The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 541 Digital Logic and Computer Design
Spring 2015

Lab #2: Hierarchical Design & Verilog Practice
Issued Wed. 1/14/15; Due Wed. 1/21/15 (11:59pm)

This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided, including screenshots of many of the steps. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission instructions are at the end.
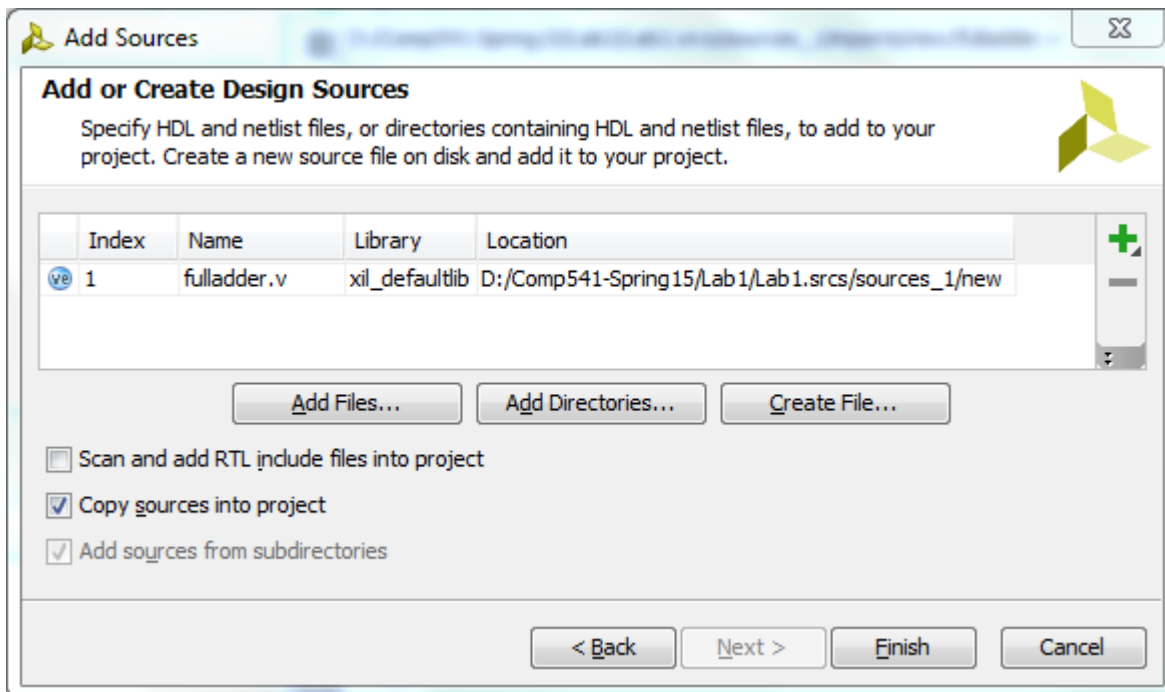
You will learn the following:

- Navigating the ISE development environment

- Designing a hierarchical system, with multiple module types

- Working with buses (multi-bit values)

