EC311 – Introduction to Logic Design
Laboratory 1 (Lab 1)
Combinational Logic Design: 3-bit ALU
Due 10/4

Introduction
In this laboratory, you will use Verilog to implement a simple Arithmetic Logic Unit (ALU). An ALU is a key component in the "datapath" of most microprocessors. You will learn more about these topics in EC413. The ALU you will make in this lab is significantly less complex than most commercial ALUs but the principles are the same. It takes inputs and performs arithmetic operations on the inputs according to the "mode" it is in.

Here are the specifications of the ALU:
1. Two 3-bit inputs, one "Carry-In" input
2. 3-bits of output, one "Carry-Out" output
3. Four modes controlled by two inputs encoded as listed:
    a. Addition - 00
    b. Subtraction -01
    c. XOR - 10
    d. AND - 11
4. Input should be controlled by the switches on the board.
5. Output should be displayed on the LEDs on the board

Here is a black-box view of the overall ALU module (Figure 1):

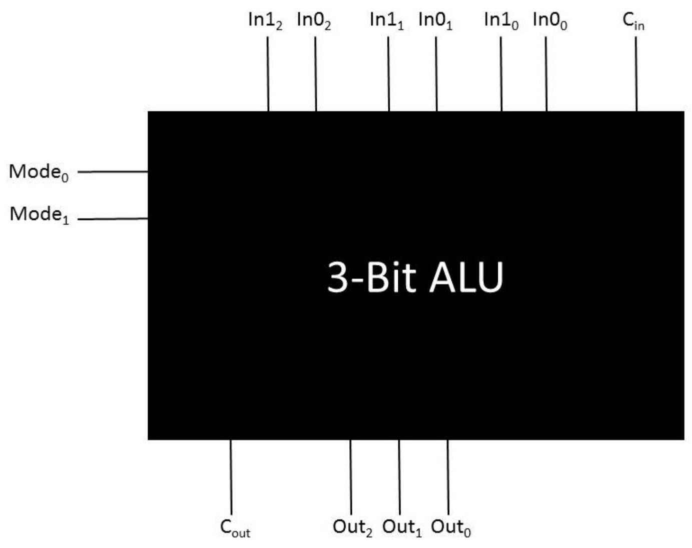


Figure 1: ALU Black Box

The Design

Your design should be PURELY combinational. That means that whenever the inputs change, a change on the outputs is present and possible. This design requires NO state, can be made purely with basic logic gates, and does not need sequential logic elements like flip-flops or latches.

Step 1: Half Adder
The first step is to design a half-adder circuit as shown in Figure 2:

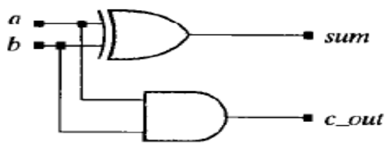
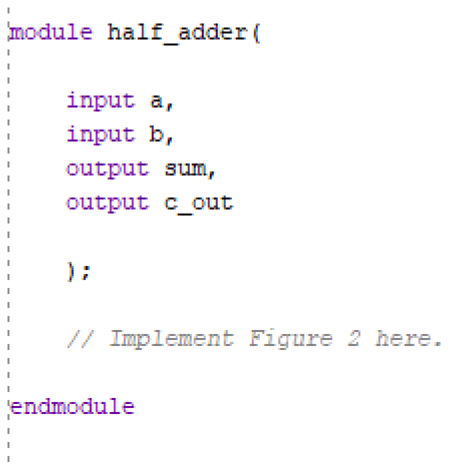


Figure 2: Schematic of a half-adder

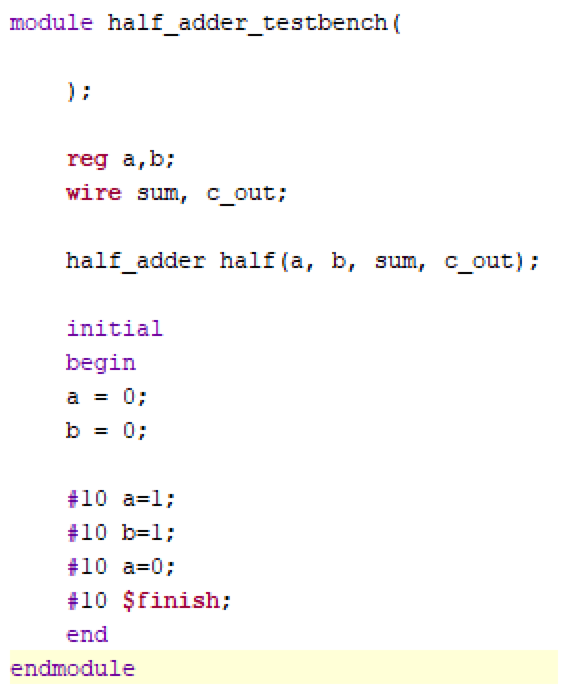
Write the Verilog module for your half adder. You can use structural or behavioral Verilog. It is your choice at this point (if you don't know what this means, ASK!). As you can see, this design allows for adding two 1-bit values without accounting for a "Carry-In".



```
module half_adder(

    input a,
    input b,
    output sum,
    output c_out

    );

    // Implement Figure 2 here.

endmodule
```

