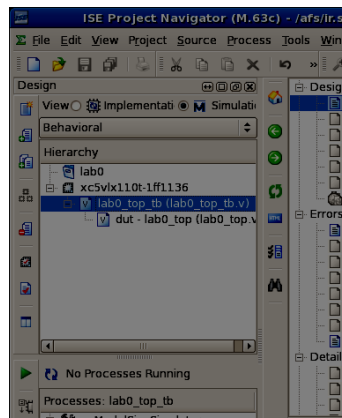
Let's try running it.

Highlight lab0_top_tb in the hierarchy panel, since this is the testbench we want to run. Expand 'iSim Simulator' in the processes panel, and double click 'Simulate Behavioral Model'.

Be sure to always select a testbench; if you run 'Simulate Behavioral Model' on a device under test itself, you will either get errors or an empty simulation result.

Since our simulation is short, everything will happen very quickly. One windows should pop up: the iSim wave display window. At the bottom there is a terminal that will report any syntax or simulation errors iSim encountered while running your testbench.

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 iSim 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. If your module has many signals, you can use the filter box to narrow down the signals shown.

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. iSim should now look something like the screenshot below. Note that the wave viewer shows a graph of signals vs. time. Our test bench only simulated for 10 time steps before hitting `$stop`, but later on we'll get much longer test benches!
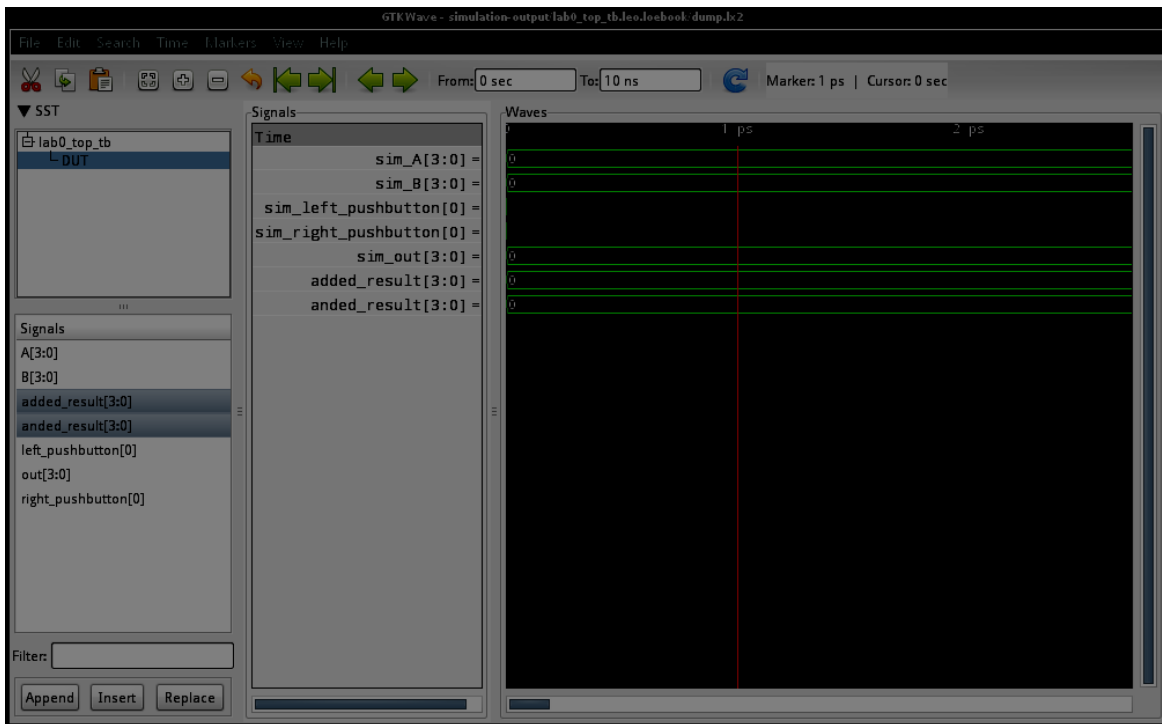
Our initial view of the testbench results