- Verilog test fixtures and stimuli, including printing and monitoring

- Verilog simulation, including the graphical viewer

In this lab assignment, you will design and test an 8-bit adder-subtractor unit. We will focus on building the design from the bottom up, giving you practice in hierarchical design. We will start with the design of a *full adder,* which will be similar to Lab 1, but with minor modifications. Then, we will string together four full adders to form a 4-bit ripple-carry adder. Next, we will combine two 4-bit adders to produce an 8-bit adder. Finally, we will introduce a conditional negation on the second operand to allow the unit to perform addition or subtraction, depending on a Boolean control input.
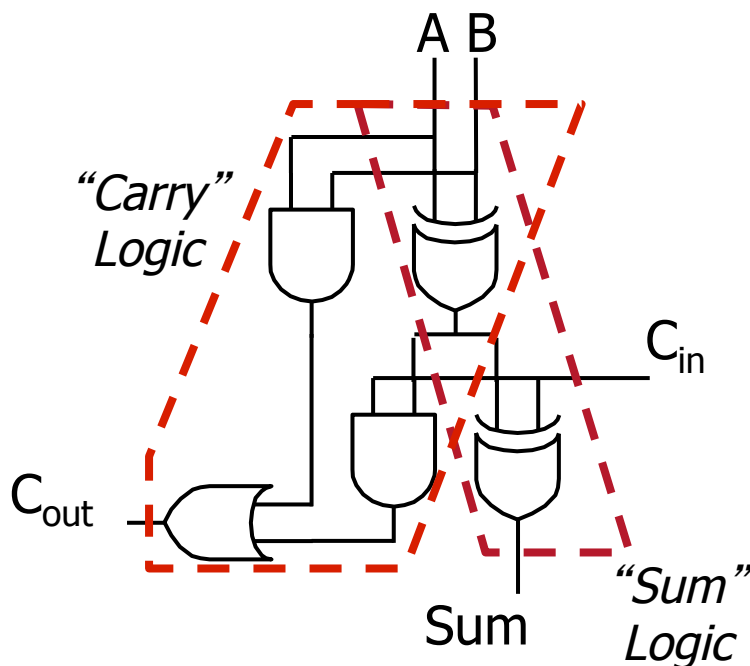
Follow the detailed steps described below.

Make a New Project

Launch Vivado, click *Create New Project,* name it Lab 2, select *RTL Project,* and select the correct part number (xc7a100tcsg324-1). In this empty project, click *Add Sources* → *Add or create design sources* → *Add Files…,* and then enter the path to the *fulladder* Verilog file from Lab 1, as shown in the picture below. *Note:* Be sure to check the box next to *Copy source into project,* so that a new copy of that file is created in this project*;* otherwise, any edits you make to this file will be reflected back in Lab 1!

For your reference, below is once again the circuit and Boolean equations for a Full Adder (from Comp411). In this assignment, we will design the full adder _structurally_ using basic logic gates, exactly following the topology of the circuit diagram, using only 5 gates (instead of the 6-gate implementation of Lab 1).

