DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2016

Lab 1: Introduction to Verilog, the Tool Chain and the DE1-SoC Board

Week of September 12 to 16

1 Introduction

Lab 1 is a *fun* walk-through of the language and tools we will use in Labs 3 through 7, namely:

- **Verilog** (the hardware description language we'll use to describe our designs),
- **ModelSim** (a program that takes our Verilog descriptions of digital circuits and simulates them to help find design errors), and
- **Quartus II** (a program which synthesizes our Verilog descriptions of digital circuits into logic gates that are implemented on the *field-programmable gate array* (FPGA) on your DE1-SoC).

Some important points to keep in mind before continuing:

- *To work with a partner you* **must** *sign up on Connect before the sign-up deadline.*
- *This lab may take you 6 hours. In CPEN 211 you must complete labs at home or in MCLD 112* **before** *your scheduled lab time in MCLD 112 which is reserved for demonstrating completed lab work.*
- *Please read and sign a copy of the "CPEN 211 Lab Peer Help Policy" now so you know the permitted ways to obtain help from your peers. Put it somewhere you will remember as your TA will not mark you until they get your signed copy. The TAs will have extra copies, but may run out.*
- *Remember the goal is to have fun! A complete marking scheme is provided in Section 3.*

Given we have not covered much in class yet, this lab assumes only that you have taken an introductory programming course introducing C syntax, such as APSC 160, that you have seen some very basic electronic circuits such as those studied in PHYS 158 (e.g., including voltages, resistors and capacitors), and that you understand the concept of a function (e.g., as seen in MAT 100). This lab involves reading, installing software, typing the Verilog code fully provided and described in this handout into a file, and then using these to program your DE1-SoC. Even if you normally shy away from lab work, I suspect you will find this lab interesting and worthwhile. ***The more that you manage to learn from this handout before Lab 1 the more you will subsequently learn when we get to related topics in lecture.***

As illustrated by this handout, the general process we will follow in Labs 3 to 7 is to:

1. Review a high-level specification of a "digital" circuit, then
2. write Verilog that describes the desired circuit, then
3. use ModelSim to identify errors by "simulating" the circuit, then
4. synthesizing the Verilog to implement the circuit on your DE1-SoC using Quartus II, then
5. "download" your design to your DE1-SoC using the Quartus II device programmer, then
6. test the actual circuit implements the desire specification using the switches and light emitting diodes (LEDs) on the DE1-SoC.

This lab will also introduce you to a very common design error called an *inferred latch*. By *design error* we mean an error that occurs when we have used correct syntax but have described the wrong thing. We introduce this error to you now because it is easy to identify and fix, because learning how to avoid it can save you time later, and because it illustrates an important concept.

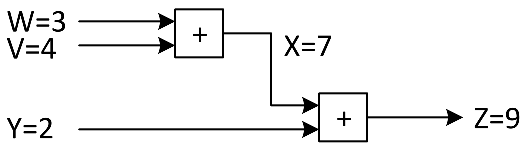


Figure 1: Interpreting Verilog code as specifying wires connecting electronic circuits.

1.1 What is Verilog?

Electronic devices such as smartphones and tablet computers process information. To do this, they use digital logic circuits. A *digital logic circuit* is a type of electronic circuit that represents this information using 1's and 0's encoded as voltages. To design a digital logic circuit an engineer could specify each individual component of the circuit manually. This approach was used many decades ago when circuits contained relatively few components. However, as a result of Moore's Law, Today's electronics typically contain billions of individual electronic components. To be able to design a single circuit containing billions of components each engineer needs to design a portion of this circuit containing potentially millions of individual components. The current approach used to achieve this is to use a hardware description language.

Verilog is one such a *hardware description language* (HDL). An HDL is a language that can be used to describe digital logic circuits. As you will see, Verilog syntax looks quite similar to "C". Perhaps because many engineers learn "C" at some point in their career and their syntax has similarities, Verilog has become the most popular HDL for industry use. However, the way Verilog "works" is very different from software programming languages like "C". In Verilog "variables" become wires. Operations on those variables become hardware blocks. As a result, all assignments in Verilog will be evaluated **concurrently**. This is a very important concept, yet for many people it is counterintuitive—at least initially.

For instance, in APSC 160 when learning C you may have seen something like:

```
Z = X + Y;
X = W + V;
```

If you saw this in "C" you would say, "Ah! I know what's going on: `Z` takes on the sum of variables `X` and `Y`. Then, a new value of `X` is computed using `W` and `V`." In "C", which is a sequential programming language, you'd be right. For example, if `X`, `Y`, `W` and `V` were initially 1, 2, 3, and 4 respectively, then after executing the above code `Z` would be 3 and `X` would be 7. However, if it was Verilog, and it was written in this way:

```
assign Z = X + Y;
assign X = W + V;
```

then the previous reasoning would be wrong! All *assign statements* in Verilog happen at the same time, so, in the above example, `X` is always taking the value of the sum of `W` and `V`. If `W` and `V` are initially 3 and 4 then `X` would have to be 7. It is this sum that is then added to `Y` to produce `Z`. So if `Y` is 2, then `Z` would be 9 (instead of 3). The order of assign statements does NOT matter because they describe the behavior of actual wires that are always being evaluated—they have no concept of sequence. The correct way to "think about" or "visualize" the above two lines of Verilog is as connections between blocks as illustrated in Figure 1. The way to interpret Figure 1 is that numbers "flow" along the arrows into the blocks. You may wonder if we can force `X` to 1 somehow. The only way to do this would be if the sum of `W` and `V` was 1.

Verilog also contains special syntax to indicate that order between statements *does* matter. We will see our first example of this syntax when we discuss testbenches later in this lab.

1.2 What is an FPGA?

Your DE1-SoC contains a field-programmable gate array (FPGA) that you will use to implement and test your designs. An FPGA is an array of *logic elements* that are "programmable." What this means is that it can be used to build any circuit we want. Each logic element includes a hardware block called a *lookup* *table*. The lookup table can implement any function of four input variables where each input can be either a 0 or 1 and the function output is either 0 or 1. One logic element of the FPGA could perform a portion of the addition of two 2-bit numbers (recall that a bit can be 0 or 1) or it could determine when corresponding bits of two 2-bit numbers are both equal to 1 (known as a *bitwise AND* function). Some parts of the FPGA can also be used to remember numbers or drive input and output pins of the FPGA to communicate to the outside world. Again, their behavior depends on how they are programmed. You might wonder if you could also implement a circuit as a discrete chip using a Verilog description. The answer is "Yes!" You will build up an understanding of how that can be done if you take all of CPEN 311, ELEC 402 and EECE 403.

2 Lab Procedure

2.1 Step 0: Installing Quartus II 15.0 Web Edition and ModelSim-Altera

First, let's download and install Quartus and ModelSim. This outline assumes you have Windows 7 or 10 on your computer. We will use Version 15.0 of Quartus II Web Edition. It is important to use the "Web Edition" as it does not require purchasing a license. It is important to use version 15.0 as your TAs will mark your files using this version. ModelSim-Altera, which you will make heavy use of in this course, does not work properly with Windows 8. Windows 7 is recommended and available free to UBC students via DreamSpark (see https://help.ece.ubc.ca/Microsoft_Software_Available_to_Students,_Faculty,_and_Staff_through_Dreamspark). To obtain Quartus, first go to https://www.altera.com/myaltera/mal-index.jsp to get a myAltera account. Then, visit http://dl.altera.com/15.0/?edition=web to download the software. 14 GB of free disk space are required. If you have never used Cygwin or 7-Zip we recommend you select the tab to download individual components. In that case be sure you download *Quartus II Web Edition*, *ModelSim-Altera Edition* and *Cyclone V device support*. If you know Cygwin or 7-Zip, download the combined file and extract using 7-Zip or with "tar xvf Quartus-web-15.0.0.145-windows.tar" in Cygwin. Then run QuartusSetupWeb-15.0.0.145-windows.exe to install all the components (this will be in a directory "components" if you downloaded the .tar file and extracted using Cygwin or 7-Zip). You may see a security dialog. If so, select "Yes" to allow the install program to run. Use the default install location. If you do otherwise, ensure there are NO spaces in the path name. For those using Mac OS X, the following video created by Andrew Simpson (an undergraduate TA for CPEN 211 who took the course last year), walks through how to download and install VirtualBox, Windows 7, Quartus and ModelSim: https://www.youtube.com/watch?v=YPeyq5VKAzQ.

2.2 Step 1: Design of a Simple Circuit

In this lab we design a circuit that uses the switches labeled SW0 through SW7 on the DE1-SoC as two 4-bit inputs that we will call "A" and "B". The design performs simple operations on A and B. We will output the result to 4 light emitting diodes (LEDs) on the FPGA board. We implement two functions: "A & B" and "A + B" (where & means bitwise-AND, and + means addition).

2.2.1 Specifying the design

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

- When the "left" pushbutton (labeled KEY1) is pressed, the output LEDs (labeled LEDR0-LEDR3) should show switch inputs A (labeled SW0-SW3) bitwise-ANDed with switch inputs B (labeled SW4-SW7). This means each output LED lights up only if the corresponding input switches for both A *and* B are in the "on" position. On the DE1-SoC this will mean that LEDR0 will light up only if both SW0 and SW4 are up, LEDR1 will light up only if both SW1 and SW5 are up, and so on.

- When the right pushbutton (labeled KEY0) is pressed, the output LEDs should show switch inputs A added to switch inputs B. The four LEDs (LEDR0-LEDR3) display the 4-bit sum of the 4-bit numbers you input in binary on the switches A and B. By *binary* we mean 0 is represented as 0000, 1 as 0001, 2 as 0010, 3 as 0011, 4 as 0100, 5 as 0101, etc...