Figure 3: Verilog template for half-adder

Write a test module for your half adder. An example is shown below.



```
module half_adder_testbench(

    );

    reg a,b;
    wire sum, c_out;

    half_adder half(a, b, sum, c_out);

    initial
    begin
    a = 0;
    b = 0;

    #10 a=1;
    #10 b=1;
    #10 a=0;
    #10 $finish;
    end
endmodule
```

Figure 4: Example testbench for a half-adder

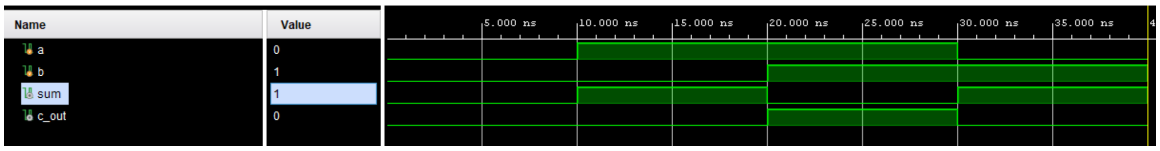


Figure 5: Example waveform for a half-adder

Simulate the circuit and verify the functionality of the half-adder by observing the waveforms. **Do not proceed further until you are confident that your half adder is working correctly**.

Once you have a fully functional half-adder module, which you will have if you have followed the above steps without any errors, you will use the half-adder module hierarchically to build and test a full adder.

Step 2: Full Adder
Now create a 1-bit “Full Adder”

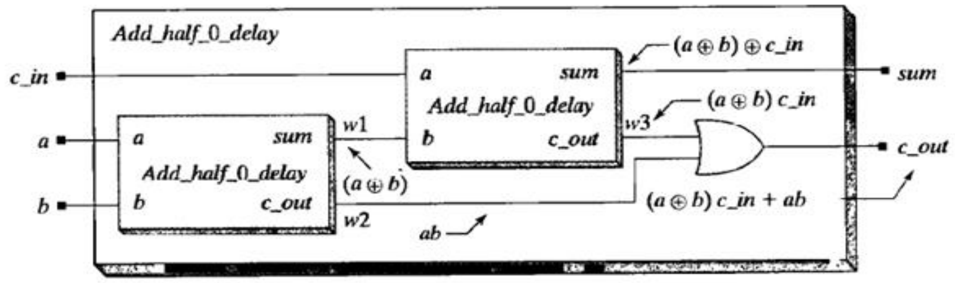
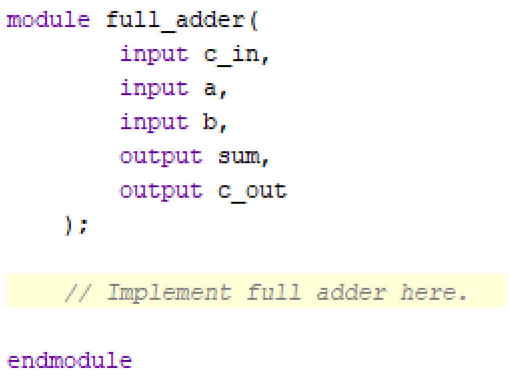


Figure 6: 1-bit Full Adder Diagram



```
module full_adder(
        input c_in,
        input a,
        input b,
        output sum,
        output c_out
    );

    // Implement full adder here.

endmodule
```

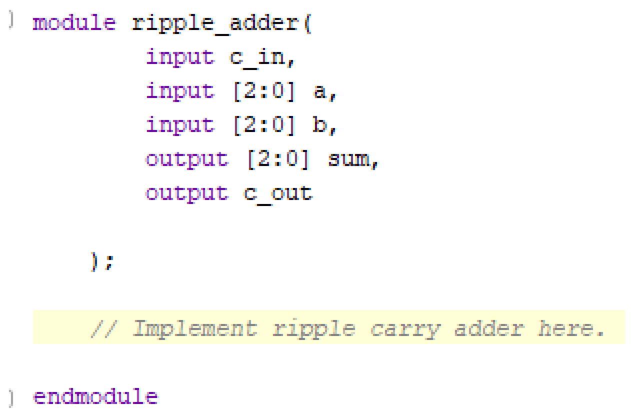
Figure 7: Verilog template for 1-bit full-adder

Create both the Verilog for the module and testbench for your 1-bit full adder. **Do not proceed to the next step until you are confident that your 1-bit full adder works.**

Step 3: Ripple Carry Adder

You are now going to “chain together” these 1-bit full adders to make larger adders for more bits. This design is called a “Ripple Carry Adder”. Think about where it gets its name from.