$$C_{out} = C_{in}(A \oplus B) + AB$$
$$Sum = C_{in} \oplus (A \oplus B)$$

Edit the definition of *fulladder* to make it purely structural as follows. A skeleton of the Verilog description is provided, with some details erased. Your task is to fill in the blanks. Be sure that your Verilog description exactly matches the circuit above.
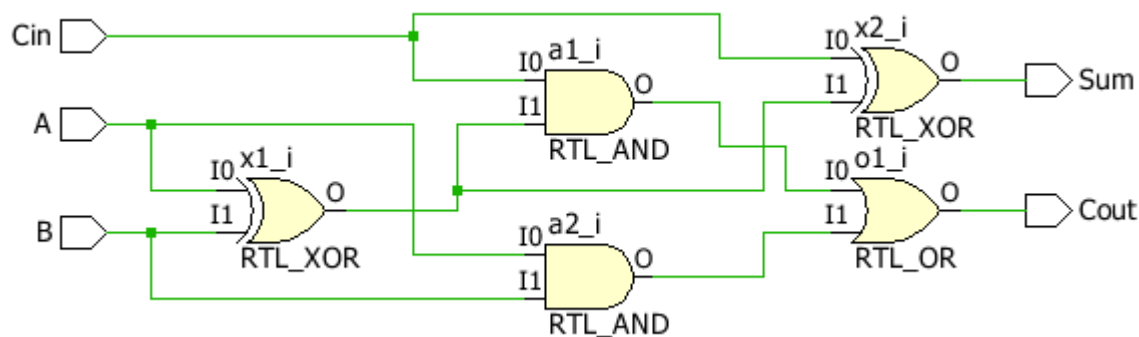
```
module fulladder(
    input A,
    input B,
    input Cin,
    output Sum,
    output Cout
    );

    wire t1, t2, t3;
    xor x1(t1, A, B);
    xor x2(Sum, Cin, ███);
    and a1(t2, █████████);
    and a2(████████ ;
    or  o1(Cout,███████) ;

endmodule
```
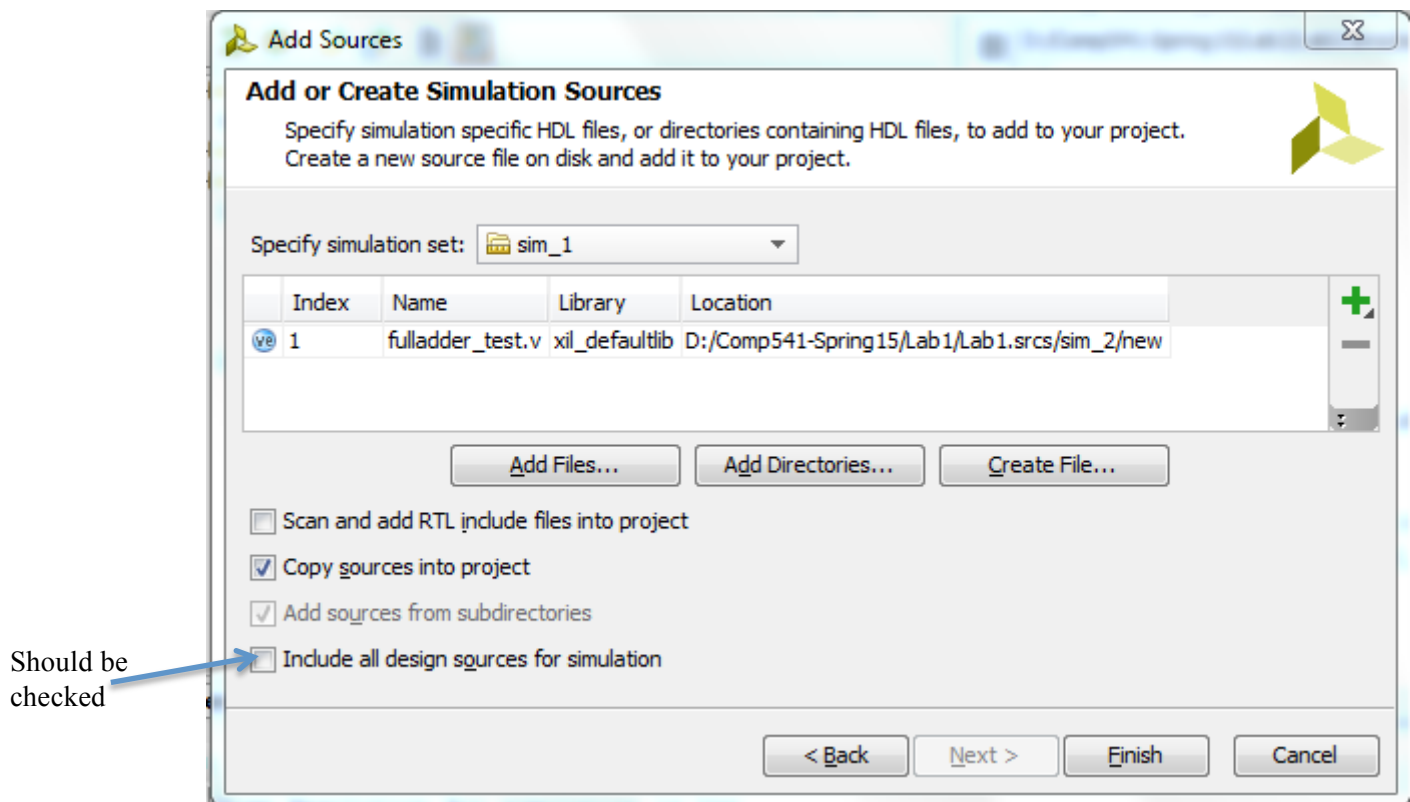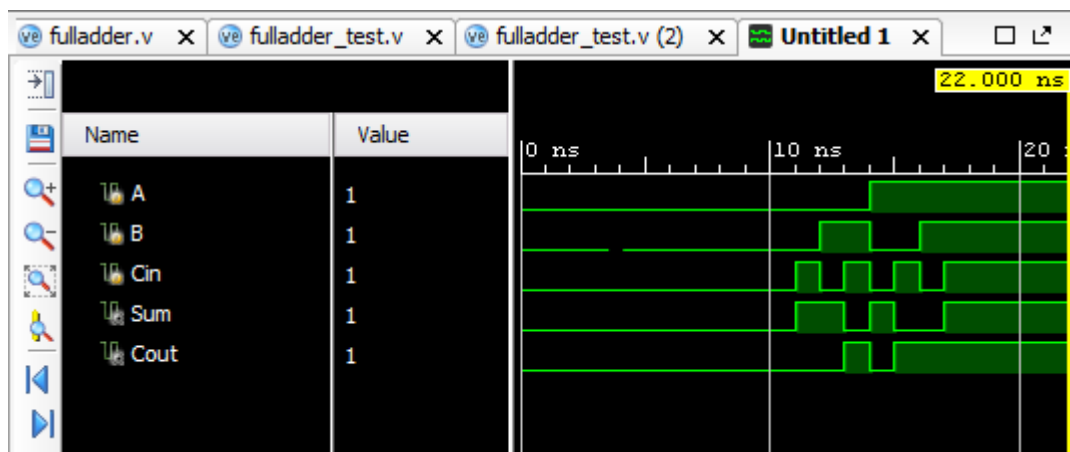
Now, the full adder implementation consists of only 5 gates. Highlight the full adder module under *Design Sources* and click *Elaborated Design* under *RTL Analysis*. You should see the following schematic diagram:

Let us re-use the tester from Lab 1 to test this implementation. Click *Add Sources* → *Add or create simulation sources*. Then, be sure *Copy sources into project* is <u>checked</u>, *Include all design sources for simulation* is <u>checked</u>, and click *Add Files...*, navigate to the path shown in the picture, and select the tester from Lab 1, `fulladder_test.v`.
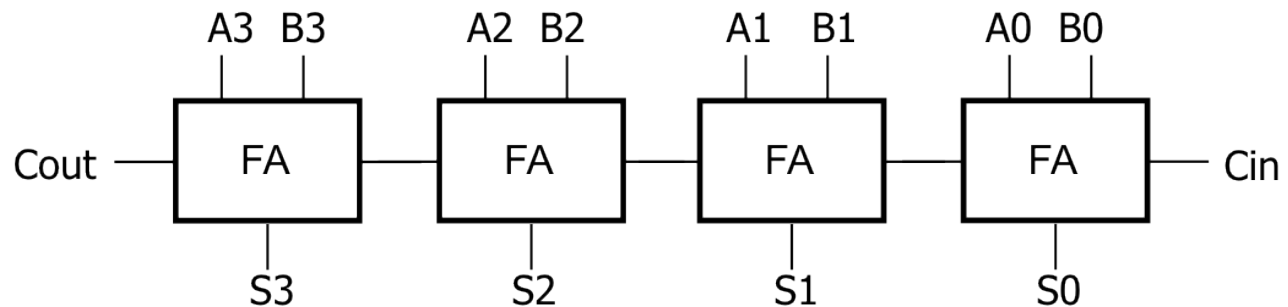
Should be
checked

Now run the simulator and observe the output waveforms. They should be identical to the results from Lab 1, reproduced here:

Designing a 4-bit ripple-carry adder

Let us now design a 4-bit ripple-carry adder by stringing together four full adders (FAs). The diagram of a 4-bit adder (again, from Comp411) is shown here for reference.
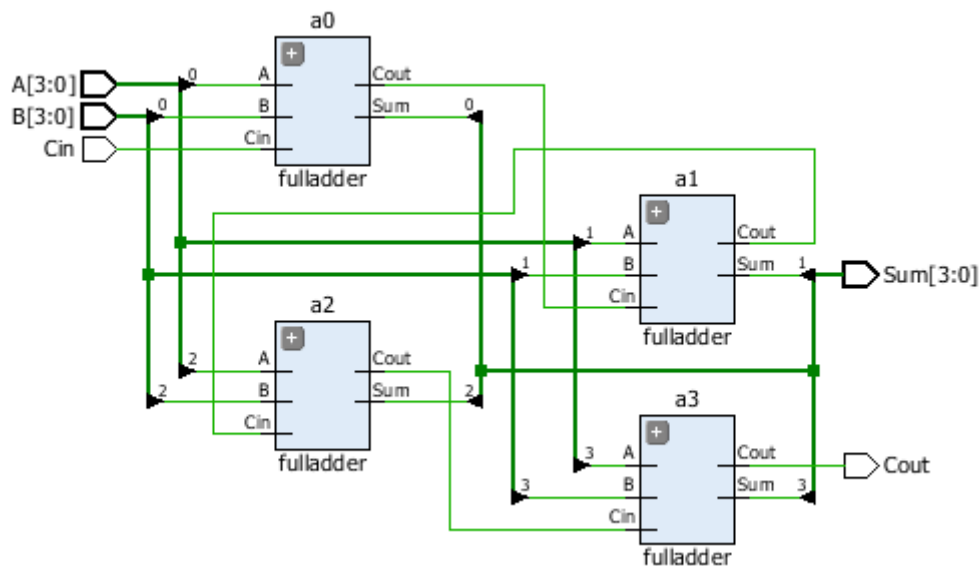

The corresponding Verilog code is shown here, but portions of it have been obscured. Please fill in appropriately. (You do not need to add any extra lines of code; just fill in the missing details into what is provided.)

```verilog
module adder4bit(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] Sum,
    output Cout
    );

    wire C1, C2, C3;
    fulladder a0(A[0], B[0], Cin, Sum[0], C1);
    fulladder a1(A[1], B[1],      , Sum[1],    );
    fulladder a2(A[2], B[2],      , Sum[2],    );
    fulladder a3(A[3], B[3],      , Sum[3],    );

endmodule
```

Before you can enter this code, you will need to create a new source file. Click *Add Sources* → *Add or create design sources,* and name it `adder4bit`.

Generate its schematic (*RTL Analysis* → *Elaborated Design*). It should look like in the picture below. Observe that this circuit diagram is topologicaly equivalent to the block diagram above. Observe how the 4-bit inputs *A[3:0]* and *B[3:0]* are drawn, and the places where individual bits are peeled off (little cone shaped symbols indicate that a single bit is being selected out of a multibit signal). Also observe how individual bits of the output are combined into a multibit *Sum[3:0]* output (with mirror-image cones indicating how single bits feed into a wider signal).

Create your own tester to test your 4-bit adder. Be sure to assign it to a new simulation set (sim_2), and mark it as active (right-click and choose *Make Active*). *Really, do test your 4-bit adder before moving on!*

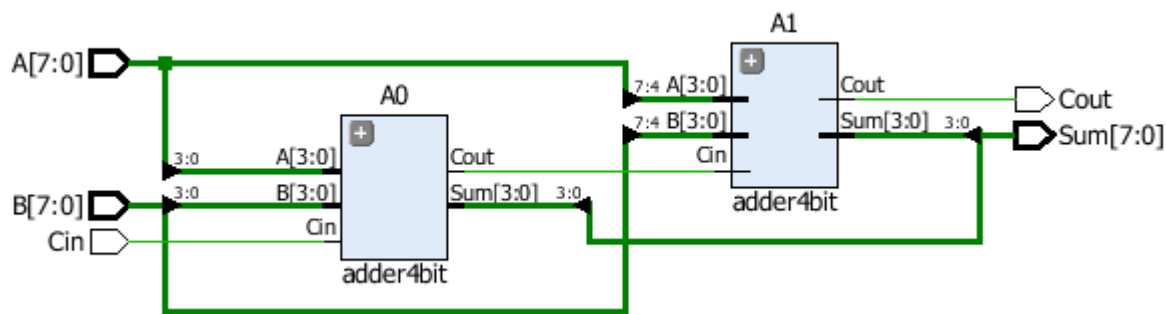**Designing an 8-bit ripple-carry adder**