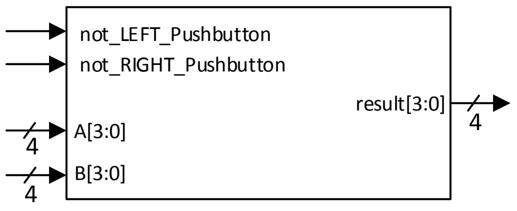


Figure 2: Circuit Interface Specification

Signal	Direction	Width	Purpose
not_LEFT_pushbutton	Input	1 bit false (0) = pushed true (1) = not pushed	When the left button (KEY1) is pushed we show A bitwise anded with B (e.g., if A=1101 and B=1010 then result=1000)
not_RIGHT_pushbutton	Input	1 bit false (0) = pushed true (1) = not pushed	When the right button (KEY0) is pushed we show A plus B (e.g., if A=0001 and B=0011 then result=0100 which is 4 in binary)
A	Input	4 bits (binary 0000-1111)	This is our A input from the first 4 switches on the board (SW0-SW3)
B	Input	4 bits (binary 0000-1111)	This is our B input from the second 4 switches on the board (SW4-SW7)
result	Output	4 bits	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 1 Top Level” block.

- Pressing both buttons yields the same result as only pressing the right pushbutton.
- If neither button is pressed, the red LEDs should not turn on.
- The pushbuttons use negative logic: “pressed” is represented with ‘0’ and “not pressed” with ‘1’.

The description above defines the inputs and outputs between the external world and the circuit. To clarify the specification we can draw a top-level block diagram in Figure 2 which omits internal components but identifies inputs and outputs. We can also organize the information in the specification based upon input and output signals as shown in Table 1.

2.2.2 Circuit Design

Next, *before* writing any Verilog, we *design* a circuit meeting our specification. Conceptually, we do this by thinking how to achieve the desired behavior by combining predefined blocks of hardware. As this step requires knowing what predefined hardware blocks exist and how they work, and since we haven’t covered that yet, we do this for you in Lab 1. Figure 3 shows our completed design. Our specification says the circuit needs to calculate two functions: bitwise-AND and addition so we need to include blocks that do this. In Figure 3 the bitwise-ADD required by our specification is performed by the symbol labeled “AND”. The addition is performed by the block labeled “ADD”. In the figure, lines represent wires. The “4” labeling the bottom two wires on the left indicates these lines are actually four separate wires bundled together into a *bus*. Our specification also says the value on the result output should depends upon the push buttons. You may have some ideas about how to implement similar behavior in “C” using an *if* statement, and given Verilog looks like “C”, you might be tempted to immediately write some code to capture this idea. However, a very important difference when designing hardware (versus software) is that we **must** first identify what type of hardware can implement the behavior **before** writing code. In this case, a good

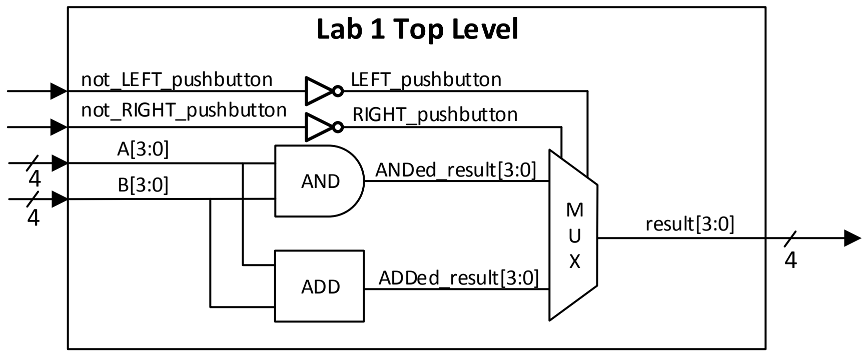


Figure 3: Block diagram showing internal logic.

```
module lab1_top (
    input not_LEFT_pushbutton,
    input not_RIGHT_pushbutton,
    input [3:0] A,
    input [3:0] B,
    output reg [3:0] result );

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

Figure 4: Module declaration

hardware block for implementing the desired behavior is a *multiplexer*, represented Figure 3 by the trapezoidal symbol labeled “MUX”. For now you can think of this multiplexer as “copying” either the 4-bit value on `ANDed_result[3:0]` or `ADDed_result[3:0]` to the 4-bit output `result[3:0]`. If only the wire labeled `LEFT_pushbutton` is set to 1, then the top input to the MUX is copied and if only the wire labeled `RIGHT_pushbutton` is set to 1, then the bottom input is copied. Finally, the triangle with a little circle on one corner is the symbol for a NOT gate, which outputs a 1 if the input is 0, and a 0 if the input is 1.

2.2.3 Declaring the module interface in Verilog

After finishing the design, we start writing Verilog. The first step is coding up the external interface to the design, after referring to the top-level block diagram in Figure 2. The result is shown in Figure 4. This code is referred to as a *module declaration*. The Verilog reserved words **`module`** and **`endmodule`** identify the beginning and end of the declaration. We choose the name “`lab1_top`” to identify this module. **As explained in Section 2.3, modules are NOT the same as software functions.**

In Figure 4 the notation `[3:0]` applied to inputs `A`, `B` and output `result` is a bit-depth specification. It means these signals are each 4-bits wide. Moreover, it indicates that 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. In some respects, A, B and result can each be considered an array of bits, with the first bit having an index of zero. Both “`not_LEFT_pushbutton`” and “`not_RIGHT_pushbutton`” are a single bit as they do not include a bit-depth specification. Notice `result` is declared as **`reg`**. Students are often unsure when adding **`reg`** is required. We use **`reg`** for `result` because `result` will be assigned from an “always block”.

While you can cut and paste it from the PDF, you will benefit if you retype the code in Figure 4. Use a text editor specifically designed for coding. Both ModelSim and Quartus include editors you can use to create .v files. Expert engineers typically prefer using either “vi” or “emacs” (these might also be the only “correct” answers to the interview question, “which text editor do you prefer?”). Vi and emacs are included on both Linux and Apple (accessible via the terminal). On MCLD 112 computers you can access vi and

```
wire [3:0] ANDed_result;
wire [3:0] ADDed_result;
wire LEFT_pushbutton;
wire RIGHT_pushbutton;
```

Figure 5: Internal signal declarations

```
assign ANDed_result = A & B;
assign ADDed_result = A + B;
```

Figure 6: AND and ADD operations using assign statements

emacs via Cygwin, which you can install at home from Cygwin.com. Whichever editor you use, save the code from Figure 4 in a file called `lab1_top.v` in a directory or folder called "lab1". Note common errors are forgetting to save changes in ModelSim before compiling or making changes to the wrong copy of a file.

2.2.4 Describing the Internal Logic in Verilog

Next, we write a Verilog description of the circuit designed in Section 2.2.2 and captured in the block diagram in Figure 3. We start by creating the wires we will need to connect various blocks with the code shown in Figure 5. Next we create the ADD and AND operation blocks. To do this, we will make use of the fact that in Verilog we can drive a value onto a wire using an *assign* statement as shown in Figure 6. Note that the signal on the left side of "=" in an assign statement *must* be declared with **`wire`**. As alluded to in Section 1 with reference to Figure 1, regardless of what occurs elsewhere in the circuit the assignment is continuously driving a value onto the wires on the left side of the "=".

Next, recall the push button switches use negative logic meaning when we are not pressing the button, the output is true or '1' and when we press the button, the value is '0'. Since this is the opposite of how we normally think about pressing a button, we "inverted" these inputs using the two not gates shown in Figure 3. We create these not gates with the Verilog in Figure 7 where ~ is the bitwise-NOT operator in Verilog.

Finally, we just need to create the multiplexer. As we will see in class, there are many ways to describe a multiplexer in Verilog. For our purposes, a convenient approach is using an "always block" *similar* to that shown in Figure 8. An *always block* starts with the reserved word **`always`**. This specific always block contains some code enclosed by a **`begin`** and **`end`**. To help explain the remainder of the Verilog code in Figure 8 we describe how a tool like ModelSim *simulates* hardware described by the code between **`always`** and **`end`**. **However, the always block only describes the desired *behavior*; Quartus will configure a corresponding circuit using the logic elements inside the FPGA on your DE1-SoC.** When ModelSim simulates this Verilog, the "`@*`" tells it to simulate the logic between **`begin`** and **`end`** whenever *any* signal read between **`begin`** and **`end`** changes. Thus, any time that `LEFT_pushbutton`, `RIGHT_pushbutton`, `ADDed_result`, `ANDed_result`, or `ADDed_result` changes the code between **`begin`** and **`end`** is immediately evaluated regardless of whatever else is going on in other parts of the circuit. This code contains a *case* statement that begins with the reserved word **`case`** and ends with **`endcase`**. You can think of a case statement as somewhat like an *if* statement in "C" (it is actually somewhat closer to a "C" *switch* statement that a few of you may have heard about). The code inside the parenthesis is used to select *one* of the three lines below to evaluate. The curly braces `{}` are used to "sandwich signals together" in Verilog. Specifically, we combined the 1-bit wires `LEFT_pushebutton` and `RIGHT_pushbutton` and create a 2-bit value. On the next line, the 2 in the

```
assign LEFT_pushbutton = ~ not_LEFT_pushbutton;
assign RIGHT_pushbutton = ~ not_RIGHT_pushbutton;
```

Figure 7: Converting push button switches to positive logic