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. Now you can actually see values changing over time, as in the screenshot below:
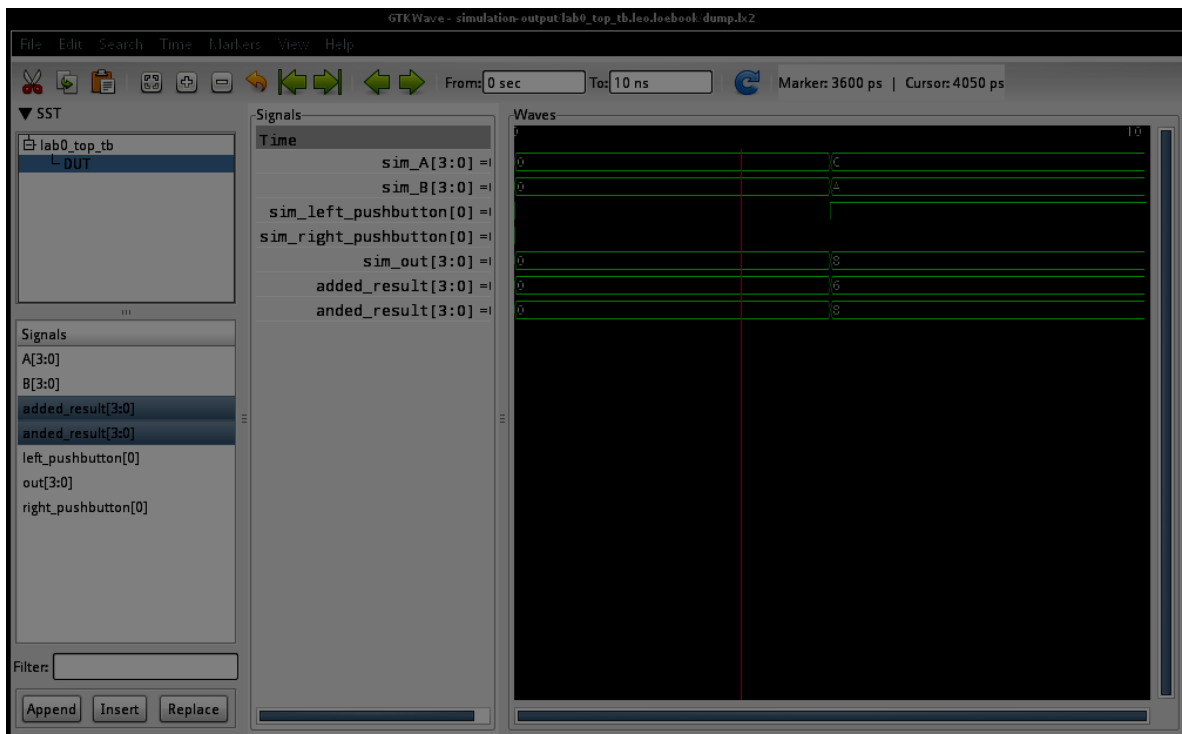
Our testbench, fully zoomed out

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 output only takes on the value 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 Radix → Binary. There are a lot of display formats available; be sure to choose the one that makes the most sense for the signal, as it makes the waveforms that much easier to understand.
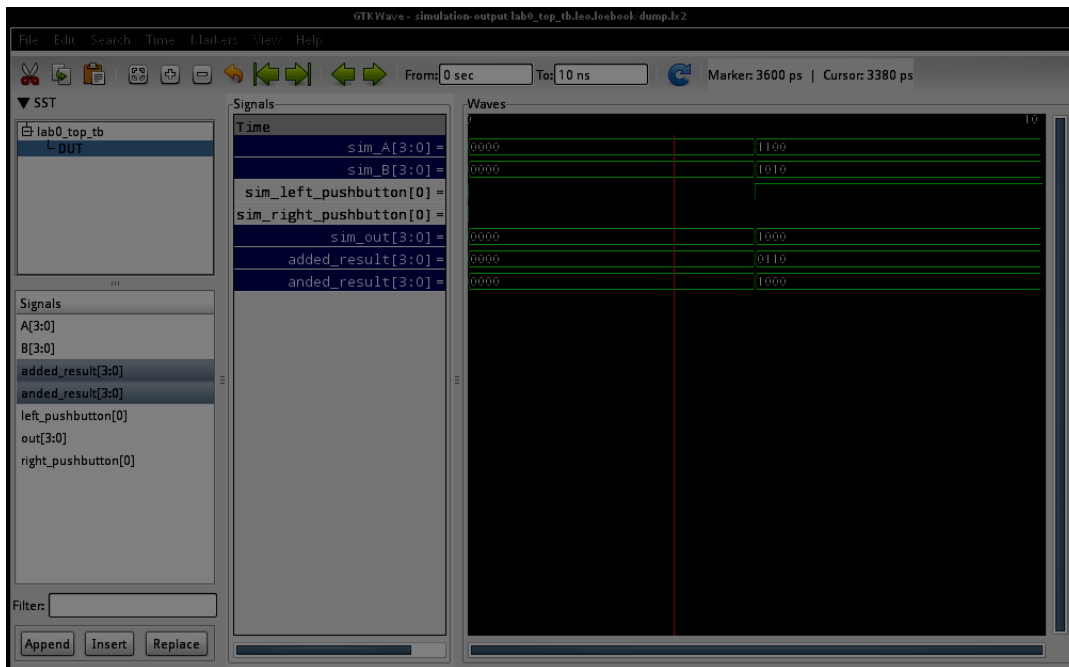
Displaying results in binary

So that we don't have to set up the wave view again, choose File → Save, or just hit Control+S. The next time you run this simulation, iSim will pop up with the display configuration as it was at the point when you last saved it. Go ahead and close iSim.