Now we will design an 8-bit adder using two 4-bit adders. The procedure is very similar: create a new source file, and this time use "adder8bit" as the name for the module. Use the following Verilog skeleton, and fill in the missing details.

```
module adder8bit(
    input [7:0] A,
    input [7:0] B,
    input Cin,
    output [7:0] Sum,
    output Cout
    );

    wire C3;
    adder4bit A0 (A[3:0], B[3:0], Cin, Sum[3:0], ⬤);
    adder4bit A1 (▬▬, ▬▬, ⬤, ▬▬, ▬);

endmodule
```
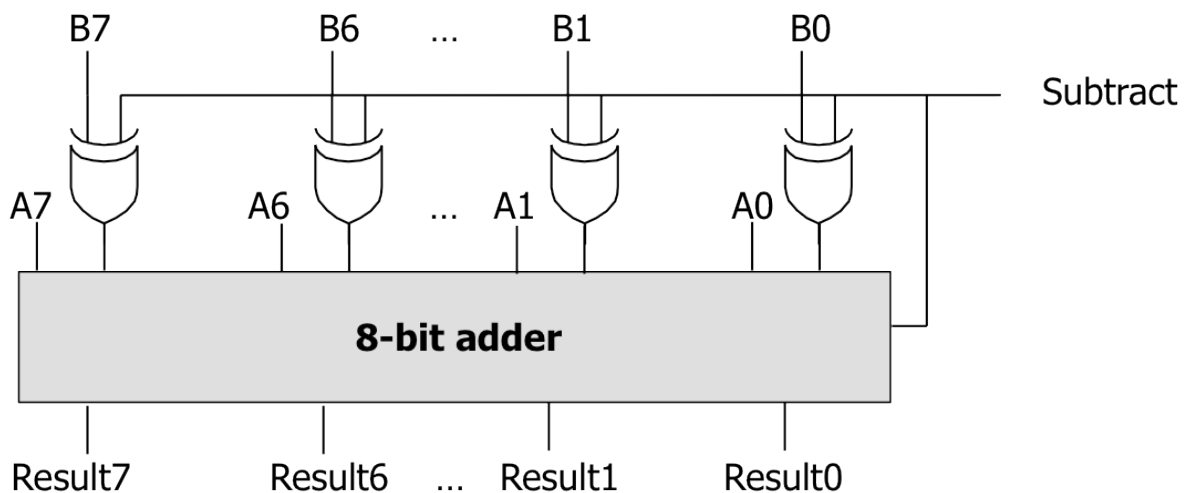
Generate its schematic diagram.

Each module that has a "+" sign on it can be expanded to show its internals (i.e., zoom in and zoom out). Spend a few minutes navigating the hierarchy in this schematic.

Once again, create your own test fixture to test your 8-bit adder. Be sure to assign it to a new simulation set (sim_3), and mark it as active (right-click and choose *Make Active*). *Really, do test it before moving on!*

Designing an 8-bit Adder-Subtractor

Now you will design a circuit that can perform 8-bit additions as well as subtractions. That is, given *A* and *B,* the circuit will produce either the sum *A+B,* or the difference *A-B,* depending on whether the value of a Boolean input *Subtract* is 0 or 1, respectively. This circuit was also covered in Comp411, but is repeated here for reference. Note that there is no C_{in} and no C_{out}.