```
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
```

Figure 8: Verilog describing MUX (note, this Verilog contains an error we will later correct)

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

Figure 9: Assignment statement describing a mux - NOTE: DO NOT ADD to your Verilog file

funny notation `2'b01` indicates a constant value that is 2-bits wide. The `'` is an apostrophe (same key as the quotation mark on most keyboards). The "`b`" means the constant is specified in binary. The remaining part, `01`, is the actual constant value. For example, if `LEFT_pushebutton` is 0 and `RIGHT_pushbutton` is 1, then the hardware block copies the value from the bus `ADDed_result` to `result`. The code in Figure 8 has a subtle yet very common error called an "inferred latch". Take a moment now to see if you can *guess* what is *missing*. We will explain the "inferred latch" problem and its solution in Section 2.4.

In Figure 4 we declared `result` as both **`output`** and **`reg`**. We added **`reg`** because `result` is *set* inside an always block. If we set `result` with **`assign`** then `result` would need to be a **`wire`** as assign statements define wires. We *can read* **`reg`** and **`wire`** signals in both always and assign statements. For example, instead of using an always block we could have instead used an assign statement as shown in Figure 9 where the syntax "`a?b:c`" returns `b` if `a` is 1 and `c` if `a` is 0. This does roughly the same thing as our always block, but will act slightly different. The difference is intentional and will see the effect later in the lab. For now, use the version in Figure 8 and save any changes to `lab1_top.v`.

2.3 Step 2: Testing your design in ModelSim

At this point we could compile our Verilog in Quartus and download it to the FPGA on the DE1-SoC. However, if the Verilog contains an error it will be harder to find on the FPGA because we only have access to the inputs and outputs. Also each time we change our Verilog it takes several minutes to compile it with Quartus. We address both limitations using ModelSim to test our Verilog.

Start ModelSim by going to the windows start menu and typing "modelsim" in the "Search" dialog at the bottom then selecting ModelSim-Altera. Once ModelSim opens, create a new project for Lab 1 by going to "File -> New -> Project ..." The project name is "lab1" and the location should be in the lab1 folder you just created. The "Add items to project" dialog will be shown. If you already created `lab1_top.v` then add it using "Add Existing File". If you haven't yet created it then create it using "Create New File". If the dialog does not show (or you closed it), then click in the project window on the left or select "Window->Project", then select "Project -> Add to Project -> ..." to add a new or existing file.

Next, compile the Verilog by going to the "Compile" menu and selecting "Compile All" as illustrated in Figure 10. You should see some text generated in the "Transcript" window. If you have errors they will be highlighted in red. (In general, if you get error messages you do not understand, you should search online for what they mean.) Now the file is compiled for the simulator, but we can't actually test it yet.

2.3.1 Conceptual Description of a Test Bench

To test `lab1_top` we need to generate inputs representing the positions of the switches on the DE1-SoC. We do this with a *testbench* illustrated in Figure 11, which you should study carefully. A testbench evaluates

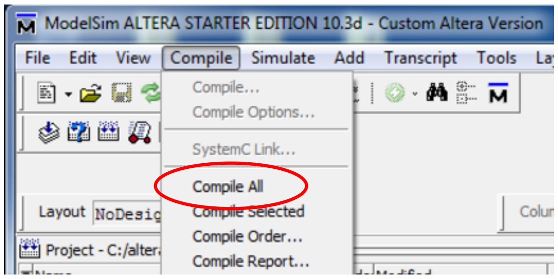


Figure 10: Compiling the project

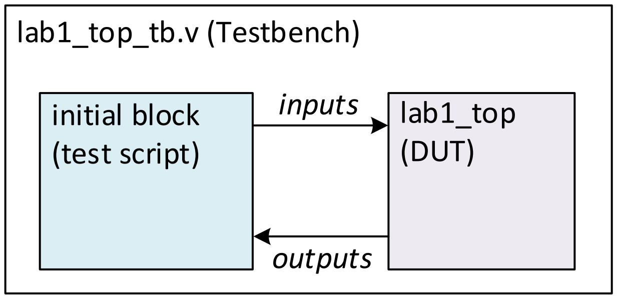


Figure 11: Testbench conceptual illustration

whether a *device under test* (DUT) works as expected. In Lab 1, the DUT is an "instance" of the `lab1_top` module. The testbench feeds inputs from a *test script* into the DUT and monitors the DUT outputs.

In industry, where designs get very complex, test scripts are often written in "C" or "C++" and interfaced to the DUT using something called the Verilog *Programming Language Interface* (PLI). To keep our focus on concepts we use Verilog to specify our test scripts. However, as we will not generate hardware for the testbench we are free to use more flexible syntax in our test scripts. Thus, our test scripts will contain code that runs in the sequence the code is written, but unfortunately we cannot use that style to describe hardware.

2.3.2 Designing Test Scripts

A very common question students ask is, "How do I decide what input values to set in the test script?" The challenge is having too many possible input combinations to be able to test *all* of them. For example, consider how we would test if our add statement is correct. How many possible 4-bit numbers can we add together? Well, 2^4 is 16, and we have two inputs, A and B, so we have $16 \times 16 = 256$ possible additions. We can test 256 input combinations fairly easily. However, the adder in your computer takes in two 64-bit values. There are 1.84×10^{19} possible choices in a 64-bit value, and with two of them 3.4×10^{38} possibilities. If we had a computer that could test 100 billion possibilities a second that would take 1×10^{20} years to finish! That is roughly 7.8 million times the age of the universe! So, it's unlikely the manufacturer actually tested all the possibilities. To keep the number of input combinations small *and* catch errors, we rely on understanding the circuit to figure out what is important to check. For example, it is not too difficult to check that when we press the buttons we get the AND or the addition of the values for a few values of A and B. If our logic adds 5 and 6 correctly, we can argue it is likely going to add 5 and 7 correctly.

2.3.3 Writing the Lab 1 testbench

Create a file `lab1_top_tb.v` and add it to your project in ModelSim like you did for `lab1_top.v`. Just like any other Verilog module, a test bench is a module, but it doesn't have any inputs or outputs. To start our testbench add the code shown in Figure 12 in `lab1_top_tb.v`. Next, add declarations to the module for signals used to generate inputs and capture the outputs of our DUT as shown in Figure 13.

Next, we need to create a copy of our DUT and connect it to these signals. To use a circuit, whether

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

endmodule
```

Figure 12: Testbench module declaration

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

wire [3:0] sim_result;
```

Figure 13: Declare signals to connect to device under test (DUT)

in a testbench or inside another circuit we need to make a copy or "instance" of the circuit. The difference between a module definition, such as for `lab1_top` in Section 2.2.4, and an instance of that module is like the difference between the picture of the DE1-SoC from the Terasic ordering website (the "definition") and having your own DE1-SoC so you can demo Lab 1 (one of many potential "instances" of the DE1-SoC you will see in MCLD 112). When we instantiate a module inside another module we are telling Verilog to build a copy of the module inside the new module. While the syntax for instantiation (shown in Figure 14) *looks* like a function call in "C" the meaning is ***completely*** different. In "C" 5 calls to a function run the same code 5 times, one after the other. In Verilog, instantiating a module 5 times means creating 5 copies that exist at the *same* time and take up 5 times the space (FPGA logic elements) versus one module instance.

Figure 14 shows how we instantiate `lab1_top`. The label `DUT` identifies the instance, but any other identifier could be used. The remaining code connects inputs and output of the `lab1_top` instance to wires in the tesbench module using a dot notation syntax we suggest for all labs: The notation "`.X(Y)`" connects an input or output, `X`, declared in the module being instantiated with a **`wire`**, `Y`, declared in the testbench or module in which it is instantiated. We can easily see `sim_A` connects to `A` in the `lab1_top` instance. We connect `not_LEFT_pushbutton` to `sim_LEFT_button` after first applying the bitwise not operator ~.

Next, we add the test script that defines the input signals to the DUT using an *initial* block. Initial blocks can only be used in testbenches (they do not synthesize to logic). When simulation starts, Verilog steps through all initial blocks. You can have multiple initial blocks, but make sure you don't try to set the same signal from multiple places or you'll get confused. Figure 15 contains the initial block we will use to start.

In an initial block the lines between **`begin`** and **`end`** are evaluated one at a time in the order found. In our test script, `sim_LEFT_button` is assigned the 1-bit value of 0 (that's what `1'b0` means) in parallel with `sim_RIGHT_button` being assigned 0, as well as `sim_A` and `sim_B` being assigned values of 0 (but this time 4-bit of zeros, which is what `4'b0` means). The delay (`#`) tells the simulator to run for a given number of time steps before evaluating the next statement. The `$stop` command tells the simulator to stop, and