Exploding complexity

This testbench by no means rigorously tests our module, so we're going to want to add more test cases. In general, much of the work involved in writing testbenches goes into figuring out which cases we want to test.

For example, 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. 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. A good computer could likely simulate this adding circuit in a few minutes. But the adder in your laptop likely 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 take 1×10^{20} years to finish. (That's roughly 7.8 million times the age of the universe.) Considering this is just one component of a much larger circuit, to exhaustively test our designs

becomes literally impossible.

Instead, we will use our knowledge of the design to figure out interesting "edge cases" and test those. If the 4-bit adder sums up 5 and 6 correctly, it probably sums up 5 and 7 correctly too. But what happens when one of the numbers is at the edge of our range of inputs? What happens when the value overflows to a number larger than 4 bits? It is a good strategy to select edge cases for your testbenches along with a handful of random inputs which aren't particularly interesting but should work anyway. Believe it or not, this is the best companies like Intel and NVIDIA have been able to come up with (ask your TA if you're interested in the subject of verification, there is a whole seminar series on it offered winter quarter called EE392T)

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

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

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

// Let's go back to ANDing
sim_left_pushbutton = 1'b1;
sim_right_pushbutton = 1'b0;
#5

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

Remember to insert this code into your initial block before the $stop, or the simulation will complete before it gets a chance to run.

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.

The log will now show the new tests. Some of the expected values are printed as 5 bits; this is because iSim will never drop precision (bits) in intermediate values (such as the ones we were giving to $display). It will only drop 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 iSim popped up with the same signals displayed that we saved before. You may have to hit the "zoom fit" button again to accommodate the longer testbench:
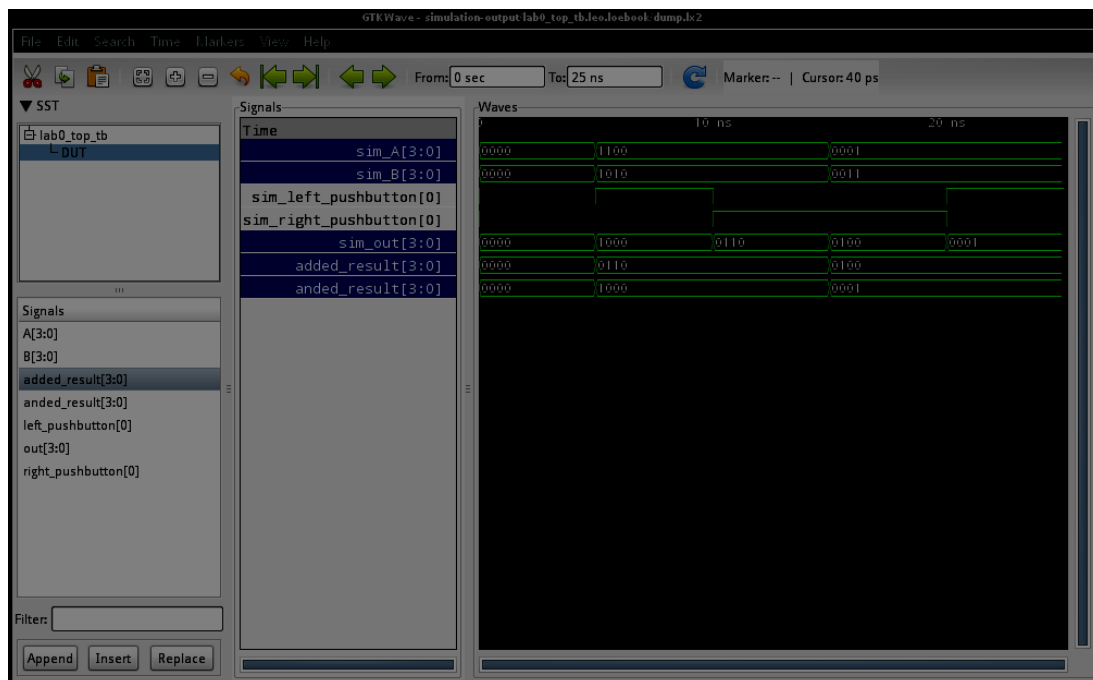
Our new test results