Once again, you will create a new source file, with the name *add_sub_8bit.* Use the Verilog template below, and fill in the missing pieces:

```
module add_sub_8bit(
    input [7:0] A,
    input [7:0] B,
    input Subtract,
    output [7:0] Result
    );

    wire [7:0] ToBornottoB;
    wire Cout;

    assign ToBornottoB[7:0] = {8{Subtract}} ^ B[7:0];
    adder8bit add8 (A[7:0], ToBornottoB[7:0], ███████, ████████, ████);

endmodule
```
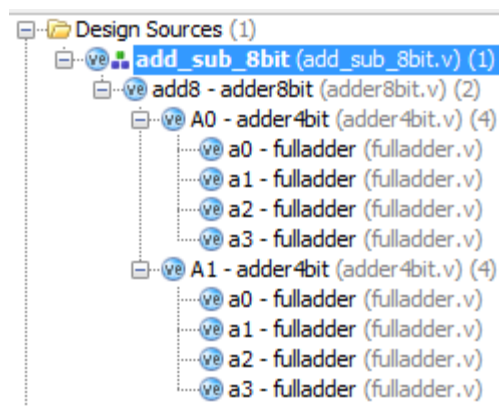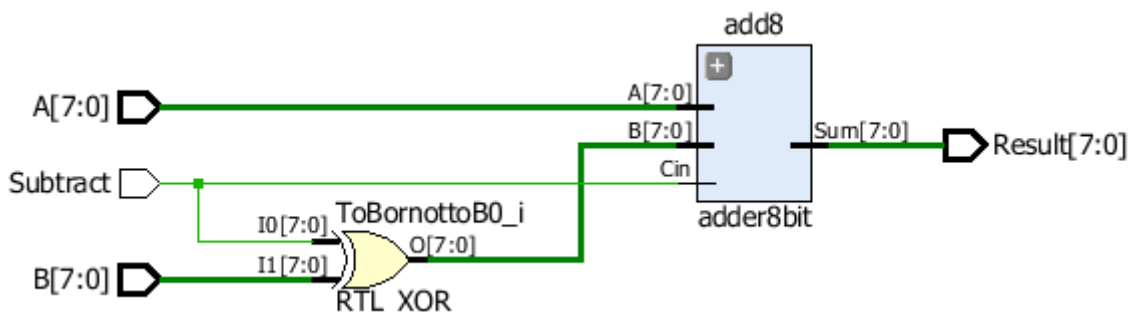
In the Verilog description above, a repetition construct is used: `{8{Subtract}}` simply means, "repeat the value of *Subtract* to produce an 8-bit value that is 00000000 if *Subtract* is 0, and 11111111 if *Subtract* is 1. The operator "^" is a <u>bitwise</u> XOR operator. Therefore, each bit of *B* is XOR'ed with *Subtract,* just as in the circuit diagram above.

Note that while the 8-bit adder has a carry out, the add_sub_8bit module does not send it out! Also, observe carefully what the carry in of the adder is connected to.

Save the file, and take a look at the hierarchy; it should look exactly like this when you expand all the nodes:

Highlight the add_sub_8bit module under *Design Sources,* and generate its schematic.

In this diagram, the single XOR gate represents a bitwise operation on 8-bit inputs. Therefore, it is equivalent to the 8 separate XOR gates.
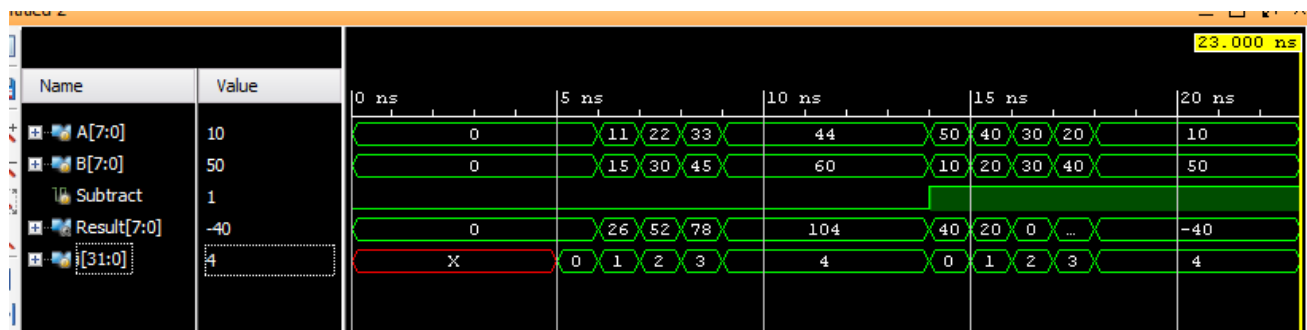
Verilog Test Bench

Click *Add Sources* → *Add or create simulation sources,* and create a new simulation set (sim_4), then create a new tester filer called *addsub_test.* Download the tester from the class website, and copy its contents into the simulation source you just created. (Alternatively, you can choose to *Add Files…* instead of *Create File,* and point to the file you downloaded.)