Create Verilog to chain your full adders together to create a 3-bit “ripple carry adder”.



```
module ripple_adder(
        input c_in,
        input [2:0] a,
        input [2:0] b,
        output [2:0] sum,
        output c_out

    );

    // Implement ripple carry adder here.

endmodule
```

Figure 8: Verilog template for 3-bit ripple carry adder. *Note:* `input[2:0] a` *means that "a" is an array of three bits. a[0] is the LSB and a[2] is the MSB.*

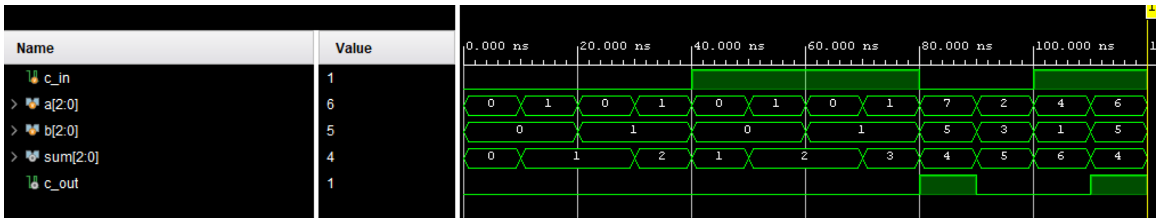


Figure 9: Example testbench simulation of a 3-bit ripple carry adder. Your simulation does not need to look exactly like this, but you should test the module as exhaustively as possible.

Create both the Verilog for the module and testbench for your 3-bit "ripple carry adder". **Do not proceed to the next step until you are confident that your 3-bit ripple carry adder works.**

Step 4: Push to the Board

Now you need to take your Ripple Carry adder and "push" your design to the board along with connecting the outputs to LEDs and the inputs to switches.

Once you are done, you should be able to actually test your design on the board and confirm that it performs as you expect. **Do not proceed to the next step until you are confident that your 3-bit ripple carry adder work ON THE BOARD using the switches and LEDs.**

Step 5: Extend your adder to do subtraction

In this portion of the lab you will extend your adder to do 2's complement subtraction. Remember that the way in which you can subtract the second number from the first is to flip the second number's bits and add one. So you need to:

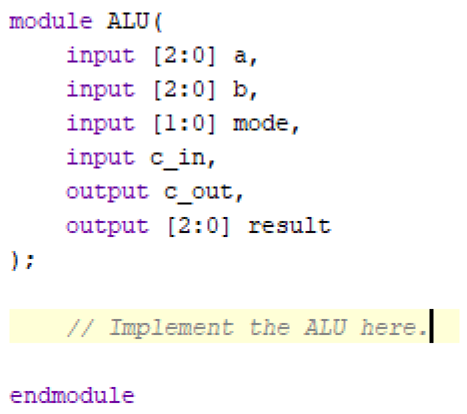
1. Add additional inputs to the Verilog module for the mode.
2. Create a mechanism to invert all of the second input's bits if the mode is subtract (01)
3. Create a mechanism to "add 1" at the same time.
   a. The "secret" here is that you can do this with the "carry-in" bit of the ALU. If this is set to 1, this effectively will do this for you. Try it out.

To do much of this you can use MUXes. More on MUXes will be posted to Piazza if needed. Feel free to google them as well. They are very simple combinational logic devices that select from multiple inputs and send them to the output.

Once you are done, you should be able to actually test your design on the board and confirm that it performs as you expect. **Do not proceed to the next step until you are confident that your 3-bit ripple carry adder can now also do subtraction ON THE BOARD using the switches and LEDs.**

Step 6: Finish off the ALU

Now you need to add mechanisms to support both XOR and AND. Think about the circuit elements you already have in the 1-bit half adder and think how you can just use them in the event that the modes are set to either XOR or AND. Again, MUXes will be very useful here.



```
module ALU(
    input [2:0] a,
    input [2:0] b,
    input [1:0] mode,
    input c_in,
    output c_out,
    output [2:0] result
);

    // Implement the ALU here.

endmodule
```

Figure 10. Verilog template for ALU.