```
lab1_top DUT (
  .not_LEFT_pushbutton(~sim_LEFT_button),
  .not_RIGHT_pushbutton(~sim_RIGHT_button),
  .A(sim_A),
  .B(sim_B),
  .result(sim_result)
);
```

Figure 14: Instantiating the lab1_top module

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

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

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

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

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

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

Figure 15: Initial test script

`$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`).

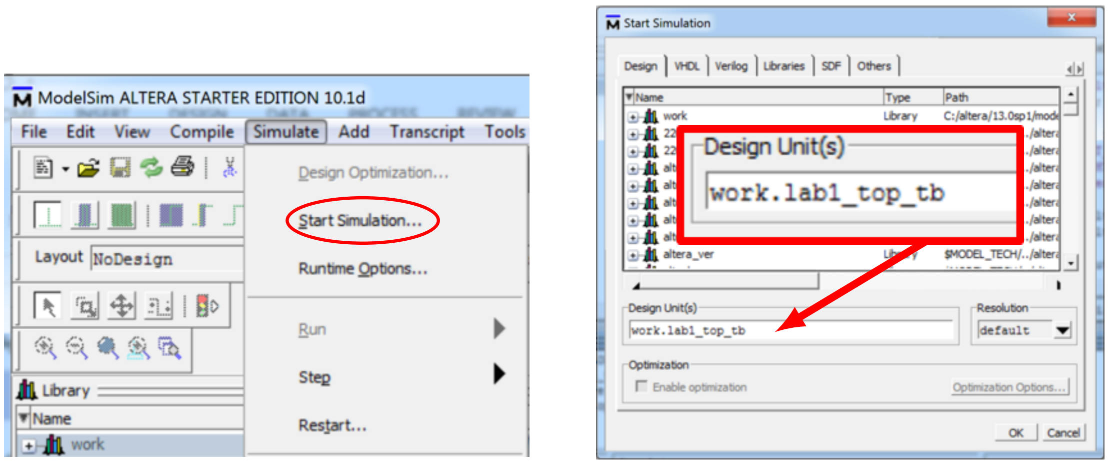
2.3.4 Simulating lab1_top with the testbench in ModelSim

Save `lab1_top_tb.v` then compile using "Compile -> Compile All". Next, start the simulation by selecting "Simulate -> Start Simulation..." (see Figure 16a) to bring up the "Start Simulation" dialog box illustrated in Figure 16b. Enter `work.lab1_top_tb` in the "Design Unit(s)" text box, and then press "OK".

ModelSim should look something like Figure 17. At this point the simulation has started, but is paused at time zero. When we simulate it will be helpful to see the values that different signals within our circuit take on at different times. To enable us to do this, ModelSim has a waveform viewer. You can open the "Wave" window by going to the "View" menu and selecting "Wave". The Wave window should look something like Figure 18. Currently, there are no signals showing in the window so as simulation time advances we will not see what goes on inside our circuit. To remedy this we add signals to the waveform window as follows.

The "Objects" window and "Instance" window (see Figure 17) work together. The "Objects" window shows the signals within the module instance highlighted in the "Instance" window. In Figure 17 the `lab1_top_tb` module is highlighted in the "Instance" window so the signals declared in `lab1_top_tb` (see Figure 13) are listed in the "Objects" window. Left click on the first signal listed in the "Objects" window (e.g., `sim_A`), then press shift and left click on the last signal in the Objects window (e.g., `sim_result`) to select all the signals. All the signals should be highlighted as shown in Figure 19b. Next, go to the "Add" menu and select "To Wave -> Selected Signals" as illustrated in Figure 19b.

So far, we have not *really* done anything we could not have done on the DE1-SoC because all the signals we have added to the waveform viewer are at the top level of our design. The real power of ModelSim is that we can look *inside* our circuit during simulation. To do this, go to the "Instance" window and select "DUT". **Notice that the signals listed in the "Objects" window changed when we did this!** Now it shows the signals *inside* the DUT instance, which is something we cannot easily see when we put our circuit on



(a) Opening simulation dialog

(b) Start simulation dialog box

Figure 16

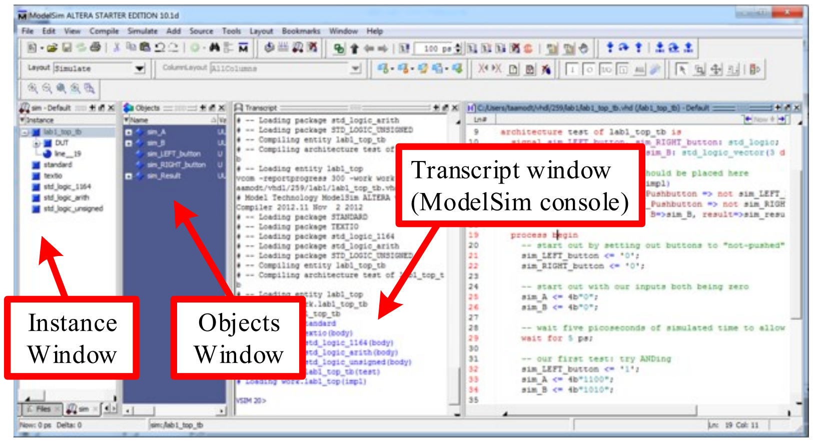


Figure 17: After starting simulation of work.lab1_top_tb (your view might differ)

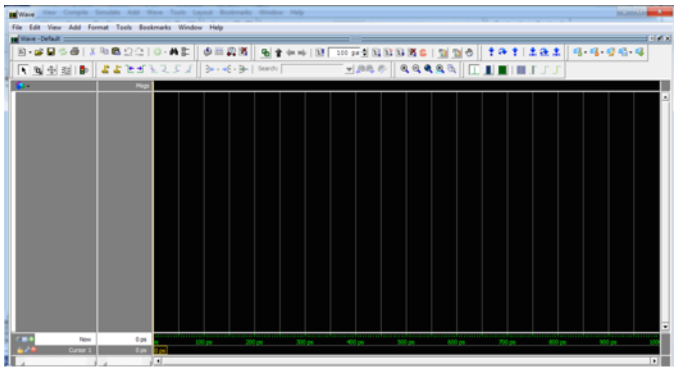
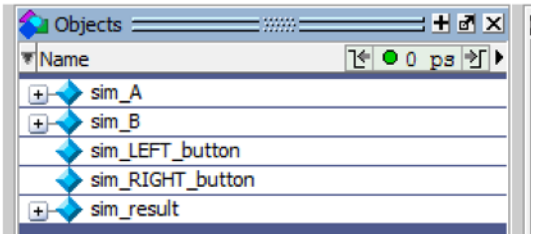
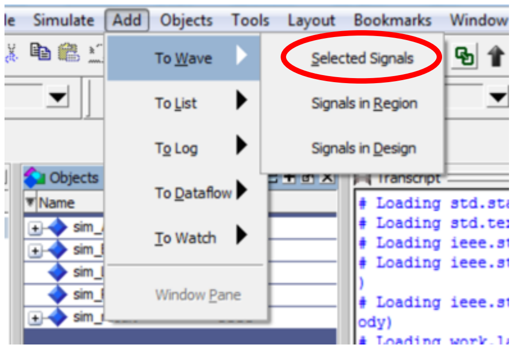


Figure 18: Waveform viewer window

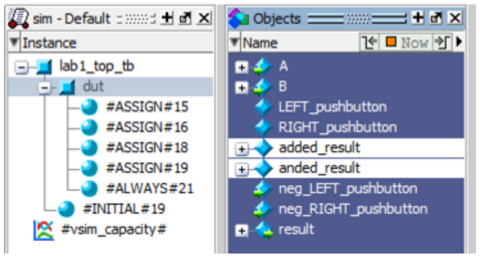


(a) Selected signals

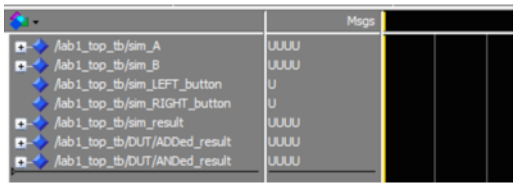


(b) Adding signals to waveform viewer

Figure 19



(a) Add internal signals to wave viewer


Msgs
/lab1_top_tb/sim_A UUUU
/lab1_top_tb/sim_B UUUU
/lab1_top_tb/sim_LEFT_button U
/lab1_top_tb/sim_RIGHT_button U
/lab1_top_tb/sim_result UUUU
/lab1_top_tb/DUT/ADDed_result UUUU
/lab1_top_tb/DUT/ANDed_result UUUU

(b) Signals added to waveform viewer

Figure 20

the DE1-SoC. Select "ADDed_result" and "ANDed_result" as illustrated in Figure 20a and add them to the wave viewer. You should now see some signal names listed in the Wave window as illustrated in Figure 20b. Go "File -> Save Format ..." in the wave viewer window and save a wave.do file. You can avoid needing to repeat the process of adding signals later by reading the wave.do file with "File -> Load ...". Optionally, see if you can figure out how to organize the signals with dividers and/or changing their format.

To run the simulation choose "Simulate -> Run -> Run-all". The output in the "Transcript" window should look like Figure 21. You should see "Output is 1000, we expected 1000". The Waves window shows a graph of signals verse time and should look like Figure 22a. Our testbench simulates only 10 picoseconds because we have two #5 statements. To see what happened, zoom in by clicking in the wave pane, and typing + (using the shift key as well). Or use "Zoom Full" and then "Zoom In on Active Cursor" to magnify the details (these are buttons above the waveform window). Note the signal names have leading text describing their place in the design hierarchy (e.g., `dut\added_result`). You can shorten the names by going to the Format menu of the Wave window and selecting "Toggle Leaf Names". See Figure 22b.

Note we "push" `LEFT_button` and `result` changes as expected. Also, note `ANDed_result` and `ADDed_result` are always computed even though only one is used. Finally, **without** exiting simulation add the code in Figure 23 before "$stop;". This code will allow us to more comprehensively test the circuit.

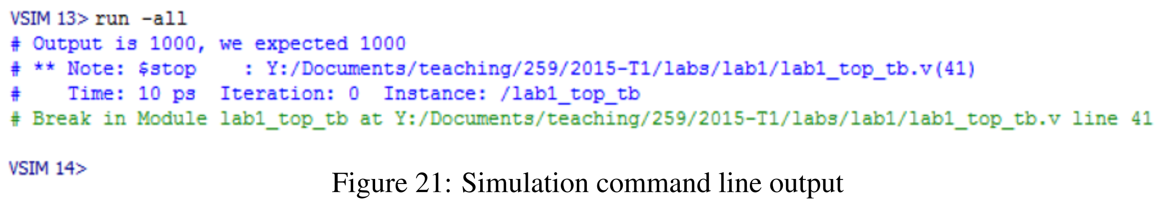
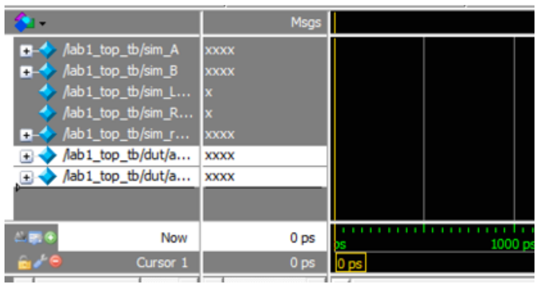
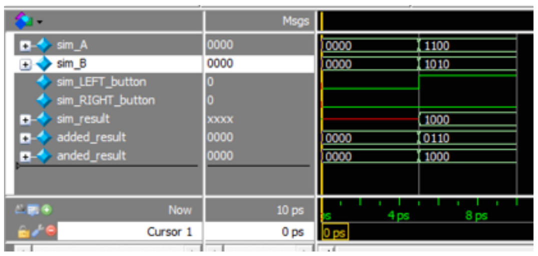