Read through every line of the test bench, and make sure you understand it! Refer to the online Verilog reference linked from the class website.
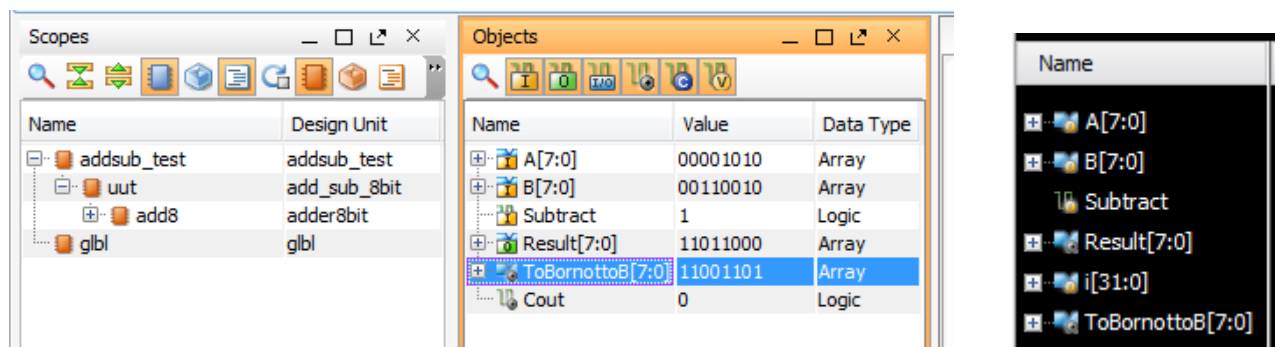
Verilog Simulation

Right-click the simulation set (sim_4) and mark it as active, then run the simulation. Since the default size of the waveform window is too small, click the "pop out" button (⬈), and resize the window and select zoom to fit. Also, since it is hard to make sense of all the 0's and 1's, select all the signals under *Name* (use shift-select), right-click, choose *Radix,* and select *Signed Decimal.* You should now see the simulation outputs in decimal.

Look through them carefully to make sure they are correct.

Now, let us display the bus *ToBornottoB* that is inside the add_sub_8bit module. Under the *Scope* pane, select **uut,** and you will see the objects and wires inside it in the *Objects* pane. Click and drag *ToBornottoB[7:0]* into the *Name* column in the waveform window:

Click on *Run* in the top menu bar, then select *Relaunch Simulation.* When asked if you want to save the waveform configuration, click *Yes* and accept the default file name. This time, once the simulation is completed, the value of *ToBornottoB[7:0]* is also displayed in the waveform window. Right-click *ToBornottoB* in the name column, and change its radix back to *Binary.* Also, change the radix of *B* back to *Binary.* Observe that they are identical for the first half of the simulation (since additions are being performed), and bitwise complements during the second half (since now subtractions are being performed).

This exercise showed you how to examine objects that are not at the top level, but down the hierarchy.

Also, observe that you can click at a particular time instant in the waveform window. The *Value* column is updated to show the values of all the signals at that time instant. There are also other buttons available for zooming in/out, skipping to next transition, etc.

Why does the value of *i* appear as *X* for the first 5 ns?

Using Display and Monitor Commands

Scroll to the bottom of the test fixture. You will see commands using **$time, $timeformat,** and **$monitor.** The **$monitor** command tells the simulator to print a message whenever any of its arguments (except for **$time** itself) changes value. The output appears in *Tcl Console* tab (select able near the bottom of the screen): scroll up a few lines to the see the output of these "print" statements in the tester.

Please refer to the online Verilog reference website for details on these commands, and make sure you understand them well! You should also look into the **$display** and **$write** commands.

What to submit: **A screenshot of the waveform window clearly showing the <u>final simulation result</u> of the adder-subtractor, i.e., with *ToBornottoB[7:0]*.**

- **The values of *ToBornottoB[7:0]* and *B[7:0]* should be shown <u>in binary</u>.**

- **The values of *Result[7:0], A[7:0], Subtract,* and *i* should be shown as <u>signed decimals.</u>**

How to submit:

- **Send email to: `comp541submit-cs@cs.unc.edu`, with "Lab 2" in the subject line.**

- **Attach the simulator screenshot using the filename `waveforms.png` (or other appropriate extension).**

- **Submit your work by 11:59pm on Wednesday, January 21.**