Annotating

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, as demonstrated below. To help with annotating, you can either take a screenshot of the window, or use the File → Print To File command. For this lab we would expect something like this:
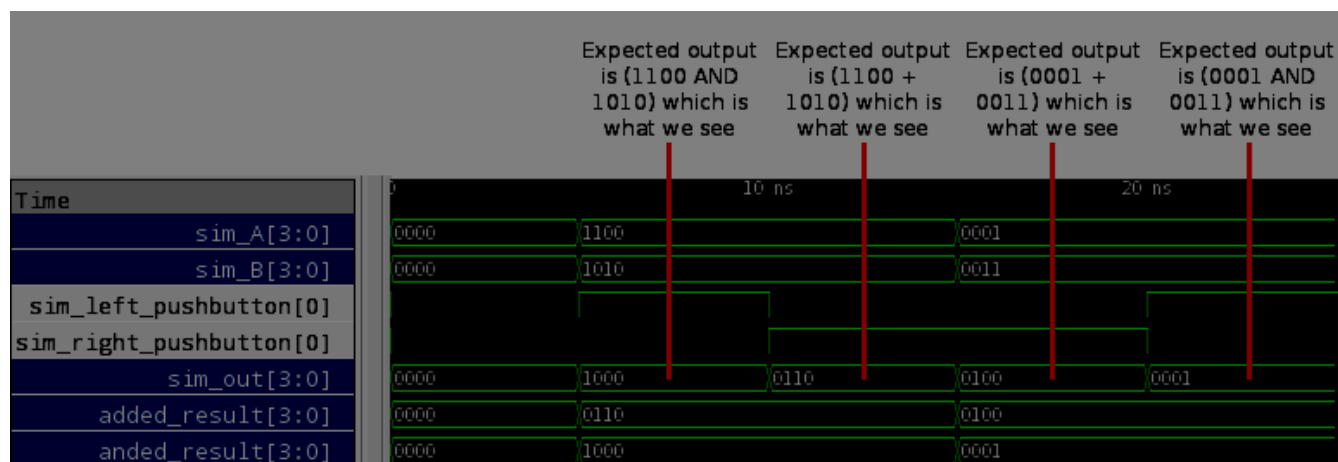
A well-annotated wave diagram

Your annotations don't have to be fancy – what's important is you give us some sense of what we're looking at, as if you were explaining it to us face to face. If you don't make any effort to annotate your diagrams, *we won't even read them*.
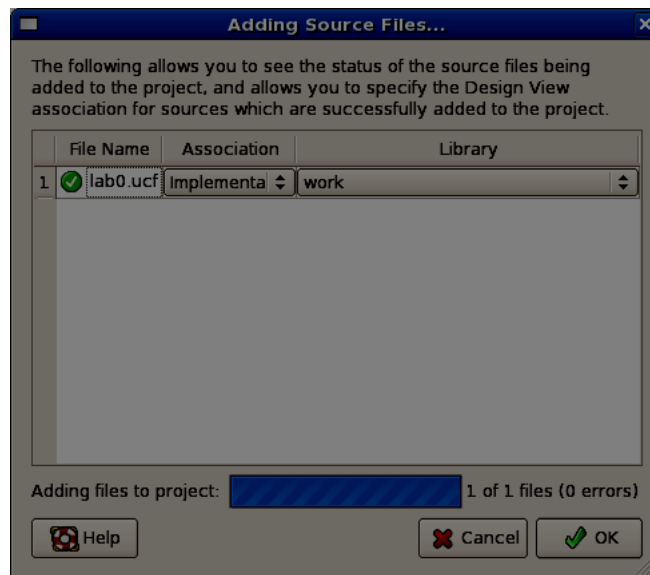
4. Synthesizing our design

Setting up for synthesis

Once we have sufficiently verified our designs in simulation, we'll synthesize them onto an actual FPGA. First, switch ISE back into 'Implementation' mode with the radio buttons above the hierarchy box. Next we're going to add a "user constraints file", or **UCF**, to the project. This file tells ISE which physical pins on the board each input and output in our top-level module connect to.

The UCF file for this lab is located in the lab starter files directory,

```
/afs/ir/class/ee108/labstarterfiles/lab<lab number>
```

In this case, that last directory will be 'lab0'. In ISE, click Project → Add Copy of Source..., and find the UCF file in the directory listed above. Select it and click 'Open' – note that ISE is making a copy of this file in your project folder you are now free to edit without affecting other groups.


Adding the UCF file to our project

Also note the 'Association' of the file. 'Implementation' 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 simulatable, will have 'All' as their association. The 'Library' should always be left as 'work'. Hit OK to add the UCF file.