```
VSIM 13> run -all
# Output is 1000, we expected 1000
# ** Note: $stop    : Y:/Documents/teaching/259/2015-T1/labs/lab1/lab1_top_tb.v(41)
#    Time: 10 ps  Iteration: 0  Instance: /lab1_top_tb
# Break in Module lab1_top_tb at Y:/Documents/teaching/259/2015-T1/labs/lab1/lab1_top_tb.v line 41

VSIM 14>
```

Figure 21: Simulation command line output



(a) Waveform before zooming

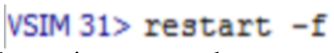


(b) Waveform after zooming and toggling leaf names

Figure 22

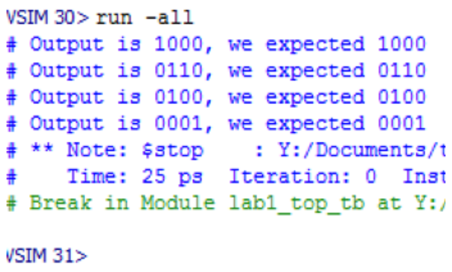
```
// Try adding
sim_LEFT_button = 1'b0;
sim_RIGHT_button = 1'b1;
sim_A = 4'b1100;
sim_B = 4'b1010;
#5;

$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));
```

Figure 23: Rest of the testbench



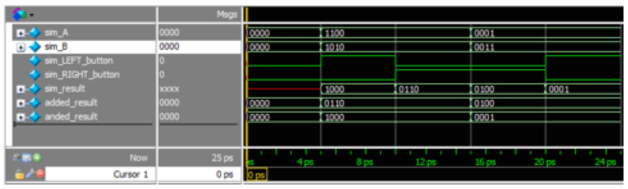
```
VSIM 31> restart -f
```

Figure 24: Transcript output when restarting simulation



```
VSIM 30> run -all
# Output is 1000, we expected 1000
# Output is 0110, we expected 0110
# Output is 0100, we expected 0100
# Output is 0001, we expected 0001
# ** Note: $stop    : Y:/Documents/t
#    Time: 25 ps  Iteration: 0  Inst
# Break in Module lab1_top_tb at Y:/

VSIM 31>
```

(a) Simulation with full testbench



(b) Waves with full testbench

Figure 25

2.3.5 A great HINT

We need to re-compile `lab1_top_tb.v` since we changed it, but we don't need to stop simulating to do that. Save `lab1_top_tb.v` and compile with "Compile -> Compile all". As long as you didn't have any errors you can now restart and re-run your simulation. To do this choose "Simulate -> Restart..." and select "OK" in the "Restart" dialog that appears. Figure 24 illustrates the Transcript pane after doing this. The text command for choosing that menu was "restart -f", so you could just type "restart -f" instead of choosing the menu. That's not much faster, but what is faster is that you will be making lots of changes and recompiling so what you want is a fast way to execute: "restart -f; run -all". In fact, if you type that in it will do the restart and then run your simulation. Even better is if you use the up-arrow key in the Transcript window it will bring you back to the previous command you entered. So, instead of choosing "Simulate -> Run -> Restart", click Restart, then "Simulate -> Run -> Run-all", you can type in "restart -f; run -all" once, and then just press up-arrow, return to do both commands. You'll really appreciate this during Labs 5 to 7!

If you get compile errors use "Simulate -> Start Simulation..." and reload wave.do instead of adding signals manually to the wave viewer. Either way, your results should look like Figure 25a and 25b.

2.4 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. To do this we need to create a new project in Quartus II and add our files to it.

Start Quartus II by clicking on the desktop shortcut created during installation. This should bring up a window like that shown in Figure 26. To create a new project, select "File -> New Project Wizard ..." as shown in Figure 27a. This will bring up a dialog like that shown in Figure 27b. Select "Next" to see the dialog in Figure 28a and enter the working directory, project name (you can use any name "lab1"), and the top level module, which should be "lab1_top" to match the top level module declaration of our synthesizable Verilog. Select "Next" and you should see the dialog in Figure 28b. Select "Next" to get to the dialog in Figure 29a. Here you add ONLY the synthesizable Verilog file "lab1_top.v". Then you need to press the "Add" button. The result should look like Figure 29b. Select "Next" to reach the FPGA device selection dialog pictured in Figure 30. The FPGA on the DE1-SoC boards has part number 5CSEMA5F31C6. To select this part number you can search through the list or, as shown (1) select Cyclone V for Family, (2) FBGA for Package, (3) 896 for pin count, (4) 6 for speed grade then (5) select the part from the shortened list. Select "Next" to reach the EDA Tool Settings dialog box. We won't change any settings here so select "Next" again to reach the summary page. Press "Finish" to create the project.

Next, we must tell Quartus which signals in our top-level module connect to which pins on the FPGA. The FPGA on the DE1-SoC is wired up to numerous peripherals (e.g., pushbuttons, switches and LEDs). The mapping between FPGA pin number and external component on the DE1-SoC board was determined by Terasic when they manufactured the board. However, the mapping between signal names in our design and pin numbers on the FPGA is under our control. Quartus uses a comma separated format (.csv) text file for specifying this mapping. The pin assignments file for lab1 are in "lab1-pins.csv" which is shown in Figure 31a. To load the pin assignments, select "Assignments -> Import Assignments ..." (Figure 31b). Enter the path to "lab1-pins.csv" using the search button (Figure 31c).

Next, select "Processing -> Start Compilation" to compile the design, as illustrated in Figure 32. You will have some warnings as shown in Figure 33. If you select the "Show Warning Messages" button illustrated in Figure 34 (see 1) you will see just the warnings messages. Some of the warnings are unavoidable and relatively harmless. However, you will see a warning about "inferring latch(es)". This warning is important to pay attention to as it is indicative of a common error. This warning means the compiler added hardware, called a latch, and there is a good chance you didn't really want it.

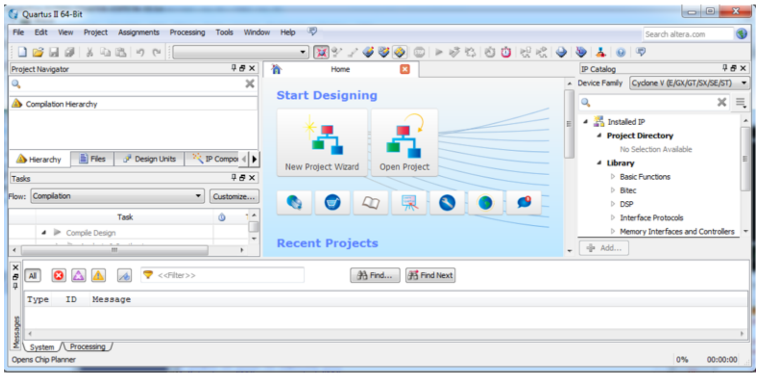
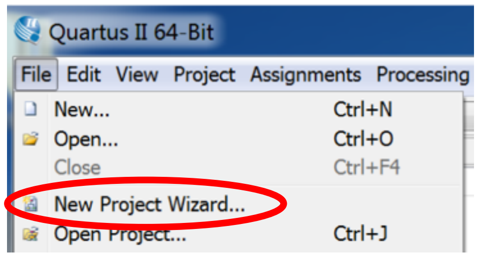
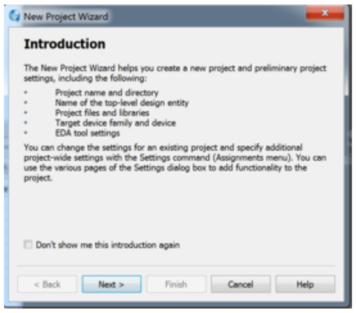


Figure 26: Quartus II



(a) Creating a new project


New Project Wizard

Introduction

The New Project Wizard helps you create a new project and preliminary project settings, including the following:

- Project name and directory
- Name of the top-level design entity
- Project files and libraries
- Target device family and device
- EDA tool settings

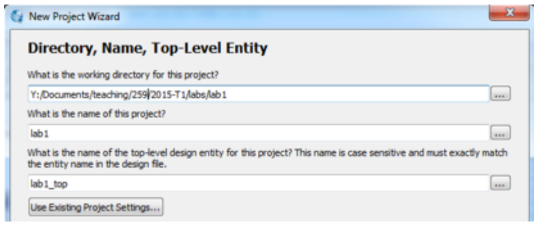
You can change the settings for an existing project and specify additional project-wide settings with the Settings command (Assignments menu). You can use the various pages of the Settings dialog box to add functionality to the project.

☐ Don't show me this introduction again

< Back | Next > | Finish | Cancel | Help

(b) New project dialog

Figure 27


New Project Wizard

Directory, Name, Top-Level Entity

What is the working directory for this project?

Y:/Documents/teaching/259/2015-T1/labs/lab1

What is the name of this project?

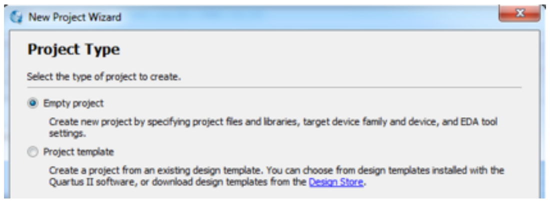
lab1

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

lab1_top

Use Existing Project Settings...

(a) directory, name, top-level module


New Project Wizard

Project Type

Select the type of project to create.

● Empty project

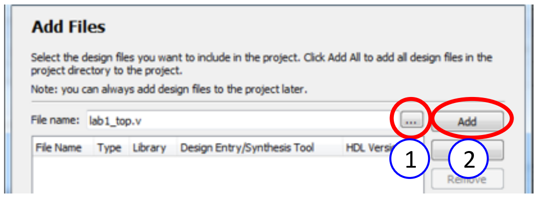
Create new project by specifying project files and libraries, target device family and device, and EDA tool settings.

○ Project template

Create a project from an existing design template. You can choose from design templates installed with the Quartus II software, or download design templates from the Design Store.

(b) Project Type

Figure 28


Add Files

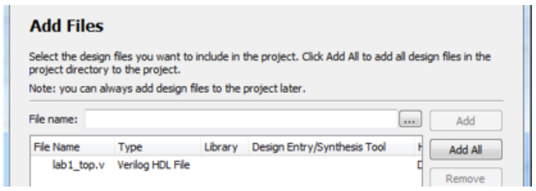
Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

Note: you can always add design files to the project later.

File name: lab1_top.v

File Name	Type	Library	Design Entry/Synthesis Tool	HDL Versi

(a) Add “lab1_top.v” to project. Be sure to browse to find the file by clicking the button with “...” in it (1) - DO NOT type the file name directly; after selecting the file, press Add (2).


Add Files

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

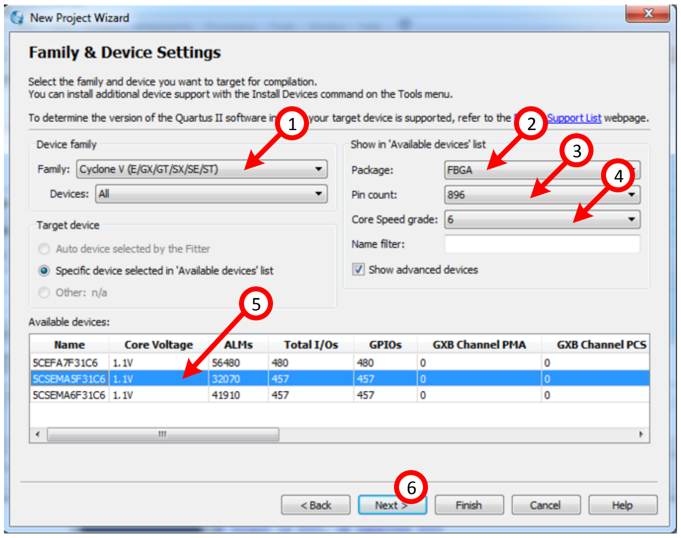
Note: you can always add design files to the project later.

File name:

File Name	Type	Library	Design Entry/Synthesis Tool
lab1_top.v	Verilog HDL File		

(b) After pressing “Add”

Figure 29

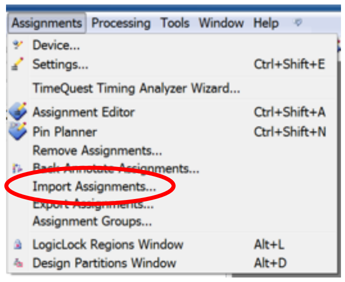

New Project Wizard
Family & Device Settings
Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.
To determine the version of the Quartus II software in which your target device is supported, refer to the Device Support List webpage.
Device family
Family: Cyclone V (E/GX/GT/SX/SE/ST)
Devices: All
Target device
Auto device selected by the Fitter
Specific device selected in 'Available devices' list
Other: n/a
Show in 'Available devices' list
Package: FBGA
Pin count: 896
Core Speed grade: 6
Name filter:
Show advanced devices
Available devices:
1
2
3
4
5
6
< Back
Next >
Finish
Cancel
Help

Name	Core Voltage	ALMs	Total I/Os	GPIOs	GXB Channel PMA	GXB Channel PCS
5CEFA7F31C6	1.1V	56480	480	480	0	0
5CSEMA5F31C6	1.1V	32070	457	457	0	0
5CSEMA6F31C6	1.1V	41910	457	457	0	0

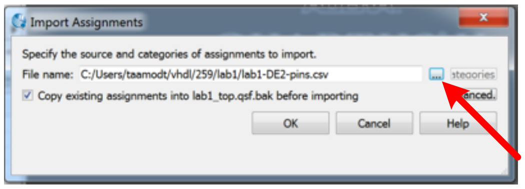
Figure 30: Selecting the 5CSEMA5F31C6 (the FPGA on your DE1-SoC board)