This is it. Get all of this to work using a set of test benches that tests all the modes with a variety of inputs. Make sure you understand what you have done. Lab 2 will build on this ALU.

General Comments

You will find that it is much easier to implement your design in Verilog because the Xilinx synthesis tools will automatically figure out how to minimize your logic expressions and implement them with gates. In other words, you will not have to draw Karnaugh maps for all of your logic expressions, nor will you have to tediously place all of the necessary gates with the schematic editor tool. In fact, logic designers in industry today rarely use the schematic editor; they instead implement their designs almost exclusively with a hardware description language (HDL) such as Verilog or VHDL. Schematic editors are used primarily to insert last minute additions or to create highly optimized or specialized circuits. Logic designers save a tremendous amount of effort and time by letting the synthesis tools handle the tedious details of converting Verilog to logic gates.

There are many different ways to describe your circuit in Verilog. For example, you could implement the negation operation using a single expression which inverts all of the bits and adds one (i.e., "assign myoutput = ~(myinput)+1;" where myinput and myoutput are 4-bit values). Alternatively, you could implement the negation operation using a case statement in which each case corresponds to a line from your truth table for the negation operation. Another option is simply writing boolean expressions for each of your output bits. The logic synthesis tools may or may not generate the same logic circuit depending on which way you decide to describe your design in your Verilog code. Naturally, they will be equivalent functionally but may have different timing, power, and area metrics. Deciding which method is most convenient in certain situations is something that you will learn only through lots of practice. Good Verilog coders understand how the tools work in-depth to create the most efficient final designs.

What to turn in (ZIP 1-4 together):

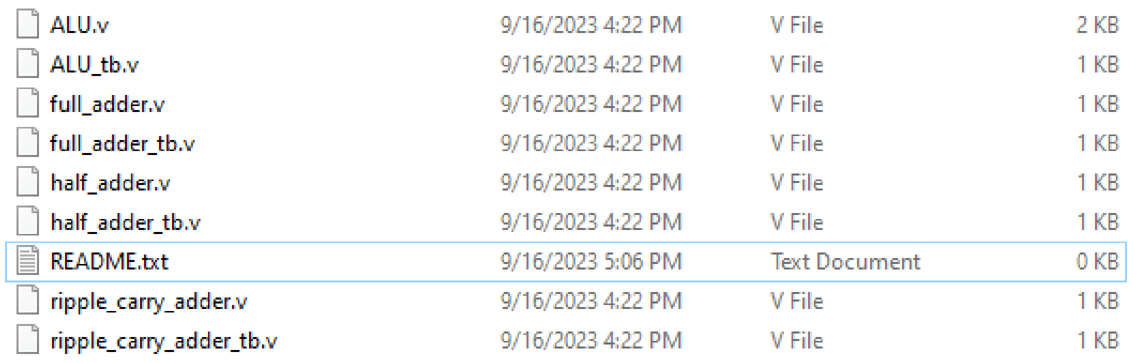
1. A README.txt file that explains your submission, its organization, and includes both team member's names and UIDs. Feel free to include any other comments you want your TA to see that will be relevant while they are grading the assignment.
2. Verilog files (.v) for half adder, 1-bit full adder, 3-bit ripple carry adder, and full ALU. Zip these together in an organized way and use good naming conventions.
3. Testbench files for each of the modules in #2. Feel free to have more if needed and explain in your README.
4. Write a short discussion about your ALU (and save as a pdf). What do you like about it? What are some of its shortcomings (think about how many numbers you can represent)? Is it fast? Slow? Big? Small? Let us know your thoughts!
5. Demo your design to your TA. Arrange a time. The lab would be ideal but that may not be possible. OH work as well. If necessary, you can record yourself using the FPGA and provide that video to your TA if needed.

Help

Read pages 572-575 in the textbook for more on adder design

Read pages 182-189 in the textbook for more on MUX design

This reference will be helpful to understand Verilog and module instantiation better: http://www.asic-world.com/verilog/syntax2.html



ALU.v	9/16/2023 4:22 PM	V File	2 KB
ALU_tb.v	9/16/2023 4:22 PM	V File	1 KB
full_adder.v	9/16/2023 4:22 PM	V File	1 KB
full_adder_tb.v	9/16/2023 4:22 PM	V File	1 KB
half_adder.v	9/16/2023 4:22 PM	V File	1 KB
half_adder_tb.v	9/16/2023 4:22 PM	V File	1 KB
README.txt	9/16/2023 5:06 PM	Text Document	0 KB
ripple_carry_adder.v	9/16/2023 4:22 PM	V File	1 KB
ripple_carry_adder_tb.v	9/16/2023 4:22 PM	V File	1 KB

Figure 11. Example submission (contents of zip file).