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. Since ISE didn't realize you were making a testbench at the time, however, it went ahead and set its association to 'all'. Fix this by right-clicking the offending module and choosing 'Source Properties'. Change its association to be 'Simulation' and hit OK.
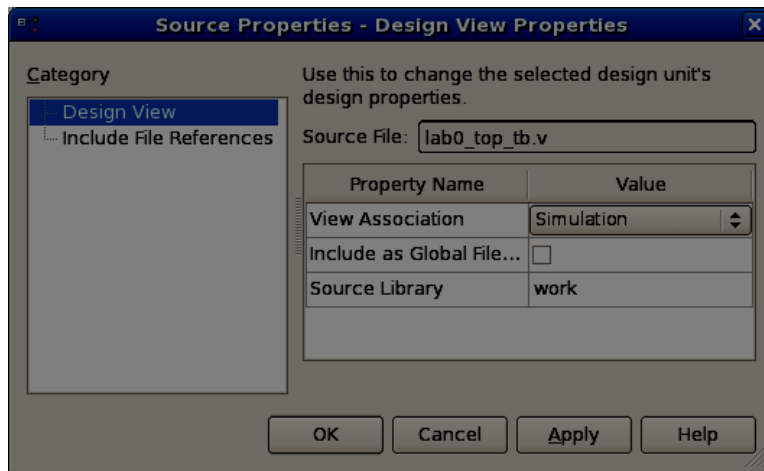
Correcting the testbench's association
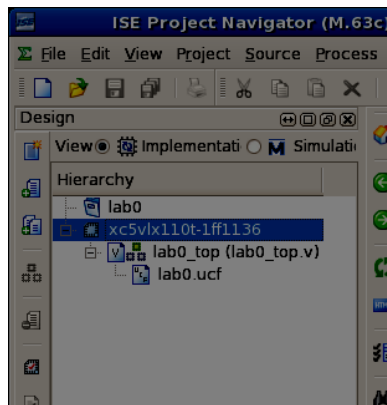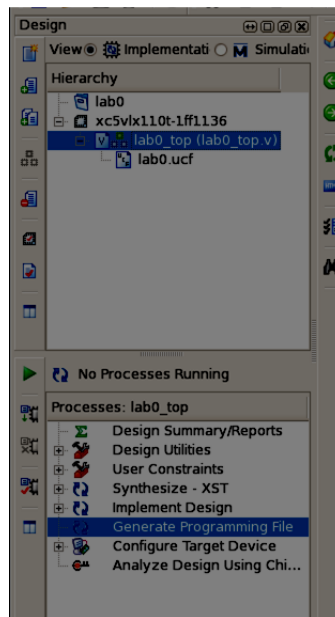

What your heirarchy should look like
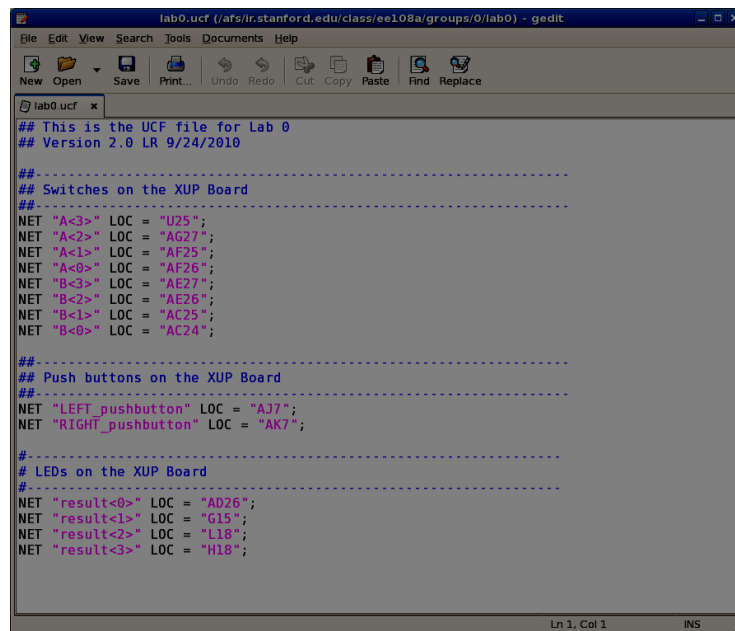
Contents of the UCF file

Now we're finally ready to synthesize.

Synthesizing

Select the 'lab0_top' module and double-click on 'Generate Programming File' in the Processes pane. After a long wait, with progress you can watch in the transcript below, your project should finish synthesizing!


Synthesizing our top-level module

At this point, we should have a **.bit file** which is sort of like an executable program for an FPGA. It contains our final synthesized design which we'll download to the FPGA in a moment. It's important that you make sure you don't have any errors or warnings after this process before continuing. To check them click on the Errors and Warnings tabs at the bottom and resolve any issues.