```
To,Location
A[0],PIN_AB12
A[1],PIN_AC12
A[2],PIN_AF9
A[3],PIN_AF10
B[0],PIN_AD11
B[1],PIN_AD12
B[2],PIN_AE11
B[3],PIN_AC9
not_LEFT_pushbutton,PIN_AA15
not_RIGHT_pushbutton,PIN_AA14
result[0],PIN_V16
result[1],PIN_W16
result[2],PIN_V17
result[3],PIN_V18
```

(a) Contents of lab1-pins.csv (please use the file provided to you)



(b) Importing pin assignments


Import Assignments
Specify the source and categories of assignments to import.
File name: C:/Users/taamodt/vhdl/259/lab1/lab1-DE2-pins.csv
Copy existing assignments into lab1_top.qsf.bak before importing
OK
Cancel
Help

(c) specify pin assignments file location

Figure 31

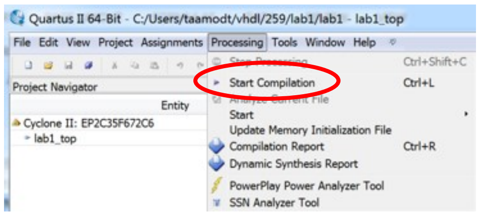
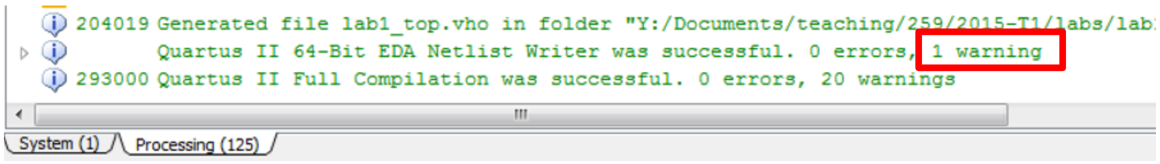
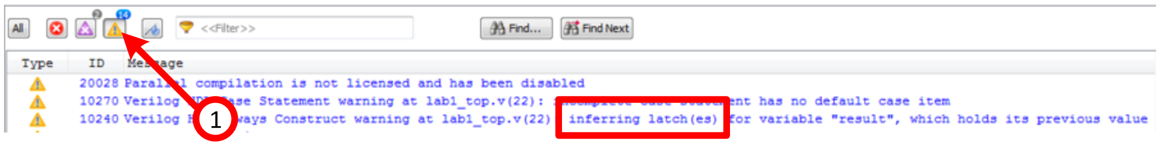

Quartus II 64-Bit - C:/Users/taamodt/vhdl/259/lab1/lab1 - lab1_top
File Edit View Project Assignments Processing Tools Window Help
Project Navigator
Entity
Cyclone II: EP2C35F672C6
lab1_top
Stop Processing Ctrl+Shift+C
Start Compilation Ctrl+L
Analyze Current File
Start
Update Memory Initialization File
Compilation Report Ctrl+R
Dynamic Synthesis Report
PowerPlay Power Analyzer Tool
SSN Analyzer Tool

Figure 32: Starting Compilation



```
204019 Generated file lab1_top.vho in folder "Y:/Documents/teaching/259/2015-T1/labs/lab
       Quartus II 64-Bit EDA Netlist Writer was successful. 0 errors, 1 warning
293000 Quartus II Full Compilation was successful. 0 errors, 20 warnings
```

Figure 33: Warnings are usually bad, check them



```
Type  ID     Message
      20028  Parallel compilation is not licensed and has been disabled
      10270  Verilog HDL Case Statement warning at lab1_top.v(22): incomplete case statement has no default case item
      10240  Verilog HDL Always Construct warning at lab1_top.v(22): inferring latch(es) for variable "result", which holds its previous value
```

Figure 34: An "inferring latch(es)" warning means there is an error in your design.

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

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

(a) Incorrect Verilog creates inferred latch

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

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

(b) Correct Verilog

Figure 35

2.4.1 Inferred Latches

That Quartus "infers" which hardware to build illustrates an important difference between software languages and hardware description languages: With Verilog we only describe the *behavior* of the circuit. The actual implementation is determined by Quartus when it performs an operation called *logic synthesis*. If your Verilog gets an inferred latch warning in Quartus the problem you will later face is your circuit behaving the way you want in some cases, but not in others. In Lab 3 to 7 that can lead to many hours wasted.

The underlying problem is as follows: A "latch" is one of many hardware blocks we will learn about that can "remember things". As we will see, hardware that does not "remember things" is called *combinational logic* (think Snapchat). Now, suppose we want to build a circuit that outputs 1 when an input button is pressed and outputs 0 when the button is not pressed. Consider the *incorrect* Verilog describing this behavior in Figure 35a. Each time `button` changes the **`if`** statement is evaluated. When we press the button, the wire `button` changes from 0 to 1 and `out` will be set to 1. Now, what is the value of `out` if `button` is not 1? Well, if button changes from 1 to 0 then the **`if`** statement condition is now false and so nothing is assigned to `out`. In a real circuit, it doesn't make sense for a wire to have an undefined value. The Verilog compiler resolves this by essentially saying, "I assume you want the circuit to remember the previous value of `out` and use it whenever the current value isn't defined. Since I know there is a special hardware block called a latch that can remember values, I'll use one of those to help me build a circuit that matches your description!" Unfortunately, remembering the old value was *not* what we wanted the circuit to do in this case.

We correct this example by applying a rule you should remember whenever describing combinational logic (hardware that doesn't remember stuff we don't want it to): "For every **`if`** we need an **`else`**. Similarly, for every **`case`** we need a **`default`**. Also for every **`reg`** that is set in one part of an if/else or case must be set in all parts of the if/else or case." Thus, to fix the above code we add the two lines highlighted in Figure 35b.

Let's return to `lab1_top`. The full warning highlighted in Figure 33 is reproduced in Figure 36. Besides the "inferring latch(es)" text, there are two critical pieces of information here. First is file location: lab1_top.v at line 22. The second is the signal involved: `result`. The relevant code with line numbers is shown in Figure 37. The **`case`** statement begins on the line mentioned in the inferred latch warning. Can you see a problem with this case statement? What is the value of `result` if neither button is pressed? As there is no line with `2'b00`, when the case statement is evaluated `result` keeps its prior value. To match this behaviour, Quartus must build hardware to remember the old value of `result` and so it decides to add