Once everything looks good, double-click on 'Configure Target Device'. This will launch a program that communicates with our FPGA board. You may get a warning saying that there is no iMPACT project file, but just click OK.
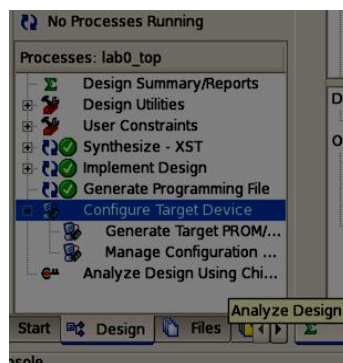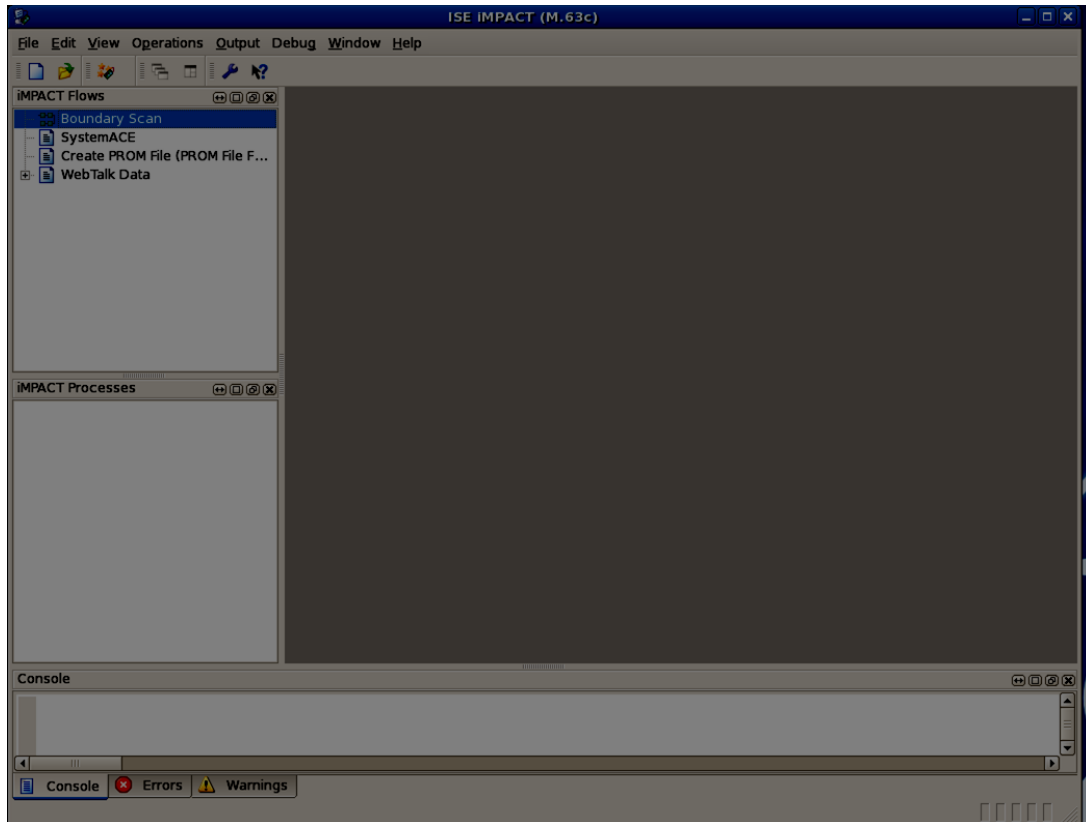

*Green checkmarks and launching iMPACT*
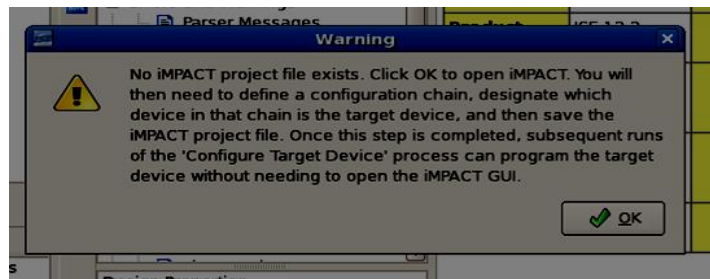
Yeah okay whatever

Once open, you should double-click on the "Boundary Scan" item in the iMPACT Flows pane.