```
Warning (10240): Verilog HDL Always Construct warning at lab1_top.v(22):
inferring latch(es) for variable "result", which holds its previous value
in one or more paths through the always construct
```

Figure 36: Full warning message about inferred latches

```
21 always @* begin
22     case( {LEFT_pushbutton, RIGHT_pushbutton} )
23         2'b01: result = ADDed_result;
24         2'b10: result = ANDed_result;
25         2'b11: result = ADDed_result; // Right push button takes precedence
26     endcase
27 end
```

Figure 37: The source of the warning message (line numbers shown on left)

latches. Is this really a problem? It is if the user expects `result` to be `4'b0000` when no button is pressed as indicated by the specification in Section 2.2.1. The inferred latch design error can happen to you due to so-called "selective attention"—paying attention to only a portion of a (combinational logic) specification.

Another way to notice design errors, including inferred latches, is to use a tool in Quartus called the "RTL Viewer" (RTL stands for *register transfer level*). The RTL Viewer let's you see the hardware that Quartus has synthesized based upon your Verilog input. Go to "Tools -> Netlist Viewers -> RTL Viewer" and a window like Figure 38 will open up. Compare it with the specification in Figure 3. The rectangles highlighted in Figure 38 are labeled "LATCH" and are not in our intended hardware design from Figure 3.

Important: In Lab 3 to 7 you want to find (and fix) this problem while testing in ModelSim. Knowing "where to look" for a problem in your Verilog is the secret to keeping these labs fun! Unfortunately, ModelSim does not give an "inferred latch" warning. However, ModelSim *does* enable us to notice and find this design error. Look at the waveform for `sim_result` before `sim_LEFT_button` goes high in Figure 22b. It is *undefined* ("xxxx"), and so we can see it doesn't match our specification in Section 2.2.1. When you see an incorrect signal apply the following *debug technique* to identify the "root cause" *before* changing your Verilog: First, ask "where is the erroneous signal, in this case `sim_result`, set?" Figure 14 shows `sim_result` is connected to the `result` output of the `DUT` instance. Based on this we add `/lab1_top_tb/DUT/result`, which is the signal declared as "**`output reg`** `[3:0] result`" in the `lab1_top` module in Figure 4, to the wave viewer. Then we re-simulate and check whether `/lab1_top_tb/DUT/result` shows the same problem as `/lab1_top_tb/sim_result`. We do this because tracing a "wrong value" to its source *step-by-step* is guaranteed to find the problem. Indeed, an undefined signal can also result from connecting signals incorrectly (e.g., when instatiating a module). You should find these two signals looks the same. So, we have learned that the undefined values come from inside `lab1_top`. Thus, we look at the intended design for `lab1_top` in Figure 3. We see `result` is supposed to be driven by the block labeled MUX. Now, if the inputs to a block are undefined during simulation the outputs will also be undefined. The inputs to the MUX are supposed to be `LEFT_pushbutton`, `RIGHT_pushbutton`, `ANDed_result` and `ADDed_result`. So, we know to add `/lab1_top_tb/DUT/LEFT_pushbutton` and `/lab1_top_tb/DUT/RIGHT_pushbutton` to the wave viewer and resimulate to check if any of the inputs to the MUX are undefined. Doing this, you will see the inputs to the MUX are all defined at time 0 and also that `LEFT_pushbutton` and `RIGHT_pushbutton` are 0. This narrows the problem in both space and time: to the code we wrote for the MUX and when `LEFT_pushbutton` and `RIGHT_pushbutton` are 0. Now we know where in our Verilog code to look for a problem: the always statement used to describe MUX. Looking at this code carefully we will likely notice `result` is not assigned a value when both buttons are not pressed, violating the rule mentioned earlier. A video walking through dealing with syntax and simulation errors is here: https://youtu.be/2c3CZouKJKs.

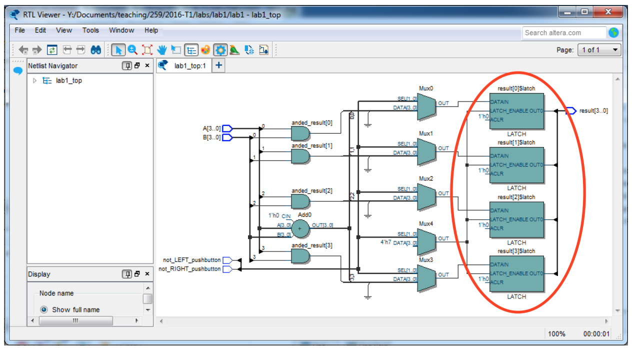


Figure 38: Spotting inferred latches in the RTL Viewer

```
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
        default: result = 4'b0;
    endcase
end
```

Figure 39: The corrected Verilog specifies the value of output `result` of the "combinational logic" multiplexer for *all* inputs so as to eliminate any implied memory (and thus the "inferred latch").

We can eliminate the inferred latches by adding a **`default`** at the end of our case statement to define the value of `result` when no button is pressed (see Figure 39). Recompile and simulate with ModelSim to confirm the red lines go away. Also, recompile in Quartus and verify the inferred latch warning is gone. Look at the circuit again in the RTL Viewer to convince yourself the change works as described.

You should now be ready to download our design to the FPGA on the DE1-SoC to try it there.

2.5 Step 4: Downloading the design to the DE1-SoC

First, attach the power cable to the connector in the top-left of your DE1-SoC board, and the USB cable to the left "BLASTER" port close to the power socket. Power-on the DE1-SoC board by pressing the big red power button underneath the power socket. Your DE1-SoC board should now look like Figure 40.

Now, connect the other end of the USB cable to a USB port on your computer. If you are using the departmental computers, the DE1-SoC USB-Blaster drivers will have been pre-installed for you. However, if you are using your own computer, you should see that it will prompt you to install a driver for the new USB device that you have just attached. Tell it to search in the Quartus install directory, which by default is:

```
C:\altera\15.0\quartus\drivers\usb-blaster-ii
```

Next, in Quartus, select "Tools -> Programmer" as illustrated in Figure 41a. This will open up a window like that shown in Figure 41b. The first time you run the programmer it is likely it will show "No Hardware".

Click on the "Hardware Setup" button in the top left of the Programmer. From the "Currently selected hardware" drop-down box, select "USB-Blaster", as in Figure 42a. If USB-Blaster option is not present, then you may not have installed the drivers correctly: double-check the Device Manager. Additionally, ensure that the USB cable is plugged in to the BLASTER port on the DE1-SoC board, the other end is in a working port on your computer, and that the board is powered on.

Next, click on "Auto Detect" as shown in Figure 42b. Next, in the pop-up dialog, select 5CSEMA5 as shown in Figure 43a. Next, right click the chip symbol labeled 5CSEMA5 and select Edit as shown in Figure 43b (depending upon the version of DE1-SoC you have the chip labeled 5CSEMA5 may be the chip

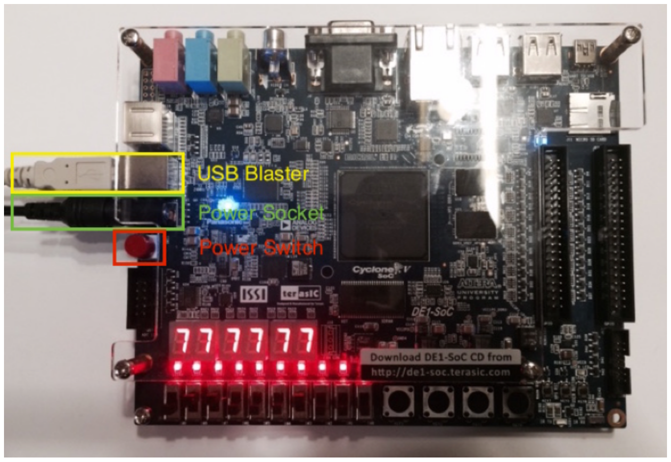
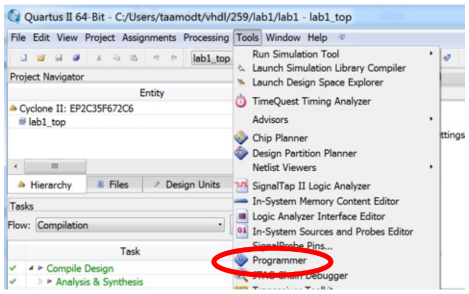


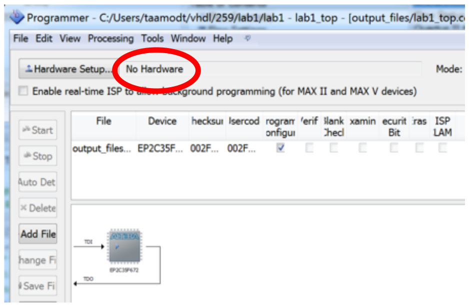
Figure 40: DE1-SoC Board


Quartus II 64-Bit - C:/Users/taamodt/vhdl/259/lab1/lab1 - lab1_top

Cyclone II: EP2C35F672C6

Flow: Compilation
File Edit View Project Assignments Processing Tools Window Help
Project Navigator
Entity
lab1_top
Hierarchy
Files
Design Units
Tasks
Task
Compile Design
Analysis & Synthesis
Run Simulation Tool
Launch Simulation Library Compiler
Launch Design Space Explorer
TimeQuest Timing Analyzer
Advisors
Chip Planner
Design Partition Planner
Netlist Viewers
SignalTap II Logic Analyzer
In-System Memory Content Editor
Logic Analyzer Interface Editor
In-System Sources and Probes Editor
SignalProbe Pins...
Programmer
JTAG Chain Debugger

(a) Opening the device programmer


Programmer - C:/Users/taamodt/vhdl/259/lab1/lab1 - lab1_top - [output_files/lab1_top.c

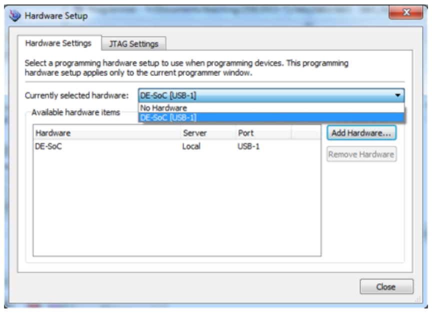
Hardware Setup... No Hardware

Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/Configure	Verify	Blank-Check	Examine	Security Bit	Erase	ISP CLAMP
output_files...	EP2C35F...	002F...	002F...	☑	☐	☐	☐	☐	☐	☐

(b) FPGA Programmer dialog

Figure 41

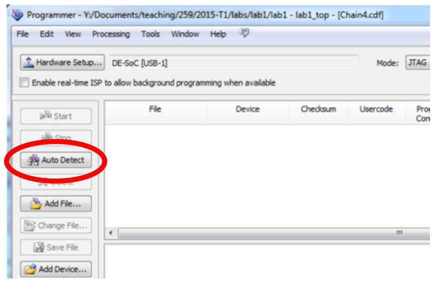

Hardware Setup

Select a programming hardware setup to use when programming devices. This programming hardware setup applies only to the current programmer window.

Currently selected hardware: DE-SoC [USB-1]

Hardware	Server	Port
DE-SoC	Local	USB-1

(a) Selecting USB-Blaster

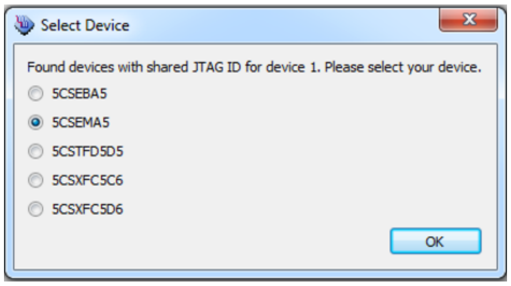

Programmer - Y:/Documents/teaching/259/2015-T1/labs/lab1/lab1 - lab1_top - [Chain4.cdf]

Hardware Setup... DE-SoC [USB-1] Mode: JTAG

Enable real-time ISP to allow background programming when available

(b) Select "Auto Detect"

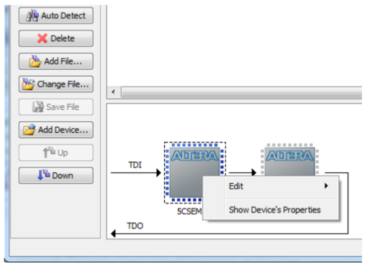
Figure 42


Select Device

Found devices with shared JTAG ID for device 1. Please select your device.

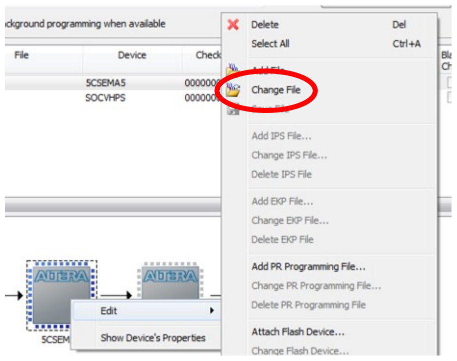
- ○ 5CSEBA5
- [✓ 5CSEMA5]
- ○ 5CSTFD5D5
- ○ 5CSXFC5C6
- ○ 5CSXFC5D6

(a) Select 5CSEMA5

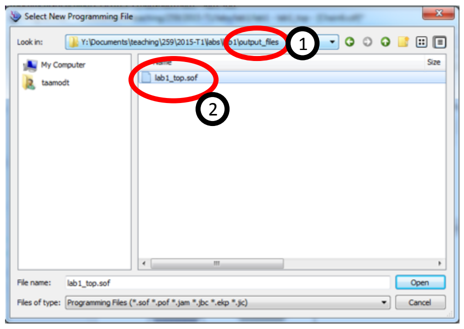


(b) Right-click on FPGA symbol labeled "5CSEMA5" and select "Edit"

Figure 43



(a) then select "Change File"



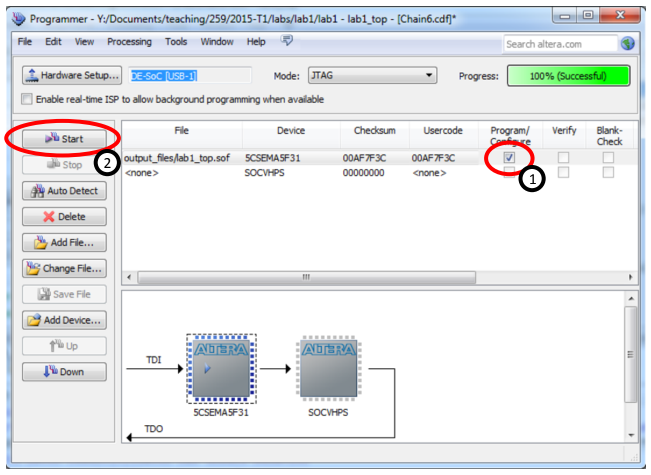
Look in: Y:\Documents\teaching\259\2015-T1\labs\lab1\output_files

File name: lab1_top.sof

Files of type: Programming Files (*.sof *.pof *.jam *.jbc *.ekp *.jic)

(b) Navigate to "output_files" subdirectory and select "lab1_top.sof" file then press "Open"

Figure 44



Hardware Setup: DE-SoC [USB-1] Mode: JTAG Progress: 100% (Successful)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check
output_files/lab1_top.sof	5CSEMA5F31	00AF7F3C	00AF7F3C	✓		
<none>	SOCVHPS	00000000	<none>			

Figure 45: Ensure "Program/Configure" checked then press "Start"

on the left, as shown above, or it may be the chip on the right). Next, select "Change File" in the pop-up menu as shown in Figure 44a. Next, navigate to the subdirectory "output_files" and select "lab1_top.sof" as shown in Figure 44b. The ".sof" file contains a bitstream representation of the design, which results from compiling your Verilog. This bitstream is what is actually used to program the FPGA (we'll learn more about how FPGAs work in class).

Next, ensure the "Program/Configure" check box is selected for the 5CSEMA5F31 then click on the "Start" button in the top left corner of the Programmer as shown in Figure 45. If you do not see "100% (Successful)" in green in the top right consult the debugging hints below.

Once the programmer is done downloading it's time to test out your design! Go ahead and play with your DE1-SoC. The left button has been connected to the pushbutton labeled KEY1 and the right button has been connected to the pushbutton labeled KEY0. A is connected to SW3 through SW0 and B is connected to SW7 through SW4. You should see the results on the 4 LEDs labeled LEDR3 through LEDR0.

If not, here is a handy debug checklist:

- To get some tips from your instructor specifically on debugging syntax and simulation errors you may want to watch the following video: https://youtu.be/2c3CZouKJKs.
- Make sure you circuit works in your ModelSim simulation before trying it on the DE1-SoC.

- Before programming be sure you see the .sof file and “Program/Configure” checked. If it is not, click “Add File” to search for “lab1_top.sof”.
- Try to move all switches from SW0 through SW7 to the “on” position (up) and then press the push button switches KEY0 and/or KEY1. Recall what the circuit is supposed to do (see Table 1 and the corresponding bullet point list in Section 2.2.1).
- Did you remember to load the pin assignments file before compiling your Verilog in Quartus? The pin assignments file (note, this is for Lab 1 only), lab1-pins.csv is available on Piazza.
- Are you using Quartus II version 15.0?
- If you are in MCLD 112 working on an ECE computer, is your Z: drive full? (If it is, Quartus will not be able to create any new files and will crash. Delete a few things to free up disk space and try again.)
- Is the FPGA model set to: Cyclone V 5CSEMA5F31C6? Change using “Assignments -> Device”.
- Have you attached the USB cable to the “BLASTER” port on the DE1-SoC?
- Is the board powered on?
- Is the USB-Blaster driver installed correctly?

3 Marking Scheme

- Demonstrating you can simulate lab1_top.v and lab1_top_tb.v with Modelsim. **[5 marks]**

You will get:	5/5	if you can compile and simulate the design and zoom in the waveform view.
	3/5	if you can compile but don’t know how to simulate your Verilog.
	1/5	if you can start ModelSim but don’t know how to compile or simulate.

- Demonstrating compiling and programming lab1_top.v onto your DE1-SoC. **[4 marks]**

You will get:	4/4	if you compile your Verilog both before and after the bug fix in Figure 39 and demonstrate both versions operating on the DE1-SoC.
	2/4	if you can compile your Verilog in Quartus and program the DE1-SoC successfully but there is something wrong with your circuit or you did not apply the bug fix to the Verilog.
	0/4	if you don’t have a DE1-SoC or don’t know how to program it.

- Explaining the inferred latch problem and how to fix it. For full marks you need to explain two things: (a) what was bad about the simulated behavior in Figure 48 from time 0 to 5 ps before the fix (i.e., instead of “xxxx” what should “result” have been according to the specification in Table 1? What is wrong with “xxxx”?) and (b) why the change in Figure 47 corrects the problem. **[1 mark]**
- **Bonus [1 mark]:** You will need to go beyond what we will have covered in class to get this bonus mark. For a bonus mark you need to understand the operation of the circuit well enough to predict the result (e.g., pattern of LEDs that are on) when any arbitrary values of A, B and pushbutton are input to the circuit. The TA will ask you a few examples.