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".
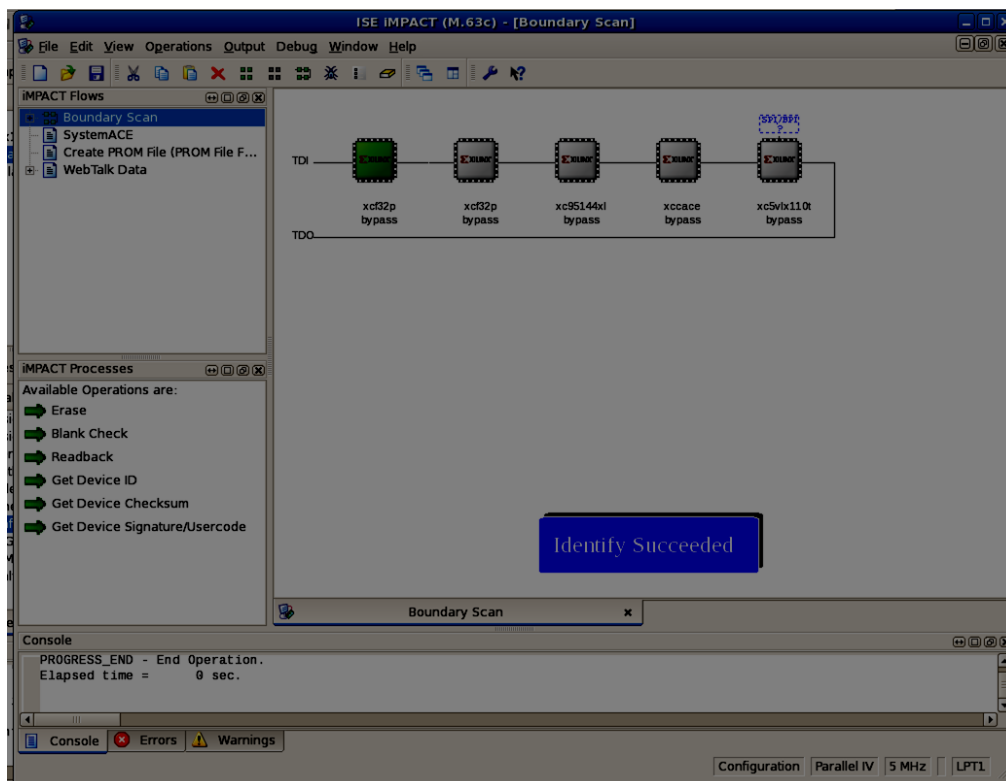
Right click on the device that is labeled "xc5vlx110t" and select "Assign New Configuration File". Navigate to and select the "lab0_top.bit" file that sits in your lab0 directory. When prompted, do NOT attach SPI or BPI PROM to the device:
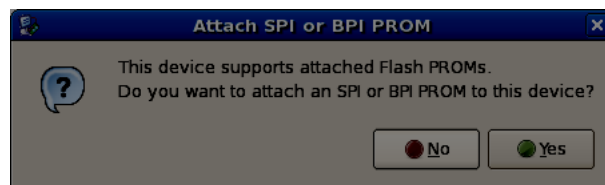

NOPE

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.

Once it's done downloading and you see the 'Program Succeeded' box, it's time to test out your design! Go ahead and play with the left and right buttons (the left and right push buttons on the FPGA board diagram below) and the 8 switches (the slide switches in the board diagram below). You should see the results on the 8 LEDs to the top of the 8 slide switches on the FPGA board.

You're done!

01/07/16   EE108 Lab 0: Introduction to Verilog
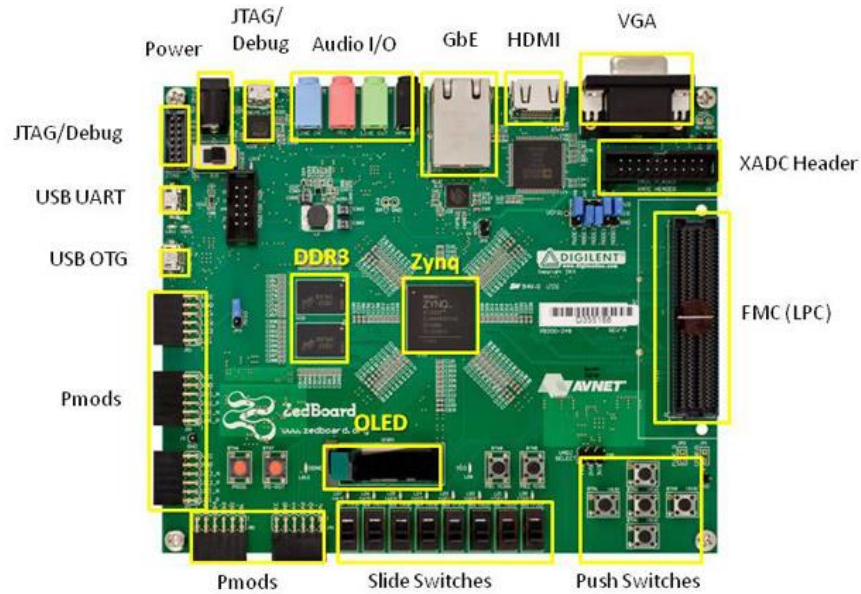
* SD card cage and QSPI Flash reside on backside of board

All the widgets on our development board. Mostly we'll be concerned with the push switches and the LEDs above the slide switches.

5. Other topics

Inferring a latch, just to see what that's like

Let's go back to lab0_top.v and remove the 'default' and '2'b00' clauses from the case statement. Now, our module's case statement looks like this and its behavior is undefined when left_pushbutton and right_pushbutton are both 0 or both 1:

```
always @(*) begin
  case (select)
    2'b01: out = added_result;
    2'b10: out = anded_result;
  endcase
end
```

If we run our testbench again we should see some red in the waveform:
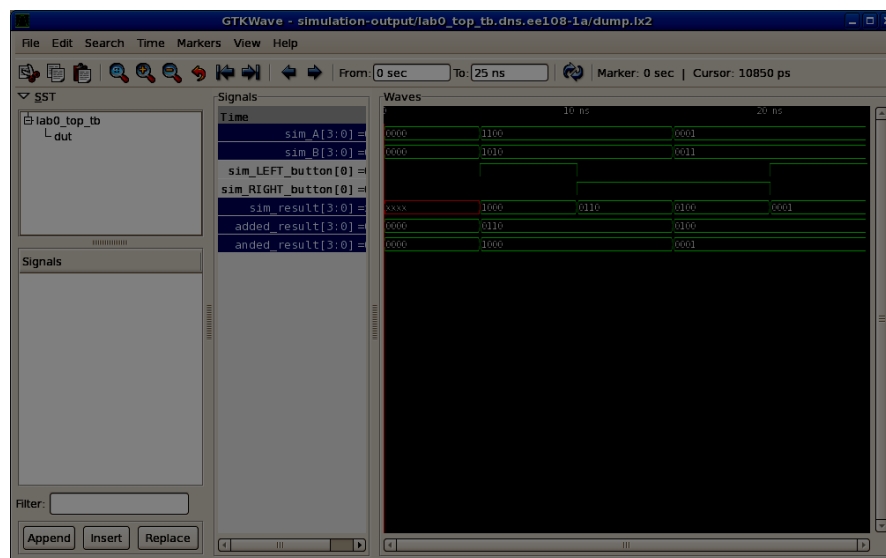
Uh oh

Red X's in simulation means the value of the signal was undefined for that stretch of time, and is a very strong indicator we inferred a latch somewhere. The physical analogue of this is the bit could be 1 or 0, and we have no way of predicting which. If you ever encounter these red X's, you should take extra care to understand where they are coming from as they likely mean you're doing something wrong.

You may also see blue signals labeled 'HiZ' – this indicates the wire or reg we defined is never assigned anything at all. In real life this is a dangling wire which does not connect to the rest of the circuit, giving it a very high impedance (a high Z value).

This often comes up when we misspell variable names and Verilog infers that these typos are 1-bit wires. They aren't connected to anything, since we didn't even intend for them to exist, so they show up as HiZ in the simulator.

If we try to synthesize with this missing case, ISE will report to us that latches are being inferred. After synthesizing, clicking on the 'Warnings' tab will show us something like this:

```
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.
```

ISE reporting an inferred latch

You should never see these warnings in a correct lab solution.

The difference between & and &&

Verilog has a set of bitwise operators (&, |, !) and a set of logical operators (&&, ||, ~). If we're only operating on one-bit values, they both work identically. But once we use them on multi-bit signals, differences arise:
- Logical operators always output a result that is a single bit, *true* or *false*.
- Bitwise operators always output a result that is the same bit width as their inputs.

We can think of 3'b101 & 3'b011 as

	3'b	1	0	1
&	3'b	0	1	1
	3'b	0	0	1

While 3'b101 && 3'b011 is

	3'b	1	0	1	→	1'b1
&&	3'b	0	1	1	→	1'b1
						1'b1

The same follows for OR (|,||) and INVERT/NOT (~,!). Logical operators really try to boil the world down into true or false. Any number that's not 0, as far as they're concerned, is 1.

On a slightly related topic, we can AND, OR, or XOR all of the bits of a signal together by placing the signal to the right of the corresponding operator and nothing to the left:

```
wire [2:0] woop;
wire       anded_signal_1;
wire       anded_signal_2;

// Equivalent:
assign anded_signal_1 = woop[0] & woop[1] & woop[2];
assign anded_signal_2 = &woop;
```